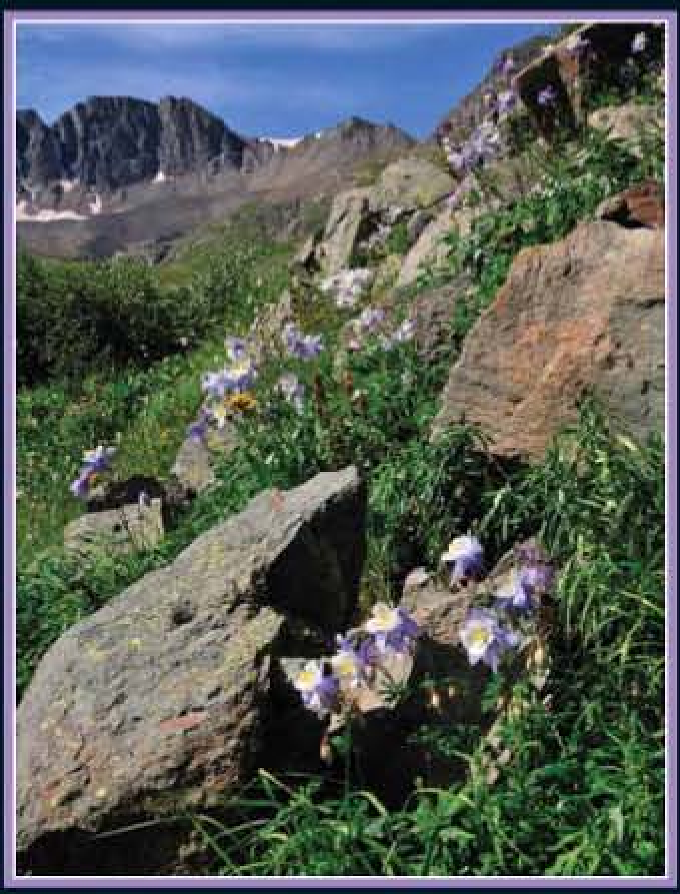
Advanced Digital Design with the Verilog HDL

Second Edition



Michael D. Ciletti

Featuring Verilog Updates

Preface

mudassarsardar.blogspot.com

Simplify, Clarify, and Verify

Behavioral modeling with a hardware description language (HDL) is the key to modern design of application-specific integrated circuits (ASICs). Today, most designers use an HDL-based design method to create a high-level, language-based, abstract description of a circuit, and verify its functionality and timing. The language used to teach design methodology in the first edition of this text, IEEE 1464-1995, has undergone two revisions to improve the effectiveness and efficiency of the language: IEEE 1364-2001 followed by a revision in 2005, known as Verilog-2001 and Verilog-2005, respectively.

The motivation behind this edition is basically the same as that which guided the first edition: students preparing to contribute to a productive design team must know how to use a HDL at key stages of the design flow. Thus, there is a need for a course going beyond the basic principles and methods learned in a first course in digital design. This book is written for such a course.

The quantity of books discussing HDLs far exceeds that which was available at the time of the first edition, and most of these are still oriented toward explanations of language syntax, rather than toward design, and are not well-suited for classroom use. Our focus is on design methodology enabled by an HDL. Thus, the language itself has a subordinate role. In this edition, we have made a strong effort to demonstrate by examples the importance of partitioning a digital machine to expose its datapath, status (feedback) signals, and controller (finite state machine). This effort leads, we think, to a much clearer and straightforward approach to designing and verifying complex digital machines. We present an abundance of simulation results, with annotation to help students (1) understand the operation of a sequential machine and (2) appreciate the time-sequential interaction between the signals produced by the controller, the operations in the datapath, and the signals reported back to the controller from the datapath, all with the aim of developing synthesizable, latch-free, race-free designs.

The language enhancements of Verilog 2001, 2005 have been used to reexamine and simplify the code of our models. We emphasize industry practices, and do not unwittingly encourage academic styles of modeling without regard for whether a model can be synthesized. Solutions to selected problems, together with additional solved exercises that are not included in the text, are available for instructors at the companion Web site: http://www.pearsonhighered.com/ciletti. The first edition treated synchronous FIFOs; this edition treats synchronous and asynchronous FIFOs, and presents

an elaborate example of using an asynchronous FIFO to synchronize transfer of data across clock domains.

Design practice is always in a flux. One tension is between the approach of writing a model in C for an embedded controller versus writing a model in Verilog. The C-based approach executes statements; Verilog models execution of multiple concurrent behaviors of a machine. The latter approach compiles hardware; the former compiles statements for a preexisting hardware unit. The compiled hardware of the Verilog model produces equivalent I/O signals at the interface to the host machine for a particular application. The distinction here is that embedded code does not equal embedded hardware. This text aims to teach the hardware modeling/compilation paradigm and to anticipate the results of synthesis. Programming in C anticipates the data produced by a program, and the machine/processor is itself transparent. In contrast, modeling in Verilog anticipates the hardware that will produce the I/O signals demanded by the application, and requires the developer to think in terms of time-sequential control of register operations. Verilog modeling encourages this understanding about the nature of a digital machine.

Our goal in this book is to build on a student's background from a first course in logic design by (1) briefly reviewing basic principles of combinational and sequential logic, (2) introducing the use of HDLs in design, (3) emphasizing descriptive styles that will allow the reader to quickly design working circuits suitable for ASICs and/or field-programmable gate array (FPGA) implementation, and (4) providing design examples having a range of difficulty. The end-of-chapter problems encourage students to simplify, clarify, and verify their designs. The nature of many of the problems is that of open-ended design, requiring verification that the design meets a prescribed specification.

The widely used Verilog HDL (IEEE Standard 1364) serves as a common framework supporting the design activities treated in this book. The first edition focused on developing, verifying, and synthesizing designs of digital circuits, and *not* on the Verilog language. This edition maintains that focus.

Most students taking a second course in digital design will be familiar with at least one programming language and will be able to draw on that background in reading this text. We cover only the core and most widely used features of Verilog. In order to emphasize *using* the language in a synthesis-oriented design environment, we have purposely placed many details, features, and explanations of syntax in the appendices and at the publisher's Web site for reference on an "as needed" basis, but an appendix provides the complete formal syntax of the language.

Most entry-level courses in digital design introduce finite state machines using state transition diagrams, and algorithmic state machine (ASM) charts. We do likewise, but we make heavier use of ASM charts and demonstrate their utility in developing behavioral models of sequential machines. The important problem of systematically designing a finite state machine to control a datapath in a complex digital machine is treated in-depth with ASMD charts, i.e. ASM charts annotated to display the register operations of the controlled datapath. The design of a RISC CPU and other important hardware units are given as examples. Our companion Web site includes the RISC machine's source code and an assembler that can be used to develop programs for applications. The machine also serves as a starting point for developing a more robust instruction set and architectural variants.

The Verilog language is introduced in an integrated, but selective manner, only as needed to support design examples. The text has a large set of examples illustrating how to address the key steps in a VLSI circuit design methodology using the Verilog HDL. The source code of the examples has been verified to be correct. Source code for all of the examples and their testbenches are available at the publisher's Web site.

The Intended Audience

This book is for students in an advanced course in digital design, and for professional engineers interested in learning Verilog by example, in the context of its use in the design flow of modern integrated circuits. The level of presentation is appropriate for seniors and first-year graduate students in electrical engineering, computer engineering, and computer science, as well as for professional engineers who have had an introductory course in logic design. The book presumes that the reader has a basic background in Boolean algebra and its use in logic circuit design, and a familiarity with synchronous finite state machines. Building on this foundation, the book addresses the design of several important circuits used in computer systems, digital signal processing, image processing, data transfer across clock domains, built-in self-test (BIST) and, other applications. The examples cover the key design problems of modeling, architectural trade-offs, pipelining, multiprocessor implementations, functional verification, timing analysis, test generation, fault simulation, design for testability, logic synthesis, and post-synthesis verification.

What's New in this Edition

- Exploits key features of Verilog 2001, 2005
- Illustrates and promotes a synthesis-ready style of register transfer level (RTL) and algorithmic modeling with Verilog 2001, 2005
- Provides an in-depth treatment of algorithms and architectures for digital machines (e.g. an image processor, digital filters, and circular buffers) with Verilog 2001, 2005
- Includes comprehensive design examples (e.g. a RISC machine and various datapath controllers) with Verilog 2001, 2005
- Includes numerous annotated and explained graphical illustrations of simulation results
- Contains over 150 fully verified examples with Verilog 2001, 2005
- Contains a worked example with JTAG and BIST for testing with Verilog 2001, 2005
- Contains an Appendix with full formal syntax of the Verilog 2001, 2005 HDL
- Treats asynchronous and synchronous FIFOs

Special Features of the Book

- Provides a brief review of basic principles in combinational and sequential logic
- Focuses on modern digital design methodology
- Demonstrates the utility of ASM and ASMD charts for behavioral modeling
- Clearly distinguishes between synthesizable and nonsynthesizable loops
- Provides practical treatment of timing analysis, fault simulation, testing, and design for testability, with examples
- Provides several problems with a wide range of difficulty after each chapter
- Combines a solution manual with an on-line repository of additional worked exercises
- Lists an index of all models developed in the examples
- Includes a set of FPGA-based, lab-ready exercises linked to the book (e.g. arithmetic and logic unit (ALU), programmable lock, a keypad scanner with a FIFO, a serial communications link with error correction, an SRAM controller, and first in, first out (FIFO) memory, RISC CPU, and FIFO)
- Supported by an ongoing Companion Web site containing:
  1. Source files of all models developed in the examples
  2. Source files of testbenches for simulating all of the examples
  3. An Instructor's Classroom Kit containing transparency files for a complete course based on the subject matter is available for instructors only

4. Solutions to selected problems is available for instructors only
5. Additional worked problems
6. Jump-start tutorials helping students get immediate results with selected software tools (e.g. simulator)
7. Answers to frequently asked questions (FAQs)

Sequences for Course Presentation

The material in the text begins with a brief review of combinational and sequential logic design, but then progresses in the order dictated by the design flow for an ASIC or an FPGA. Chapters 1 to 6 treat design topics through synthesis, and should be covered in order, but Chapters 7 to 10 can be covered in any order. The homework exercises are challenging, and the lab-ready FPGA-based exercises are suitable for a companion lab or for end-of-semester projects. Chapter 10 presents several architectures for arithmetic operations, affording a diversity of coverage. Chapter 11 treats post-synthesis design validation, timing analysis, fault simulation, and design for testability. The coverage of these topics can be omitted, depending on the level and focus of the course.

Some Caveats

We do not adhere to common practice for using upper and lower case text, or for using courier font in code listings. Our choices have been based on maximizing the overall visual appeal and readability of the listed code. The visual result offsets, we think, the extra care required to ensure that the code is composed correctly in our examples. Block diagrams have been simplified to reduce the visual clutter, so we typically show only the actual, external names of signals, and omit their formal, internal counterparts. D-type flip-flops are used almost exclusively because they play a dominant role in synthesis with modern EDA tools.

Chapter Descriptions

Chapter 1 briefly discusses the role of HDLs in design flows for cell-based ASICs and FPGAs. Chapters 2 and 3 review mainstream topics that would be covered in a first course in digital design, using classical methods, i.e. Karnaugh maps. This material will refresh the reader's background, and the examples will be used later to introduce HDL-based methods of design. Chapters 4 and 5 introduce modeling of combinational and sequential logic with the Verilog HDL, and place emphasis on coding styles that are used in behavioral modeling. Chapter 6 addresses cell-based synthesis of ASICs, and introduces synthesis of combinational and sequential logic. Here we pursue two main objectives: (1) present synthesis-friendly coding styles and (2) form a foundation that will enable the reader to anticipate the results of synthesis, especially when synthesizing sequential machines. Many sequential machines are partitioned into a datapath and a controller. Chapter 7 covers examples that illustrate how to design a controller for a datapath, including machines having feedback of status signals from the datapath to the controller. The designs of a simple RISC CPU and a UART[1] serve as platforms for the subject matter.

[1]Universal Asynchronous Receiver and Transmitter (UART), a circuit used in data transmission between systems.

Chapter 8 covers PLDs, complex PLDs (CPLDs), ROMs, and SRAMs, and then expands the synthesis target to include FPGAs. Chapter 9 treats the modeling and synthesis of computational units and algorithms found in computer architectures, digital filters, and other signal processors. Chapter 10 develops and refines algorithms and architectures for the arithmetic units of digital machines. In Chapter 11, we use the Verilog HDL in conjunction with fault simulators and timing analyzers to revisit a selection of previously designed machines and consider performance/timing issues and testability, to complete the treatment of design flow tasks that rely heavily on designer intervention. This chapter models the test access port (TAP) controller defined by IEEE 1149.1 standard (commonly known as the JTAG standard), and presents an example of its use. Another elaborate example covers BIST.

Acknowledgments

The author is grateful for the support of colleagues and students who expanded his vision of Verilog and contributed to this text. It represents the combined experience of my developing and teaching courses at the University of Colorado, taking sabbaticals in industry with Hewlett-Packard and Ford Microelectronics Inc., Prisma Inc., spending a sabbatical at the Technical University of Delft (Netherlands), and developing and presenting short courses in Europe and Asia. Some of the companies now exist only in memory, but I am deeply grateful to many individuals in industry and at the University of Colorado for their helping to shape my path in the realm of VLSI circuit design. The reviewers of the original manuscript for the first edition provided encouragement, critical judgment, and many helpful suggestions. Dr. Jim Tracey and Dr. Rodger Ziemer supported and encouraged my initial efforts to focus my work on the design of VLSI circuits. Deepak Goel (Ford Microelectronics) introduced me to VLSI design on the then state-of-the-art Daisy workstations at Ford Microelectronics. Bill Fuchs (Simucad, Inc.) supported my efforts to acquire an industrial-strength Verilog simulator. Tom Saponas and Dave Ritchey (Hewlett-Packard) placed me at the helm of a project to reverse-engineer a dynamic timing analyzer. Two students, David Uranek and Jerry Barnett, assured success. Dave Still (Prisma Corp.), my manager during a summer job, provided the environment and encouragement for me to tackle a significant problem in modeling of a high-performance, multicore system; Stu Sutherland (Sutherland HDL) helped the author gain a deeper appreciation for the issue of race conditions that can creep into the models of a digital system. These insights led to my adhering to the disciplined style of using nonblocking assignments for modeling edge-sensitive behavior and blocked assignments for modeling level-sensitive behavior, and to my efforts to help students understand the operation and design of synchronous digital machines. Rich discussions with Dr. Hubert Kaeslin, friend and colleague (Swiss Federal Institute of Technology – Zurich), led to my delving more deeply into algorithms and architectures for digital processors. Kirk Sprague and Scott Kukel were helpful in developing a Hamming encoder to work with a UART. Cris Hagan's thesis led to the models presented in Chapter 9 for decimators and other functional units found in digital signal processors. Rex Anderson proofread several chapters and scrubbed down the work of the first edition. Students Terry Hansen and Lisa Horton provided the inspiration for the coffee vending machine example, and developed the assembler that supports the RISC CPU. Thanks also to the many students whose classroom dialogue was helpful in the second edition. Profs. Greg Tumbush (University of Colorado at Colorado Springs) and Chen-Huan Chiang (Temple University) provided critical and helpful suggestions for this edition. Thanks also to the many students whose classroom dialogue was helpful in the second edition. Scott Disanno and Irwin Zucker shepherded this edition through the publication process, and Haseen Khan orchestrated the composition of the book from my manuscript. My deep thanks to all of you.

Dedication

This book is dedicated to the memory of Sr. Laurencia Rihn, RSM, and Fr. Jerry Wilson, CSC. My life has been shaped by their faith, encouragement, and love. To my wife, Jerilynn, and our children, Monica, Lucy, Rebecca, Christine, and Michael, their spouses, Mike McCormick, David Steigerwald, Peter Van Dusen, and Michelle Puhr Ciletti, and our grandchildren, the "cousin dozen": Michael Angus, Katherine, Brigid, David, Jackson, Samantha, Peter, Matthew, Ella, Anthony, Abigail, and Joseph—thank you for the journey and love we've shared.

Contents

1 Introduction to Digital Design Methodology 1
1.1 Design Methodology—An Introduction 2
1.1.1 Design Specification 4
1.1.2 Design Partition 4
1.1.3 Design Entry 4
1.1.4 Simulation and Functional Verification 5
1.1.5 Design Integration and Verification 6
1.1.6 Presynthesis Sign-Off 6
1.1.7 Gate-Level Synthesis and Technology Mapping 6
1.1.8 Postsynthesis Design Validation 7
1.1.9 Postsynthesis Timing Verification 8
1.1.10 Test Generation and Fault Simulation 8
1.1.11 Placement and Routing 8
1.1.12 Physical and Electrical Design Rule Checks 9
1.1.13 Parasitic Extraction 9
1.1.14 Design Sign-Off 9
1.2 IC Technology Options 9
1.3 Overview 11
References 11

2 Review of Combinational Logic Design 13
2.1 Combinational Logic and Boolean Algebra 13
2.1.1 ASIC Library Cells 13
2.1.2 Boolean Algebra 16
2.1.3 DeMorgan's Laws 18

2.2 Theorems for Boolean Algebraic Minimization 18
2.3 Representation of Combinational Logic 21
2.3.1 Sum-of-Products Representation 23
2.3.2 Product-of-Sums Representation 26
2.4 Simplification of Boolean Expressions 27
2.4.1 Simplification with Exclusive-Or 36
2.4.2 Karnaugh Maps (SOP Form) 36
2.4.3 Karnaugh Maps (POS Form) 39
2.4.4 Karnaugh Maps and Don't-Cares 40
2.4.5 Extended Karnaugh Maps 41
2.5 Glitches and Hazards 42
2.5.1 Elimination of Static Hazards (SOP Form) 45
2.5.2 Summary: Elimination of Static Hazards in Two-Level Circuits 48
2.5.3 Static Hazards in Multilevel Circuits 49
2.5.4 Summary: Elimination of Static Hazards in Multilevel Circuits 52
2.5.5 Dynamic Hazards 52
2.6 Building Blocks for Logic Design 55
2.6.1 NAND–NOR Structures 55
2.6.2 Multiplexers 60
2.6.3 Demultiplexers 61
2.6.4 Encoders 62
2.6.5 Priority Encoder 63
2.6.6 Decoder 64
2.6.7 Priority Decoder 66
References 67
Problems 67

3 Fundamentals of Sequential Logic Design 69
3.1 Storage Elements 69
3.1.1 Latches 70
3.1.2 Transparent Latches 71
3.2 Flip-Flops 71
3.2.1 D-Type Flip-Flop 71
3.2.2 Master–Slave Flip-Flop 73
3.2.3 J-K Flip-Flops 75
3.2.4 T Flip-Flop 75
3.3 Busses and Three-State Devices 76
3.4 Design of Sequential Machines 80
3.5 State-Transition Graphs 82
3.6 Design Example: BCD to Excess-3 Code Converter 84
3.7 Serial-Line Code Converter for Data Transmission 90
3.7.1 Design Example: A Mealy-Type FSM for Serial Line-Code Conversion 92
3.7.2 Design Example: A Moore-Type FSM for Serial Line-Code Conversion 93

3.8 State Reduction and Equivalent States 95
References 99
Problems 100

4 Introduction to Logic Design with Verilog 103

4.1 Structural Models of Combinational Logic 104
4.1.1 Verilog Primitives and Design Encapsulation 104
4.1.2 Verilog Structural Models 107
4.1.3 Module Ports 108
4.1.4 Some Language Rules 108
4.1.5 Top-Down Design and Nested Modules 109
4.1.6 Design Hierarchy and Source-Code Organization 111
4.1.7 Vectors in Verilog 113
4.1.8 Structural Connectivity 114
4.2 Logic System, Design Verification, and Test Methodology 118
4.2.1 Four-Value Logic and Signal Resolution in Verilog 119
4.2.2 Test Methodology 120
4.2.3 Signal Generators for Testbenches 123
4.2.4 Event-Driven Simulation 125
4.2.5 Testbench Template 125
4.2.6 Sized Numbers 126
4.3 Propagation Delay 126
4.3.1 Inertial Delay 129
4.3.2 Transport Delay 131
4.4 Truth Table Models of Combinational and Sequential Logic with Verilog 131
References 138
Problems 138

5 Logic Design with Behavioral Models of Combinational and Sequential Logic 141

5.1 Behavioral Modeling 141
5.2 A Brief Look at Data Types for Behavioral Modeling 143
5.3 Boolean Equation-Based Behavioral Models of Combinational Logic 143
5.4 Propagation Delay and Continuous Assignments 146
5.5 Latches and Level-Sensitive Circuits in Verilog 148
5.6 Cyclic Behavioral Models of Flip-Flops and Latches 150
5.7 Cyclic Behavior and Edge Detection 152
5.8 A Comparison of Styles for Behavioral Modeling 154
5.8.1 Continuous Assignment Models 154
5.8.2 Dataflow/RTL Models 156
5.8.3 Algorithm-Based Models 160
5.8.4 Naming Conventions: A Matter of Style 161
5.8.5 Simulation with Behavioral Models 162
5.9 Behavioral Models of Multiplexers, Encoders, and Decoders 162

5.10 Dataflow Models of a Linear-Feedback Shift Register 171
5.11 Modeling Digital Machines with Repetitive Algorithms 173
5.11.1 Intellectual Property Reuse and Parameterized Models 178
5.11.2 Clock Generators 180
5.12 Machines with Multicycle Operations 182
5.13 Design Documentation with Functions and Tasks: Legacy or Lunacy? 183
5.13.1 Tasks 184
5.13.2 Functions 185
5.14 Algorithmic State Machine Charts for Behavioral Modeling 187
5.15 ASMD Charts 191
5.16 Behavioral Models of Counters, Shift Registers, and Register Files 195
5.16.1 Counters 195
5.16.2 Shift Registers 202
5.16.3 Register Files and Arrays of Registers (Memories) 206
5.17 Switch Debounce, Metastability, and Synchronizers for Asynchronous Signals 208
5.18 Design Example: Keypad Scanner and Encoder 214
References 223
Problems 223

6 Synthesis of Combinational and Sequential Logic 235
6.1 Introduction to Synthesis 236
6.1.1 Logic Synthesis 237
6.1.2 RTL Synthesis 245
6.1.3 High-Level Synthesis 246
6.2 Synthesis of Combinational Logic 247
6.2.1 Synthesis of Priority Structures 252
6.2.2 Exploiting Logical Don't-Care Conditions 253
6.2.3 ASIC Cells and Resource Sharing 258
6.3 Synthesis of Sequential Logic with Latches 260
6.3.1 Accidental Synthesis of Latches 262
6.3.2 Intentional Synthesis of Latches 266
6.4 Synthesis of Three-State Devices and Bus Interfaces 269
6.5 Synthesis of Sequential Logic with Flip-Flops 272
6.6 Synthesis of Explicit State Machines 275
6.6.1 Synthesis of a BCD-to-Excess-3 Code Converter 276
6.6.2 Design Example: Synthesis of a Mealy-Type NRZ-to-Manchester Line Code Converter 281
6.6.3 Design Example: Synthesis of a Moore-Type NRZ-to-Manchester Line Code Converter 283
6.6.4 Design Example: Synthesis of a Sequence Recognizer 284
6.7 Registered Logic 292
6.8 State Encoding 300

6.9 Synthesis of Implicit State Machines, Registers, and Counters 302
6.9.1 Implicit State Machines 303
6.9.2 Synthesis of Counters 304
6.9.3 Synthesis of Registers 305
6.10 Resets 309
6.11 Synthesis of Gated Clocks and Clock Enables 313
6.12 Anticipating the Results of Synthesis 314
6.12.1 Synthesis of Data Types 314
6.12.2 Operator Grouping 314
6.12.3 Expression Substitution 315
6.13 Synthesis of Loops 318
6.13.1 Static Loops without Embedded Timing Controls 318
6.13.2 Static Loops with Embedded Timing Controls 321
6.13.3 Nonstatic Loops without Embedded Timing Controls 324
6.13.4 Nonstatic Loops with Embedded Timing Controls 326
6.13.5 State-Machine Replacements for Unsynthesizable Loops 329
6.14 Design Traps to Avoid 334
6.15 Divide and Conquer: Partitioning a Design 335
References 336
Problems 337

7 Design and Synthesis of Datapath Controllers 345
7.1 Partitioned Sequential Machines 345
7.2 Design Example: Binary Counter 347
7.3 Design and Synthesis of a RISC Stored-Program Machine 353
7.3.1 RISC SPM: Processor 355
7.3.2 RISC SPM: ALU 355
7.3.3 RISC SPM: Controller 355
7.3.4 RISC SPM: Instruction Set 356
7.3.5 RISC SPM: Controller Design 358
7.3.6 RISC SPM: Program Execution 372
7.4 Design Example: UART 375
7.4.1 UART Operation 376
7.4.2 UART Transmitter 377
7.4.3 UART Receiver 387
References 399
Problems 400

8 Programmable Logic and Storage Devices 415
8.1 Programmable Logic Devices 417
8.2 Storage Devices 417
8.2.1 Read-Only Memory (ROM) 418
8.2.2 Programmable ROM (PROM) 420

8.2.3 Erasable ROMs 421
8.2.4 ROM-Based Implementation of Combinational Logic 423
8.2.5 Verilog System Tasks for ROMs 423
8.2.6 Comparison of ROMs 426
8.2.7 ROM-Based State Machines 426
8.2.8 Flash Memory 430
8.2.9 Static Random Access Memory (SRAM) 430
8.2.10 Ferroelectric Nonvolatile Memory 452
8.3 Programmable Logic Array (PLA) 454
8.3.1 PLA Minimization 457
8.3.2 PLA Modeling 459
8.4 Programmable Array Logic (PAL) 463
8.5 Programmability of PLDs 464
8.6 Complex PLDs (CPLDs) 465
8.7 Field-Programmable Gate Arrays 466
8.7.1 The Role of FPGAs in the ASIC Market 467
8.7.2 FPGA Technologies 469
8.7.3 XILINX Virtex FPGAs 470
8.8 Embeddable and Programmable IP Cores for a System-on-a-Chip (SoC) 470
8.9 Verilog-Based Design Flows for FPGAs 472
8.10 Synthesis with FPGAs 473
References 476
Related Web Sites 476
Problems and FPGA-Based Design Exercises 476

9 Algorithms and Architectures for Digital Processors 515

9.1 Algorithms, Nested-Loop Programs, and Data Flow Graphs 516
9.2 Design Example: Halftone Pixel Image Converter 519
9.2.1 Baseline Design for a Halftone Pixel Image Converter 522
9.2.2 NLP-Based Architectures for the Halftone Pixel Image Converter 526
9.2.3 Minimum Concurrent Processor Architecture for a Halftone Pixel Image Converter 532
9.2.4 Halftone Pixel Image Converter: Design Tradeoffs 547
9.2.5 Architectures for Dataflow Graphs with Feedback 547
9.3 Digital Filters and Signal Processors 554
9.3.1 Finite-Duration Impulse Response Filter 557
9.3.2 Digital Filter Design Process 558
9.3.3 Infinite-Duration Impulse Response Filter 563
9.4 Building Blocks for Signal Processors 566
9.4.1 Integrators (Accumulators) 566
9.4.2 Differentiators 570
9.4.3 Decimation and Interpolation Filters 570

9.5 Pipelined Architectures 576
9.5.1 Design Example: Pipelined Adder 579
9.5.2 Design Example: Pipelined FIR Filter 583
9.6 Circular Buffers 586
9.7 Asynchronous FIFOs—Synchronization across Clock Domains 589
9.7.1 Simplified Asynchronous FIFO 590
9.7.2 Clock Domain Synchronization for an Asynchronous FIFO 599
References 619
Problems 620

10 Architectures for Arithmetic Processors 627

10.1 Number Representation 627
10.1.1 Signed Magnitude Representation of Negative Integers 628
10.1.2 Ones Complement Representation of Negative Integers 629
10.1.3 Twos Complement Representation of Positive and Negative Integers 630
10.1.4 Representation of Fractions 632
10.2 Functional Units for Addition and Subtraction 632
10.2.1 Ripple-Carry Adder 632
10.2.2 Carry Look-Ahead Adder 633
10.2.3 Overflow and Underflow 638
10.3 Functional Units for Multiplication 638
10.3.1 Combinational (Parallel) Binary Multiplier 639
10.3.2 Sequential Binary Multiplier 642
10.3.3 Sequential Multiplier Design: Hierarchical Decomposition 644
10.3.4 STG-Based Controller Design 646
10.3.5 Efficient STG-Based Sequential Binary Multiplier 652
10.3.6 ASMD-Based Sequential Binary Multiplier 658
10.3.7 Efficient ASMD-Based Sequential Binary Multiplier 664
10.3.8 Summary of ASMD-Based Datapath and Controller Design 669
10.3.9 Reduced-Register Sequential Multiplier 670
10.3.10 Implicit-State-Machine Binary Multiplier 675
10.3.11 Booth's Algorithm Sequential Multiplier 687
10.3.12 Bit-Pair Encoding 702
10.4 Multiplication of Signed Binary Numbers 710
10.4.1 Product of Signed Numbers: Negative Multiplicand, Positive Multiplier 710
10.4.2 Product of Signed Numbers: Positive Multiplicand, Negative Multiplier 710
10.4.3 Product of Signed Numbers: Negative Multiplicand, Negative Multiplier 710
10.5 Multiplication of Fractions 711
10.5.1 Signed Fractions: Positive Multiplicand, Positive Multiplier 714
10.5.2 Signed Fractions: Negative Multiplicand, Positive Multiplier 714
10.5.3 Signed Fractions: Positive Multiplicand, Negative Multiplier 714
10.5.4 Signed Fractions: Negative Multiplicand, Negative Multiplier 715

10.6 Functional Units for Division 715
10.6.1 Division of Unsigned Binary Numbers 716
10.6.2 Efficient Division of Unsigned Binary Numbers 724
10.6.3 Reduced-Register Sequential Divider 734
10.6.4 Division of Signed (2s Complement) Binary Numbers 739
10.6.5 Signed Arithmetic 739
References 742
Problems 742

11 Postsynthesis Design Tasks 749
11.1 Postsynthesis Design Validation 749
11.2 Postsynthesis Timing Verification 753
11.2.1 Static Timing Analysis 755
11.2.2 Timing Specifications 757
11.2.3 Factors That Affect Timing 760
11.3 Elimination of ASIC Timing Violations 766
11.4 False Paths 767
11.5 System Tasks for Timing Verification 769
11.5.1 Timing Check: Setup Condition 770
11.5.2 Timing Check: Hold Condition 770
11.5.3 Timing Check: Setup and Hold Conditions 771
11.5.4 Timing Check: Pulsewidth Constraint 773
11.5.5 Timing Check: Signal Skew Constraint 773
11.5.6 Timing Check: Clock Period 774
11.5.7 Timing Check: Recovery Time 774
11.6 Fault Simulation and Manufacturing Tests 775
11.6.1 Circuit Defects and Faults 776
11.6.2 Fault Detection and Testing 780
11.6.3 *D*-Notation 782
11.6.4 Automatic Test Pattern Generation for Combinational Circuits 786
11.6.5 Fault Coverage and Defect Levels 788
11.6.6 Test Generation for Sequential Circuits 788
11.7 Fault Simulation 792
11.7.1 Fault Collapsing 793
11.7.2 Serial Fault Simulation 793
11.7.3 Parallel Fault Simulation 794
11.7.4 Concurrent Fault Simulation 794
11.7.5 Probabilistic Fault Simulation 794
11.8 JTAG Ports and Design for Testability 794
11.8.1 Boundary Scan and JTAG Ports 795
11.8.2 JTAG Modes of Operation 796

11.8.3 JTAG Registers 798
11.8.4 JTAG Instructions 800
11.8.5 TAP Architecture 801
11.8.6 TAP Controller State Machine 803
11.8.7 Design Example: Testing with JTAG 807
11.8.8 Design Example: Built-In Self-Test 830
References 845
Problems 845

A Verilog Primitives 851

A.1 Multiinput Combinational Logic Gates 851
A.2 Multioutput Combinational Gates 853
A.3 Three-State Logic Gates 854
A.4 MOS Transistor Switches 855
A.5 MOS Pull-Up/Pull-Down Gates 860
A.6 MOS Bidirectional Switches 860

B Verilog Keywords 863

C Verilog Data Types 865

C.1 Nets 865
C.2 Register Variables 866
C.3 Constants 870
C.4 Referencing Arrays of Nets or Regs 871

D Verilog Operators 873

D.1 Arithmetic Operators 873
D.2 Bitwise Operators 875
D.3 Reduction Operators 875
D.4 Logical Operators 876
D.5 Relational Operators 877
D.6 Shift Operators 878
D.7 Conditional Operator 878
D.8 Concatenation Operator 879
D.9 Expressions and Operands 880
D.10 Operator Precedence 880
D.11 Arithmetic with Signed Data Types 881
D.12 Signed Literal Integers 882
D.13 System Functions for Sign Conversion 882
2.1.1 Assignment Width Extension 883

E Verilog Language Formal Syntax 885

F Verilog Language Formal Syntax 887

F.1 Source text 887
F.2 Declarations 890
F.3 Primitive instances 894
F.4 Module and generated instantiation 895
F.5 UDP declaration and instantiation 896
F.6 Behavioral statements 897
F.7 Specify section 901
F.8 Expressions 905
F.9 General 909

G Additional Features of Verilog 913

G.1 Arrays of Primitives 913
G.2 Arrays of Modules 913
G.3 Hierarchical Dereferencing 914
G.4 Parameter Substitution 915
G.5 Procedural Continuous Assignment 916
G.6 Intra-Assignment Delay 917
G.7 Indeterminate Assignment and Race Conditions 918
G.8 wait Statement 921
G.9 fork ... join Statement 922
G.10 Named (Abstract) Events 922
G.11 Constructs Supported by Synthesis Tools 923

H Flip-Flop and Latch Types 925

I Verilog-2001, 2005 927

I.1 ANSI C Style Changes 927
I.2 Code Management 930
I.3 Support for Logic Modeling 933
I.4 Support for Arithmetic 934
I.5 Sensitivity List for Event Control 940
I.6 Sensitivity List for Combinational Logic 940
I.7 Parameters 942
I.8 Instance Generation 944

J Programming Language Interface 949

K Web sites 951

L Web-Based Resources 953

Index 955

CHAPTER 1

Introduction to Digital Design Methodology

Classical design methods relied on schematics and manual methods to design a circuit, but today computer-based languages are widely used to design circuits of enormous size and complexity. There are several reasons for this shift in practice. No team of engineers can correctly design and manage, by manual methods, the details of state-of-the-art integrated circuits (ICs) containing several million gates, but using hardware description languages (HDLs) designers easily manage the complexity of large designs. Even small designs rely on language-based descriptions, because designers have to quickly produce correct designs targeted for an ever-shrinking window of opportunity in the marketplace.

Language-based designs are portable and independent of technology, allowing design teams to modify and re-use designs to keep pace with improvements in technology. As physical dimensions of devices shrink, denser circuits with better performance can be synthesized from an original HDL-based model.

HDLs are a convenient medium for integrating intellectual property (IP) from a variety of sources with a proprietary design. By relying on a common design language, models can be integrated for testing and synthesized separately or together, with a net reduction in time for the design cycle. Some simulators also support mixed descriptions based on multiple languages.

The most significant gain that results from the use of an HDL is that a working circuit can be synthesized automatically from a language-based description, bypassing the laborious steps that characterize manual design methods (e.g., logic minimization with Karnaugh maps).

HDL-based synthesis is now the dominant design paradigm used by industry. Today, designers build a software prototype/model of the design, verify its functionality, and then use a synthesis tool to automatically optimize the circuit and create a netlist in a physical technology.

HDLs and synthesis tools focus an engineer's attention on functionality rather than on individual transistors or gates; they synthesize a circuit that will realize the desired functionality, and satisfy area and/or performance constraints. Moreover, alternative architectures can be generated from a single HDL model and evaluated quickly to perform design tradeoffs. Functional models are also referred to as behavioral models.

HDLs serve as a platform for several tools: design entry, design verification, test generation, fault analysis and simulation, timing analysis and/or verification, synthesis, and automatic generation of schematics. This breadth of use improves the efficiency of the design flow by eliminating translations of design descriptions as the design moves through the tool chain.

Two languages enjoy widespread industry support: Verilog™ [1] and VHDL [2]. Both languages are IEEE (Institute of Electrical and Electronics Engineers) standards; both are supported by synthesis tools for ASICs (application-specific integrated circuits) and FPGAs (field-programmable gate arrays). Languages for analog circuit design, such as Spice [3], play an important role in verifying critical timing paths of a circuit, but these languages impose a prohibitive computational burden on large designs, cannot support abstract styles of design, and become impractical when used on a large scale. Hybrid languages (e.g., Verilog-A) [4] are used in designing mixed-signal circuits, which have both digital and analog circuitry. System-level design languages, such as SystemC [5] and Superlog™ [6], are now emerging to support a higher level of design abstraction than can be supported by Verilog or VHDL.

1.1 Design Methodology—An Introduction

ASICs and FPGAs are designed systematically to maximize the likelihood that a design will be correct and will be fabricated without fatal flaws. Designers follow a "design flow" like that shown in Figure 1-1, which specifies a sequence of major steps that will be taken to design, verify, synthesize, and test a digital circuit. ASIC design flows involve several activities, from specification and design entry, to place-and-route and timing closure of the circuit in silicon. Timing closure is attained when all of the signal paths in the design satisfy the timing constraints imposed by the interface circuitry, the circuit's sequential elements, and the system clock. Although the design flow appears to be linear, in practice it is not. Various steps might be revisited as design errors are discovered, requirements change, or performance and design constraints are violated. For example, if a circuit fails to meet timing constraints, a new placement and routing step will have to be taken, perhaps including redesign of critical paths.

Design flows for standard-cell-based ASICs are more complex than those for FPGAs because the architecture of an ASIC is not fixed. Consequently, the performance that can be realized from a design depends on the physical placement and routing of the cells on the die, as well as the underlying device properties. Interconnect delays play a significant role in determining performance in submicron designs below 0.18 μm, in which prelayout estimates of path delays do not guarantee timing closure of the routed design.

The following sections will clarify the design flow described in Figure 1-1.

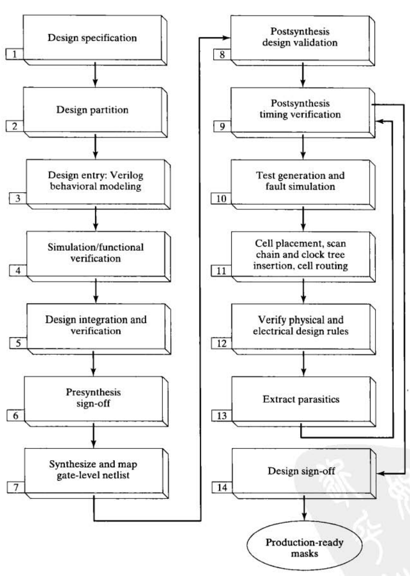


FIGURE 1-1 Design flow for HDL-based ASICs.

1.1.1 Design Specification

The design flow begins with a written specification for the design. The specification document can be a very elaborate statement of functionality, timing, silicon area, power consumption, testability, fault coverage, and other criteria that govern the design. At a minimum, the specification describes the functional characteristics that are to be implemented in a design. Typically, state transition graphs, timing charts, and algorithmic-state machine (ASM) charts are used to describe sequential machines, but interpretation of the specification can be problematic, because the HDL-based model might actually implement an unintended interpretation of the specification. The emerging high-level languages, like SystemC [5], and Superlog [6] hold the promise that the language itself provides an executable specification of the design, which can then be translated and synthesized into a circuit.

1.1.2 Design Partition

In today's methodologies for designing ASICs and FPGAs, large circuits are partitioned to form an *architecture*—a configuration of interacting functional units, such that each is described by a behavioral model of its functionality. The process by which a complex design is progressively partitioned into smaller and simpler functional units is called *top-down design* or *hierarchical design*. HDLs support top-down design with mixed levels of abstraction by providing a common framework for partitioning, synthesizing, and verifying large, complex systems. Parts of large designs can be linked together for verification of overall functionality and performance. The partitioned architecture consists of functional units that are simpler than the whole, and each can be described by an HDL-based model. The aggregate description is often too large to synthesize directly, but each functional unit of the partition can be synthesized in a reasonable amount of time.

1.1.3 Design Entry

Design entry means composing a language-based description of the design and storing it in an electronic format in a computer. Modern designs are described by hardware description languages, like Verilog, because it takes significantly less time to write a Verilog behavioral description and synthesize a gate-level realization of a large circuit than it does to develop the gate-level realization by other means, such as bottom-up manual entry. This saves time that can be put to better use in other parts of the design cycle. The ease of writing, changing, or substituting Verilog descriptions encourages architectural exploration. Moreover, a synthesis tool itself will find alternative realizations of the same functionality and generate reports describing the attributes of the design.

Synthesis tools create an optimal internal representation of a circuit before mapping the description into the target technology. The internal database at this stage is generic, which allows it to be mapped into a variety of technologies. For example, the technology mapping engine of a synthesis tool will use the internal format to migrate a design from an FPGA technology to an ASIC standard cell library, without having to reoptimize the generic description.

HDL-based designs are easier to debug than schematics. A behavioral description encapsulating complex functionality hides underlying gate-level detail, so there is less information to cope with in trying to isolate problems in the functionality of the design. Furthermore, if the behavioral description is functionally correct, it is a gold standard for subsequent gate-level realizations.

HDL-based designs incorporate documentation within the design by using descriptive names, by including comments to clarify intent, and by explicitly specifying architectural relationships, thereby reducing the volume of documentation that must be kept in other archives. Simulation of a language-based model explicitly specifies the functionality of the design. Since the language is a standard, documentation of a design can be decoupled from a particular vendor's tools.

Behavioral modeling is the predominant descriptive style used by industry, enabling the design of massive chips. *Behavioral modeling describes the functionality of a design* by specifying what the designed circuit will do, not how to build it in hardware. It specifies the input–output model of a logic circuit and suppresses details about physical, gate-level implementation.

Behavioral modeling encourages designers to (1) rapidly create a behavioral prototype of a design (without binding it to hardware details), (2) verify its functionality, and then (3) use a synthesis tool to optimize and map the design into a selected physical technology. If the model has been written in a synthesis-ready style, the synthesis tool will remove redundant logic, perform tradeoffs between alternative architectures and/or multilevel equivalent circuits, and ultimately achieve a design that is compatible with area or timing constraints. By focusing the designer's attention on the functionality that is to be implemented rather than on individual logic gates and their interconnections, behavioral modeling provides the freedom to explore alternatives to a design before committing it to production.

Aside from its importance in synthesis, behavioral modeling provides flexibility to a design project by allowing parts of the design to be modeled at different levels of abstraction. The Verilog language accommodates mixed levels of abstraction so that portions of the design that are implemented at the gate level (i.e., structurally) can be integrated and simulated concurrently with other parts of the design that are represented by behavioral descriptions.

1.1.4 Simulation and Functional Verification

The functionality of a design is verified (Step 4 in Figure 1-1) either by simulation or by formal methods [7]. Our discussion will focus on simulation that is reasonable for the size of circuits we can present here. The design flow iterates back to Step 3 until the functionality of the design has been verified. The verification process is threefold; it includes (1) development of a test plan, (2) development of a testbench, and (3) execution of the test.

1.1.4.1 Test Plan Development A carefully documented *test plan* is developed to specify what functional features are to be tested and how they are to be tested. For example, the test plan might specify that the instruction set of an arithmetic and logic unit (ALU) will be verified by an exhaustive simulation of its behavior, for a specific set of

input data. Test plans for sequential machines must be more elaborate to ensure a high level of confidence in the design, because they may have a large number of states. A test plan identifies the stimulus generators, response monitors, and the gold standard response against which the model will be tested.

1.1.4.2 Testbench Development The *testbench* is a Verilog module in which the unit under test (UUT) has been instantiated, together with pattern generators that are to be applied to the inputs of the model during simulation. Graphical displays and/or response monitors are part of the testbench. The testbench is documented to identify the goals and sequential activity that will be observed during simulation (e.g., "Testing the opcodes"). If a design is formed as an architecture of multiple modules, each must be verified separately, beginning with the lowest level of the design hierarchy, then the integrated design must be tested to verify that the modules interact correctly. In this case, the test plan must describe the functional features of each module and the process by which they will be tested, but the plan must also specify how the aggregate is to be tested.

1.1.4.3 Test Execution and Model Verification The testbench is exercised according to the test plan and the response is verified against the original specification for the design, e.g. does the response match that of the prescribed ALU? This step is intended to reveal errors in the design, confirm the syntax of the description, verify style conventions, and eliminate barriers to synthesis. Verification of a model requires a systematic, thorough demonstration of its behavior. *There is no point in proceeding further into the design flow until the model has been verified.*

1.1.5 Design Integration and Verification

After each of the functional subunits of a partitioned design have been verified to have correct functionality, the architecture must be integrated and verified to have the correct functionality. This requires development of a separate testbench whose stimulus generators exercise the input–output functionality of the top-level module, monitor port and bus activity across module boundaries, and observe state activity in any embedded state machines. *This step in the design flow is crucial* and must be executed thoroughly to ensure that the design that is being signed off for synthesis is correct.

1.1.6 Presynthesis Sign-Off

A demonstration of full functionality is to be provided by the testbench, and any discrepancies between the functionality of the Verilog behavioral model and the design specification must be resolved. *Sign-off* occurs after all known functional errors have been eliminated.

1.1.7 Gate-Level Synthesis and Technology Mapping

After all syntax and functional errors have been eliminated from the design and sign-off has occurred, a synthesis tool is used to create an optimal Boolean description and compose it in an available technology. In general, a synthesis tool removes redundant logic and seeks to reduce the area of the logic needed to implement the functionality

and satisfy performance (speed) specifications. This step produces a netlist of standard cells or a database that will configure a target FPGA.

1.1.8 Postsynthesis Design Validation

Design validation compares the response of the synthesized gate-level description to the response of the behavioral model. This can be done by a testbench that instantiates both models, and drives them with a common stimulus, as shown in Figure 1-2. The responses can be monitored by software and/or by visual/graphical means to see whether they have identical functionality. For synchronous designs, the match must hold at the boundaries of the machine's cycle—intermediate activity is of no consequence. If the functionality of the behavioral description and the synthesized realization do not match, painstaking work must be done to understand and resolve the discrepancy. Postsynthesis design validation can reveal software race conditions in the behavioral model that cause events to occur in a different clock cycle than expected.[1] We will discuss how good modeling techniques can prevent this outcome.

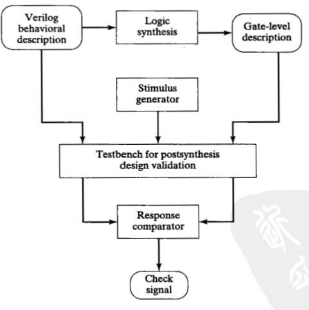


FIGURE 1-2 Postsynthesis design validation.

[1]Postsynthesis validation in an ASIC design flow is followed by a step for postlayout timing verification.

PDG

1.1.9 Postsynthesis Timing Verification

Although the synthesis process is intended to produce a circuit that meets timing specifications, the circuit's timing margins must be checked to verify that speeds are adequate on critical paths (Step 9). This step is repeated after Step 13, because synthesis tools do not accurately anticipate the effect of the capacitive delays induced by interconnect metalization in the layout. Ultimately, these delays must be extracted from the properties of the materials and the geometric details of the fabrication masks. The extracted delays are used by a static timing analyzer to verify that the longest paths do not violate timing constraints. The circuit might have to be resynthesized or re-placed and rerouted to meet specifications. Resynthesis might require (1) transistor resizing, (2) architectural modifications/substitutions, and (3) device substitution (more speed at the cost of more area).

1.1.10 Test Generation and Fault Simulation

After fabrication, integrated circuits must be tested to verify that they are free of defects and operate correctly. Contaminants in the clean-room environment can cause defects in the circuit and render it useless. In this step of the design flow a set of test vectors is applied to the circuit and the response of the circuit is measured. Testing considers process-induced faults, not design errors. Design errors should be detected before presynthesis sign-off. Testing is daunting, for an ASIC chip might have millions of transistors, but only a few hundred package pins that can be used to probe the internal circuits. The designer might have to embed additional, special circuits that will enable a tester to use only a few external pins to test the entire internal circuitry of the ASIC, either alone or on a printed circuit board.

The patterns that are used to verify a behavioral model can be used to test the fabricated part that results from synthesis, but they might not be robust enough to detect a sufficiently high level of manufacturing defects. Combinational logic can be tested for faults exhaustively, but sequential machines present special challenges, as we will see in Chapter 11. Fault simulation questions whether the chips that come off the fabrication line can, in fact, be tested to verify that they operate correctly. Fault simulation is conducted to determine whether a set of test vectors will detect a set of faults. The results of fault simulation guide the use of software tools for generating additional test patterns. To eliminate the possibility that a part could be produced but not tested, test patterns are generated before the device is fabricated, to allow for possible changes in the design, such as a scan path.[2]

1.1.11 Placement and Routing

The placement and routing step of the ASIC design flow arranges the cells on the die and connects their signal paths. In cell-based technology the individual cells are integrated to form a global mask that will be used to pattern the silicon wafer with gates.

[2]Scan paths are formed by replacing ordinary flip-flops with specially designed flip-flops that can be connected together in test mode to form a shift register. Test patterns can be scanned into the design, and applied to the internal circuitry. The response of the circuit can be captured in the scan chain and shifted out for analysis.

This step also might involve inserting a clock tree into the layout, to provide a skew-free distribution of the clock signal to the sequential elements of the design. If a scan path is to be used, it will be inserted in this step too.

1.1.12 Physical and Electrical Design Rule Checks

The physical layout of a design must be checked to verify that constraints on material widths, overlaps, and separations are satisfied. Electrical rules are checked to verify that fanout constraints are met and that signal integrity is not compromised by electrical crosstalk and power-grid drop. Noise levels are also checked to determine whether electrical transients are problematic. Power dissipation is modeled and analyzed in this step to verify that the heat generated by the chip will not damage the circuitry.

1.1.13 Parasitic Extraction

Parasitic capacitance induced by the layout is extracted by a software tool and then used to produce a more accurate verification of the electrical characteristics and timing performance of the design (Step 13). The results of the extraction step are used to update the loading models that are used in timing calculations. Then the timing constraints are checked again to confirm that the design, as laid out, will function at the specified clock speed.

1.1.14 Design Sign-Off

Final sign-off occurs after all of the design constraints have been satisfied and timing closure has been achieved. The mask set is ready for fabrication. The description consists of the geometric data (usually in GDS-II format) that will determine the photomasking steps of the fabrication process. At this point significant resources have been expended to ensure that the fabricated chip will meet the specifications for its functionality and performance.

1.2 IC Technology Options

Figure 1-3 shows various options for creating the physical realization of a digital circuit in silicon, ranging from programmable logic devices (PLDs) to full-custom ICs. Fixed-architecture programmable logic devices serve the low end of the market (i.e., low volume and low performance requirements). They are relatively cheap commodity parts, targeted for low-volume designs.

The physical database of a design might be implemented as (1) a full-custom layout of high-performance circuitry, (2) a configuration of standard cells, or (3) gate arrays (field- or mask-programmable), depending on whether the anticipated market for the ASIC offsets the cost of designing it, and the required profit. Full-custom ICs occupy the high end of the cost–performance domain, where sufficient volume or a customer with corporate objectives and sufficient resources warrant the development time and investment required to produce fully custom designs having minimal area and maximum speed. FPGAs have a fixed, but electrically programmable architecture

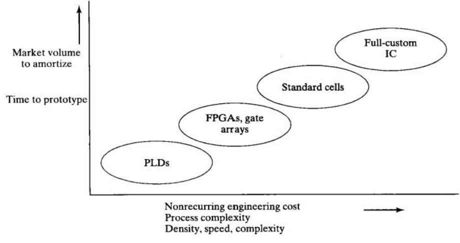


FIGURE 1-3 Alternative technologies for IC implementation.

for implementing modest-sized designs. The tools supporting this technology allow a designer to write and synthesize a Verilog description into a working physical part on a prototype board in a matter of minutes. Consequently, design revisions can be made at very low cost. Board layout can proceed concurrently with the development of the part because the footprint and pin configuration of an FPGA are known. Low-volume prototyping sets the stage for the migration of a design to mask-programmable and standard-cell-based parts.

In mask-programmable gate-array technology a wafer is populated with an array of individual transistors that can be interconnected to create logic gates that implement a desired functionality. The wafers are prefabricated and later personalized with metal interconnect for a customer. All but the metalization masks are common to all wafers, so the time and cost required to complete masks is greatly reduced, and the other nonrecurring engineering (NRE) costs are amortized over the entire customer base of a silicon foundry.

Standard cell technology predesigns and characterizes individual logic gates to the mask level and assembles them in a shared library. A place-and-route tool places the cells in channels on the wafer, interconnects them, and integrates their masks to create the functionality for a specific application. The mask set for a customer is specific to the logic being implemented and can cost over $500K for large circuits, but the NRE costs associated with designing and characterizing the cell library are amortized over the entire customer base. In high-volume applications, the unit cost of the parts can be relatively cheap compared to the unit cost of PLDs and FPGAs.

1.3 Overview

The following chapters will cover most of the steps in the design flow presented in Figure 1-1, but not cell placement and routing, design-rule checking, or parasitic extraction. These steps are conducted by separate tools, which operate on the physical mask database rather than on an HDL model of the design, and they presume that a functionally correct design has been synthesized successfully. The steps we cover are the mainstream designer-driven steps in the overall ASIC flow.

In the remaining chapters, we will review manual methods for designing combinational and sequential logic design in Chapters 2 and 3. Then we will treat combinational logic design (Chapter 4) and sequential logic design (Chapter 5) using Verilog, and by example, contrast manual and HDL-based methods. This chapter also introduces the use of ASM charts and algorithmic state machine and datapath (ASMD) charts, which prove to be very useful in writing behavioral models of sequential machines. Chapter 6 covers synthesis of combinational and sequential logic with Verilog models. This chapter equips the designer with the background to compose synthesis-friendly designs and to avoid common pitfalls that can thwart a design. Chapter 7 continues with a treatment of datapath controllers, including a RISC CPU and a UART. Chapter 8 introduces PLDs, CPLDs, RAMS and ROMS, and FPGAs. The problems at the end of this chapter specify designs that can be implemented on a widely available prototyping board. Chapter 9 covers algorithms and architectures for digital processors, and Chapter 10 treats architectures for arithmetic operations. Chapter 11 treats the postsynthesis issues of timing verification, test generation, and fault simulation, including JTAG and BIST.

Three things matter in learning design with an HDL: examples, examples, and examples. We present several examples, with increasing difficulty, and make available their Verilog descriptions. Several challenging problems are included at the end of each chapter that require design with Verilog. We urge the reader to embrace the mantra: simplify, clarify, verify.

REFERENCES

1. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*, Language Reference Manual (LRM), IEEE Std.1364–1995. Piscataway, NJ: Institute of Electrical and Electronic Engineers, 1996.
2. *IEEE Standard VHDL Language Reference Manual* (LRM), IEEE Std, 1076–1987. Piscataway, NJ: Institute of Electrical and Electronic Engineers, 1988.
3. Negel LW. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*, Memo ERL-M520, Department of Electrical Engineering and Computer Science, University of California at Berkeley, May 9, 1975.
4. Fitzpatrick D, Miller I. *Analog Behavioral Modeling with the Verilog-A Language*, Boston: Kluwer, 1998.
5. SystemC Draft Specification, Mountain View, CA: Synopsys, 1999.

6. Rich, D., Fitzpatrick, T., "Advanced Verification Using the Superlog Language," Proc. Int. HDL Conference, San Jose, March 2002.
7. Chang H, et al. *Surviving the SOC Revolution*, Boston: Kluwer, 1999.

CHAPTER 2

Review of Combinational Logic Design

This chapter will review manual methods for designing combinational logic. In Chapter 6 we will see how these steps can be automated with modern design tools.

2.1 Combinational Logic and Boolean Algebra

Combinational logic forms its outputs as Boolean functions of its input variables on an instantaneous basis. That is, at any time t the outputs y_1, y_2, and y_3 in Figure 2-1 depend on only the values of a, b, c, and d at time t. The outputs of combinational logic at any time t are a function of only the inputs at time t. The outputs of other circuits may depend on the history of the inputs up to time t, and they are called sequential circuits. Sequential circuits require memory elements in hardware.

The variables in a logic circuit are binary—they may have a value of 0 or 1. Hardware implementations of logic circuits use either positive logic, in which a high voltage level, say *5* volts, corresponds to a logical value of 1, and a low voltage, say 0, corresponds to a logical 0. In negative logic, a low electrical level corresponds to a logical 1, and a high electrical level corresponds to a 0.

Some common logic gates are shown in Figure 2-2, together with the Boolean equation that determines the value of the output of the gate as a function of its inputs, and Table 2-1 lists common symbols for hardware-based Boolean logic operations.[1]

2.1.1 ASIC Library Cells

Logic gates are implemented physically by a transistor-level circuit. For example, in CMOS (complementary metal-oxide semiconductor) technology, a logic inverter consists

[1]*Note:* The schematic symbol for the three-state buffer uses the symbol z to indicate the high impedance condition of the device.

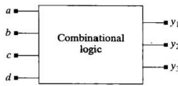


FIGURE 2-1 Block diagram symbol for combinational logic having four inputs and three outputs.

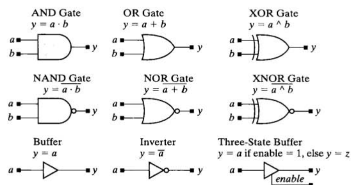


FIGURE 2-2 Schematic symbols and Boolean relationships for some common logic gates.

TABLE 2-1 Common Boolean logic symbols and operations.

Symbol	Logic Operation
+	Logic "or"
·	Logical "and"
⊕	Exclusive "or"
^	Exclusive "or"
'	Logical negation
–	Logical negation (overbar)

of a series connection of *p*-channel and *n*-channel MOS transistors having a common drain that serves as the output, and a common gate that serves as the input. When the input is low, the *p*-channel device conducts and the *n*-channel device is an open circuit. In this mode the output capacitor charges to V_{dd}. When the input is high, the *n*-channel device conducts, and the *p*-channel device is an open circuit. This discharges the output node capacitor to ground. Figure 2-3 shows (b) the pull-up and (c) pull-down paths for current in the inverter in (a).

FIGURE 2-3 CMOS transistor-level schematics: (a) inverter with output load capacitance, (b) inverter with pull-up (charging) signal paths, and (c) inverter with pull-down (discharging) signal paths.

Other logic gates can be implemented using the same basic principles of pull-up and pull-down logic. Figure 2-4 shows the transistor-level schematic for a three-input NAND gate. If one or more of the inputs is low, the output node Y is pulled up to V_{dd}; all inputs must be high to pull the output to ground.

Very large-scale integrated (VLSI) circuits that implement logic gates are fabricated by a series of processing steps in which photomasks are used to selectively dope a silicon wafer to form and connect transistors. Figure 2-5 shows a composite view of the basic masks used in an elementary process to fabricate a CMOS inverter by implanting semiconductor dopants and depositing metal and polycrystalline silicon. The masking steps are performed in a well-defined sequence, beginning with the implantation of a dopant to form the *n*-well, a region that is heavily doped with an n-type material (e.g., arsenic). A p-channel transistor is formed in the n-well by implanting a p-type (e.g., boron) source and drain regions, and an n-channel transistor is formed in the host silicon substrate. Polycrystalline silicon is deposited to form the gates of the transistors, and metal is deposited to form interconnections in and between devices. The actual processes involve many more steps and can have several more layers of metal than these simple structures. Figure 2-6 shows a cross-section of a simple ASIC cell for an inverter, revealing the doped regions.

Circuits that implement basic and moderately complex Boolean functions are characterized for their functional, electrical, and timing properties, and packaged in

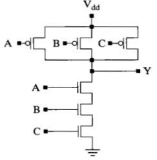


FIGURE 2-4 Transistor-level schematic for three-input CMOS NAND gate.

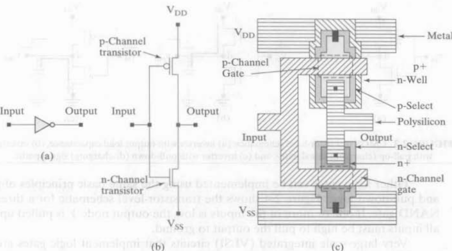


FIGURE 2-5 Views of a CMOS inverter: (a) circuit-symbol view, (b) transistor-schematic view, and (c) simplified composite fabrication mask view.

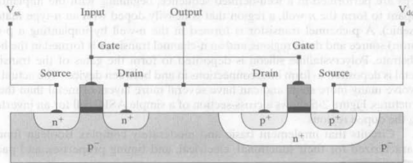


FIGURE 2-6 Simplified side view showing the doping regions of a CMOS inverter.

standard-cell libraries for repeated use in multiple designs. Such libraries commonly contain basic logic gates, flip-flops, latches, muxes, and adders. Synthesis tools build complex integrated circuits by mapping the end result of logic synthesis onto the parts of a cell library to implement the specified functionality with acceptable performance.

2.1.2 Boolean Algebra

The operations of logic circuits are described by Boolean algebra. A Boolean algebra consists of a set of values $\mathbf{B} = \{0, 1\}$ and the operators "+" and "·". The operator "+"

TABLE 2-2 Laws of Boolean algebra.

Laws of Boolean Algebra	SOP Form	POS Form
Combinations with 0, 1	$a + 0 = a$	$a \cdot 1 = a$
	$a + 1 = 1$	$a \cdot 0 = 0$
Commutative	$a + b = b + a$	$ab = ba$
Associative	$(a + b) + c = a + (b + c) = a + b + c$	$(ab)c = a(bc) = abc$
Distributive	$a(b + c) = ab + ac$	$a + bc = (a + b)(a + c)$
Idempote	$a + a = a$	$a \cdot a = a$
Involution	$(a')' = a$	
Complementarity	$a + a' = 1$	$a \cdot a' = 0$

is called the sum operator, the "OR" operator, or the disjunction operator. The operator "$\cdot$" is called the product operator, the "AND" operator, or the conjunction operator. The operators in a Boolean algebra have commutative and distributive properties such that for two Boolean variables A and B having values in $\mathbf{B}$, $a + b = b + a$, and $a \cdot b = b \cdot a$. The operators "+" and "$\cdot$" have identity elements 0 and 1, respectively, such that for any Boolean variable a, $a + 0 = a$, and $a \cdot 1 = a$. Each Boolean variable a has a complement, denoted by a', such that $a + a' = 1$, and $a \cdot a' = 0$. Table 2-2 summarizes the laws of Boolean algebra for sum-of-products (SOP) and product-of-sums (POS) Boolean expressions (more on this later). For simplicity, we have omitted showing the "$\cdot$" operator and will do so freely in the remaining examples.

A multidimensional space spanned by a set of n Boolean variables is denoted by $\mathbf{B}^n$. A point in $\mathbf{B}^n$ is called a *vertex* and is represented by an n-dimensional vector of binary valued elements, for example, (100). A binary variable can be associated with the dimensions of a Boolean space, and a point is identified with the values of the variables. A Boolean variable is represented symbolically by a literal, such as a. A literal is an instance (e.g., a) of a variable or its complement (e.g., a'). Boolean expressions are formed by strings of literals and Boolean operators. A product of literals, such as $ab'c$ is a *cube*. A cube is associated with a set of vertices, and a cube is said to "contain" one or more vertices. Figure 2-7 illustrates how each point in $\mathbf{B}^3$ can be represented (a) by a vector of binary values (e.g., its coordinates), and (b) by a cube of literals.

A *completely specified* m-dimensional Boolean function with n inputs is a mapping from $\mathbf{B}^n$ into $\mathbf{B}^m$, denoted by f: $\mathbf{B}^n \rightarrow \mathbf{B}^m$. An *incompletely specified* function is defined over a subset of $\mathbf{B}^n$, and is considered to have a value of *don't-care* at points outside the domain of definition: f: $\mathbf{B}^n \rightarrow \{0, 1, *\}$, where * denotes don't-care.

The *On_Set* of a Boolean function consists of the vertices at which the function is asserted (logically true), that is, *On-Set* $= \{x : x \in B^n \text{ and } f(x) = 1\}$. The *Off_Set* is the set of vertices at which the function is de-asserted (logically false): $Off_Set = \{x : x \in B^n \text{ and } f(x) = 0\}$. The *Don't-Care-Set* is the set of vertices at which no significance is attached to the value of the function, so *Don't-Care-Set* $= \{x : x \in B^n \text{ and } f(x) = *\}$. The *Don't-Care-Set* set accommodates input patterns that never occur or outputs that will not be observed.

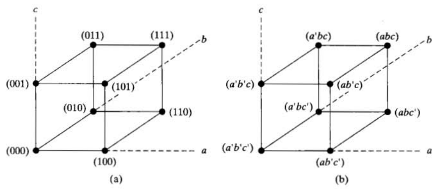


FIGURE 2-7 Points in a Boolean space: (a) represented by vectors of binary variables and (b) represented symbolically.

2.1.3 DeMorgan's Laws

DeMorgan's laws allow us to transform a circuit from an SOP form to a POS form, and vice versa. The first form of the law specifies the complement of a sum of terms:

$$(a + b + c + \dots)' = a' \cdot b' \cdot c' \cdot \dots$$

For two variables, the relationship specifies that

$$(a + b)' = a' \cdot b'$$

The Venn diagrams in Figure 2-8 illustrate the operations of DeMorgan's laws for two variables.

The second form of DeMorgan's laws specifies the complement of the product of terms:

$$(a \cdot b \cdot c \dots)' = a' + b' + c' + \dots$$

For two variables, the law states that

$$(a \cdot b)' = a' + b'$$

These relationships are illustrated by the Venn diagrams in Figure 2-9.

2.2 Theorems for Boolean Algebraic Minimization

Important theorems that are used to minimize Boolean algebraic expressions to produce efficient circuit realizations are shown in Figure 2-10, in POS and SOP form. Logical adjacency and the consensus theorem are illustrated by the Venn diagrams in Figure 2-11. The consensus term, bc, is redundant because it is covered by the union of ab and $a'c$. Laws that apply specifically to the exclusive-or operation are shown in Figure 2-12.

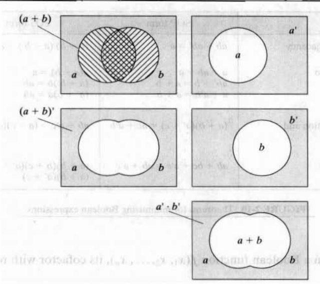


FIGURE 2-8 Venn diagrams illustrating DeMorgan's law: $(a + b)' = a' \cdot b'$.

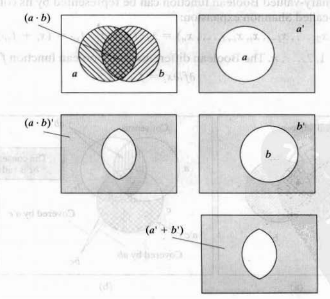


FIGURE 2-9 Venn diagrams illustrating DeMorgan's law: $(a \cdot b)' = a' + b'$.

Theorem	SOP form	POS form
Logical adjacency	$ab + ab' = a$	$(a + b)(a + b') = a$
Absorption or	$a + ab = a$ $ab' + b = a + b$ $a + a'b = a + b$	$a(a + b) = a$ $(a + b')b = ab$ $(a' + b)a = ab$
Multiplication and factoring	$(a + b)(a' + c) = ac + a'b$	$ab + a'c = (a + c)(a' + b)$
Consensus	$ab + bc + a'c = ab + a'c$	$(a + b)(b + c)(a' + c) =$ $(a + b)(a' + c)$

FIGURE 2-10 Theorems for minimizing Boolean expressions.

Given a Boolean function $f(x_1, x_2, \ldots, x_n)$, its cofactor with respect to variable x_i is

$$f_{xi} = f(x_1, x_2, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n),$$

and its cofactor with respect to $x_{i'}$ is

$$f_{xi'} = f(x_1, x_2, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n).$$

A binary-valued Boolean function can be represented by its cofactors as the following so-called Shannon expansion:

$$f(x_1, x_2, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n) = x_i \cdot f_{xi} + x_{i'} \cdot f_{xi'} = (x_i + f_{xi'}) \cdot (x_{i'} + f_{xi})$$

for all $i = 1, 2, \ldots, n$. The Boolean difference of a Boolean function f is given by

$$\partial f / \partial x_i = f_{xi} \oplus f_{xi'}$$

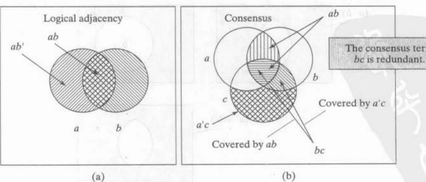


FIGURE 2-11 Venn diagrams: (a) logical adjacency and (b) consensus.

Exclusive-OR laws	
Combinations with 0, 1	$a \oplus 0 = a$
	$a \oplus 1 = a'$
	$a \oplus a = 0$
	$a \oplus a' = 1$
Commutative	$a \oplus b = b \oplus a$
Associative	$(a \oplus b) \oplus c = a \oplus (b \oplus c) = a \oplus b \oplus c$
Distributive	$a(b \oplus c) = ab \oplus ac$
Complement	$(a \oplus b)' = a \oplus b' = a' \oplus b = ab + a'b'$

FIGURE 2-12 Boolean relationships for the exclusive-or operation.

The Boolean difference of f with respect to x_i determines whether f is sensitive to a change in input variable x_i. This property has utility in algebraic methods to determine whether a test detects a fault in a circuit [1]. Also, a binary tree mapping of a Boolean function can be generated by recursively applying Shannon's expansion [2].

2.3 Representation of Combinational Logic

We will consider three common representations of combinational logic: (a) structural (i.e., gate-level) schematics, (b) truth tables, and (c) Boolean equations. An additional representation, a binary decision diagram (BDD) is a graphical representation of a Boolean function and contains the information needed to implement it [2, 3]. BDDs are used primarily within EDA software tools because they can be more efficient and easier to manipulate than truth tables. They can also be helpful in finding hazard covers [4]. We will not make use of BDDs here, but will rely on truth tables instead.

Example 2.1

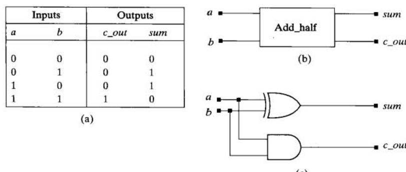
The truth table for the combinational logic of an arithmetic half adder is shown in Figure 2-13. The adder forms a sum and carry out bit from two data bits (without a carry in bit): $(c_out, sum) = a + b$, where "+" denotes arithmetic addition of the data words.

The Boolean equations (i.e., "+" denotes logical OR) describing the half adder can be derived from the truth table and written in SOP form:

$$sum = a'b + ab' = a \oplus b$$

$$c_out = a \cdot b$$

End of Example 2.1



Inputs		Outputs	
a	*b*	*c_out*	*sum*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

FIGURE 2-13 Half adder: (a) truth table, (b) block diagram symbol, and (c) schematic.

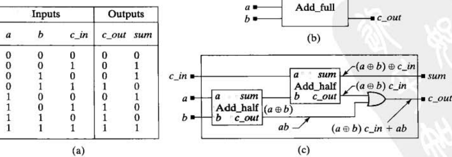
Example 2.2

A full adder forms a sum and carry-out bit from two data bits and a carry-in bit. The truth table for the combinational logic of a full adder is shown in Figure 2-14.

The Boolean equations describing *sum* and *c_out* bits are given by

$$sum = a' \cdot b' \cdot c_in + a' \cdot b \cdot c_in' + a \cdot b' \cdot c_in' + a \cdot b \cdot c_in$$

$$c_out = a' \cdot b \cdot c_in + a \cdot b' \cdot c_in + a \cdot b \cdot c_in' + a \cdot b \cdot c_in$$



Inputs			Outputs	
a	*b*	*c_in*	*c_out*	*sum*
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

FIGURE 2-14 Full adder: (a) truth table, (b) block diagram symbol, and (c) schematic for a fulladder composed of half adders and glue logic.

These can be rearranged as

$$sum = a \oplus b \oplus c_in$$

$$c_out = (a \oplus b) \cdot c_in + a \cdot b$$

The Venn diagram in Figure 2-15 shows the assertions of *sum* and *c_out* as a function of *a, b,* and *c_in*.

Truth tables become unwieldy for functions of several variables because the number of rows grows exponentially with the number of variables. Notice that the truth table of the full adder has twice as many rows as the table for the half adder.

End of Example 2.2

2.3.1 Sum-of-Products Representation

A cube is formed as the product of literals in which a literal appears in either uncomplemented or complemented form. For example, *ab'cd* is a cube but *ab'cbd* is not. A cube need not contain every literal. A *Boolean expression* is a set of cubes, and is typically expressed in an SOP form as the "OR" of product terms (cubes), rather than in set notation.

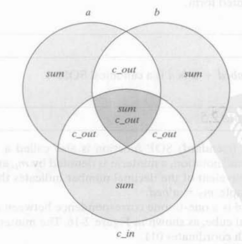


FIGURE 2-15 Venn diagram representation of the truth table for a full adder.

Example 2.3

The following expression is in SOP form: $abc' + bd$.

End of Example 2.3

Each term of a Boolean expression in SOP form is called an *implicant* of the function. A *minterm* is a cube in which every variable appears. The variable will be in either true (uncomplemented) or complemented form (but not both). Thus, a minterm corresponds to a single point (vertex) in $\mathbf{B}^n$. A cube that is not a minterm represents two or more points in $\mathbf{B}^n$. The minterms of a Boolean function correspond to the rows of the truth table at which the function has a value of 1.

Example 2.4

The cube $ab'cd$ is a minterm in $\mathbf{B}^4$. The cube abc is not a minterm. It represents the pair of vertices defined by $abcd + abcd'$.

End of Example 2.4

A Boolean expression in SOP form is said to be canonical if every cube in the expression has a unique representation in which all of the literals are in complemented or uncomplemented form.

Example 2.5

The expression $abcd + a'bcd$ is a canonical SOP.

End of Example 2.5

A *canonic* (standard) SOP function is also called a standard sum-of-products (SSOP). In decimal notation, a minterm is denoted by m_i, and the pattern of 1s and 0s in the binary equivalent of the decimal number indicates the true and complemented literals. For example, $m_7 = a'bcd$.

In $\mathbf{B}^n$, there is a one-to-one correspondence between a minterm and a vertex of an n-dimensional cube, as shown in Figure 2-16. The minterm $m_3 = a'bc$ corresponds to the vertex with coordinates 011.

A *Boolean function* is a set of minterms (vertices) at which the function is asserted. A Boolean function in SOP form is expressed as a sum of minterms.

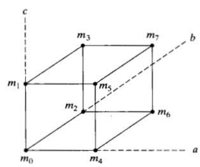


FIGURE 2-16 Correspondence between minterms and vertices in $\mathbf{B}^3$.

Example 2.6

The sum and carry bits of the full adder can be expressed as a sum of minterms, ordered as $\{a, b, c_in\}$ in $\mathbf{B}^3$:

$$sum = m_1 + m_2 + m_4 + m_7 = \Sigma\, m(1, 2, 4, 7)$$
$$c_out = m_3 + m_5 + m_6 + m_7 = \Sigma\, m(3, 5, 6, 7)$$

End of Example 2.6

Example 2.7

The set of shaded vertices in Figure 2-17 define $f = m_1 + m_2 = m_3 = a'b'c + a'bc' + a'bc$.

End of Example 2.7

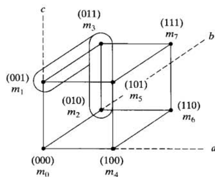


FIGURE 2-17 Set of minterms for $f = a'b'c + a'bc' + a'bc$.

2.3.2 Product-of-Sums Representation

A Boolean function can also be expressed in a POS form in which the expression is written as a product of Boolean factors, each of which is a sum of literals.

Example 2.8

The POS representation of the *c_out* bit in a full adder circuit is formed by expressing the 0s of the truth table in SOP form (see Figure 2-14):

$$c_out' = a'b'c_in' + a'b'c_in + a'bc_in' + ab'c_in'$$

Then the expression for *c_out'* is complemented, giving

$$c_out = (a'b'c_in' + a'b'c_in + a'bc_in' + ab'c_in')'$$

DeMorgan's laws can be applied to *c_out* to give the POS expression shown below:

$$c_out = (a'b'c_in')' \cdot (a'b'c_in)' \cdot (a'bc_in')' \cdot (ab'c_in')'$$
$$c_out = (a + b + c_in) \cdot (a + b + c_in') \cdot (a + b' + c_in) \cdot (a' + b + c_in)$$

End of Example 2.8

A Boolean expression in POS form is said to be *canonical* (i.e., a unique representation for a given function) if each factor has all of the literals in complemented or uncomplemented form, but not both.

A *maxterm* is an OR-ed sum of literals in which each variable appears exactly once in true or complemented form (e.g., $a + b + c_in$ is a maxterm in the POS

expression for c_out). A canonical POS expansion consists of a product of the maxterms of the truth table of a function. The decimal notation of a maxterm is based on the rows of the truth table at which the function is zero (i.e., where f' is asserted). The variables are complemented when forming the POS expression.

Example 2.9

The SOP form of the c_out' bit of a full adder was given in the previous example. The decimal notation for c_out is given by the product of the maxterms that corresponds to the cubes of c_out' as

$$c_out' = a'b'c_in' + a'b'c_in + a'bc_in' + ab'c_in'$$
$$c_out = M_0 \cdot M_1 \cdot M_2 \cdot M_4 = \Pi\, M(0, 1, 2, 4)$$
$$c_out = (a + b + c_in) \cdot (a + b + c_in') \cdot (a + b' + c_in) \cdot (a' + b + c_in)$$

A canonical SOP expression can be a very efficient representation of a Boolean function because there might be very few terms at which the function is asserted. Alternatively, f' expressed as a POS expression might be very efficient because there are only a few terms at which the function is de-asserted.

End of Example 2.9

2.4 Simplification of Boolean Expressions

An SOP expression can be implemented in hardware as a two-level AND-OR logic circuit. Although a Boolean expression can always be expressed in a canonical form, with every cube containing every literal (in complemented or uncomplemented form), such descriptions are usually inefficient and waste hardware. In practice, minimization is important because the cost of hardware implementing a Boolean expression is related to the number of terms in the expression and to the number of literals in a term, that is, in a cube in an SOP expression.

A Boolean expression in SOP form is said to be *minimal* if it contains a minimal number of product terms and literals (i.e., a given term cannot be replaced by another that has fewer literals). A minimum SOP form corresponds to a two-level logic circuit having the fewest gates and the fewest number of gate inputs.

There are four common approaches to simplifying a Boolean expression. The first is a manual graphical method that is guided by Karnaugh maps displaying logical adjacencies of the function. Manual methods are feasible only for functions that have no more than six inputs [5]. The Quine–McCluskey minimization algorithm relies on

logical adjacency and the same principles as logic minimization with Karnaugh maps [6–9]. It can be applied manually on small functions, but can also be implemented as a computer program that is effective for larger circuits. A third method, Boolean minimization [5], is also manual, and relies on clever application of the theorems describing relationships between Boolean variables to find simpler, equivalent expressions. The method is not straightforward, can be difficult, and requires experience. As a fourth alternative, the theorems that are used in Boolean minimization are now embedded in modern synthesis tools and programs such as Espresso-II [10] and mis-II (multilevel interactive synthesis) [4], in which they are used to perform logic minimization and form efficient realizations of two-level and multilevel logic circuits.

Logic minimization searches for efficient representations of Boolean functions. In a Boolean expression, a cube that is contained in another cube is said to be redundant—a cube is *redundant* if its set of vertices is properly contained in the set of vertices of another cube of the function. A Boolean expression is *nonredundant (irredundant)* if no cube contains another cube.

Example 2.10

The following Boolean expression is redundant because the vertex set of ab is a subset of the vertex set of a:

$$f(a, b) = a + ab$$

The redundant cube can be removed to give an equivalent, but more efficient representation:

$$f(a, b) = a$$

End of Example 2.10

Example 2.11

The Boolean function given by $f(c, d) = c'd' + cd$ is irredundant.

End of Example 2.11

The cubes of an irredundant expression do not share a common vertex, that is, their corresponding sets of vertices are pairwise disjoint. Boolean minimization is difficult because the minimum SOP form and minimum POS forms of a Boolean expression are not unique. Boolean minimization/simplification exploits logical adjacency by (1) repeatedly combining cubes that differ in only one literal and (2) eliminating redundant implicants.

Example 2.12

With the objective of illustrating Boolean minimization of the *function* $f(a, b, c) = abc + a'bc + abc' + a'b'c + ab'c' + a'b'c'$, we start by combining cubes that differ by only one literal, as shown in Figure 2-18, which shows the Boolean expression for f and illustrates the adjacent vertices graphically. A pair of adjacent shaded vertices can be combined into a single cube that covers both of them.

An equivalent, minimal expression is formed by removing the cubes $ab + a'b'$ and adding the cube ac' to the expression, as shown in Figure 2-19:

$$f(a, b, c) = ac' + a'c + bc + b'c'$$

Note that $f(a, b, c) = ac' + a'c + a'b'c' + abc$ is also an equivalent expression, but not minimal. Figure 2-20 shows how another equivalent and minimal expression can be obtained by applying logical adjacency to different terms of the original SOP expression.

$$f(a, b, c) = bc + ab + a'b' + b'c'$$

Each term (cube) of a Boolean expression in SOP form is called an implicant of the function. An implicant covers a vertex if the vertex is included in the set of vertices at which the implicant is asserted. An implicant may cover more than one vertex of the function. The fewer the number of literals in a cube, the larger the set of vertices covered by the cube. So the hardware implementation is minimized if a cube has as

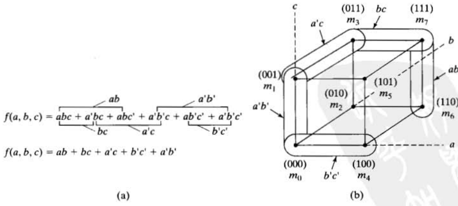


FIGURE 2-18 A Boolean function: (a) representation of adjacent cubes and (b) a graphical representation of adjacent vertices.

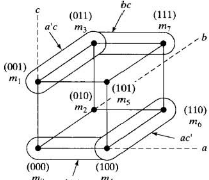


FIGURE 2-19 A second, equivalent minimal expression for the function in Figure 2-18.

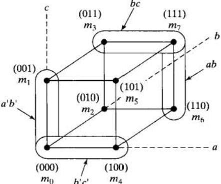


FIGURE 2-20 A third, equivalent minimal expression for the function in Figure 2-18.

few literals as possible. The function $f = abc + abc'$ shown in Figure 2-21 has two vertices that can be combined by logical adjacency into one implicant, *ab*. A representation in terms of just one implicant is preferred to another that uses two implicants because the hardware of the former realization will be simpler and cheaper.

End of Example 2.12

The vertices of the On-Set of a Boolean function provide a complete, but inefficient, description of the function because each minterm includes every literal that is used by the function. We can exploit logical adjacency to reduce the size of the terms

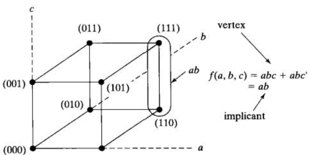


FIGURE 2-21 Combination of adjacent vertices into one implicant.

(cubes) that are used in the SOP form of the function. Terms that are logically adjacent can always be combined into a single term that has fewer literals. But be aware that merely applying logical adjacency to the cubes of a function does not necessarily minimize the function. It is essential that every vertex be covered by a cube, that is, each must belong to the vertex set of a cube; but how many such cubes are necessary to completely cover the function, and which cover is the most efficient? We will now formalize these concepts.

A *prime implicant* of the On-Set of a Boolean function is an implicant whose assertion does not imply assertion of any other implicant of the function. A prime implicant is a cube whose vertices are not properly contained in the set of vertices of some other cube of the function.

Example 2.13

Consider the function $f(a, b, c, d) = a'b'cd + a'bcd + ab'cd + abcd + a'b'c'd'$ in SOP form:

$$\begin{aligned} f(a, b, c, d) &= a'b'cd + a'bcd + ab'cd + abcd + a'b'c'd' \\ &= a'cd + acd + a'b'c'd' \\ &= cd + a'b'c'd' \end{aligned}$$

Note that $a'cd$ and acd both imply cd, so they are not prime implicants. The term $a'b'c'd'$ is a prime implicant.

End of Example 2.13

A prime implicant cannot be combined with another implicant to eliminate a literal or to be eliminated from the expression by absorption. An implicant that implies another implicant is said to be covered by it; the set of the vertices of the covered implicant is a subset of the vertices of the implicant that covers it. The covering implicant, having fewer literals, has more vertices. The set of prime implicants of a Boolean expression is unique.

Example 2.14

The expression, vertices, and prime implicants of a Boolean function in $\mathbf{B}^3$ are shown in Figure 2-22.

End of Example 2.14

A prime implicant that is not covered by any *set* of other implicants is an *essential prime implicant.* An essential prime implicant must be retained in a cover of the function.

Example 2.15

The vertices and implicants of $f(a, b, c) = a'bc + abc + ab'c' + abc'$ are shown in Figure 2-23. The set of prime implicants of f is $\{ac', ab, bc\}$. The set of essential prime implicants, an SOP expression for f, and a minimal SOP expression are also listed below.

Essential prime implicants:	$\{ac', bc\}$
SOP expression	$f(a, b, c) = ac' + ab + bc$
Minimal SOP expression:	$f(a, b, c) = ac' + bc$

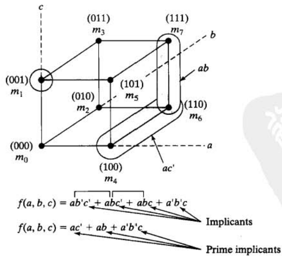


FIGURE 2-22 The vertices and prime implicants of $f = ab'c' + abc' + abc + a'b'c$.

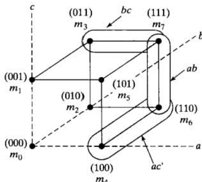


FIGURE 2-23 Vertices and implicants of $f(a, b, c) = a'bc + abc + ab'c' + abc'$.

End of Example 2.15

The process for minimizing a Boolean expression follows these steps: (1) find the set of all prime implicants and (2) find a minimal subset of implicants that covers all of the prime implicants (including the essential prime implicants). The *minimal cover* of a Boolean expression is a subset of prime implicants that covers all of its prime implicants.

Example 2.16

Consider the function given by

$$f(a, b, c, d) = a'b'cd + a'bcd + ab'cd + abcd + a'b'c'd'$$

By combining adjacent terms, we get

$$f(a, b, c, d) = a'cd + acd + a'b'c'd'$$

$$f(a, b, c, d) = cd + a'b'c'd'$$

The set of prime implicants is $\{cd, a'b'c'd'\}$, and the minimal cover is $\{cd, a'b'c'd'\}$.

End of Example 2.16

Boolean minimization combines terms that are logically adjacent, that is, that differ in only one literal.

Example 2.17

Figure 2-24 illustrates how logically adjacent terms in the expression for the carry-out bit of the full adder can be combined to obtain a minimal expression.

End of Example 2.17

Boolean minimization also exploits logical adjacency of complementary expressions.

Example 2.18

Consider the expression $(c + db)(a + e') + c'(d' + b')(a + e')$ and note that $(c + db)' = c'(d' + b')$.

Then

$$(c + db)(a + e') + c'(d' + b')(a + e') = (c + db)(a + e') + (c + db)'(a + e')$$
$$= a + e'.$$

End of Example 2.18

The absorption ($a + ab = a$) and consensus ($ab + bc + a'c = ab + ac'$) properties of Boolean algebra can be used to eliminate redundant terms in an expression.

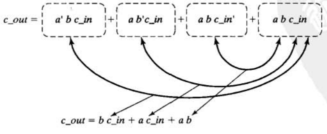


FIGURE 2-24 Boolean minimization of *c_out* in a full adder by combinations of logically adjacent terms.

Example 2.19

The consensus property will be used to reduce the expression: $f = e'fg' + fgh + e'fh$. We rearrange the terms in f to associate with the terms in the consensus law, leading to the result $f = e'fg' + fgh$, as shown in Figure 2-25.

End of Example 2.19

The absorption property can be used repeatedly to eliminate literals in the expression.

Example 2.20

Consider $f = efgh' + e'f'g'h' + e'f$ and rearrange it to get

$$\begin{aligned} f &= efgh' + e'f'g'h' + e'f = efgh' + e'(f + f'g'h') \\ &= efgh' + e'(f + g'h') \\ &= f(egh' + e') + e'g'h' \\ &= f(gh' + e') + e'g'h' \\ &= fgh' + e'f + e'g'h' \end{aligned}$$

End of Example 2.20

Sometimes it is helpful to introduce redundant terms into an expression to support absorption and logical adjacency. A Boolean expression in SOP form is preserved under the following operations: (1) adding the product of a literal and its complement (e.g., the term: aa'), (2) adding the consensus term (e.g., adding bc to $ab + a'c$), and (3) adding to a literal its product with any other literal (e.g., adding ab to a to form $ab + a$).

$$ab + bc + a'c = ab + a'c \quad\longrightarrow\quad gfh + fhe'f + g'e'f = gfh + g'e'f = e'fg' + fgh$$

Consensus term: $fe'h$

FIGURE 2-25 The consensus property and simplification of a Boolean expression.

A Boolean expression in POS form is preserved under the operations: (1) multiplying the expression by a factor consisting of the sum of any literal and its complement [e.g., multiplying by the factor $(a + a')$], (2) introducing the product consensus factor [e.g., introduce the factor $(b + c)$ in $(a + b)(a' + c)$], multiplying a literal by its sum with any other literal [e.g., $a(a + b)$]. Expanding with the consensus term is helpful when it can absorb other terms or eliminate a literal.

Example 2.21

The consensus theorem $(ab + bc + a'c = ab + a'c)$ is usually used to eliminate a redundant term (bc) that is covered by two other terms in an expression. But it can be used to add a redundant term, thereby leading to simplification of a larger expression. As an example of how to add a redundant term, consider the expression $f = bcd + bce + ab + a'c$ and note that the terms $ab + a'c$ are the result of eliminating the consensus term from $ab + bc + a'c$. So, adding the consensus term (bc) back into f gives the expression $f = b'c + bcd + bce + ab + bc + a'c$. Now it is possible to absorb terms with bc and then delete bc, leaving $f = ab + c$.

End of Example 2.21

2.4.1 Simplification with Exclusive-Or

The properties listed in Figure 2-12 for the exclusive-or can be used to simplify expressions.

Example 2.22

The expression for the *sum* output of a full adder is $sum = a' \cdot b' \cdot c_in + a' \cdot b \cdot c_in' + a \cdot b' \cdot c_in' + a \cdot b \cdot c_in$. This expression simplifies to $sum = (a \oplus b) \oplus c_in = a \oplus b \oplus c_in$, which requires a pair of two-input exclusive-or gates in hardware.

End of Example 2.22

2.4.2 Karnaugh Maps (SOP Form)

Karnaugh maps (K-maps) provide a graphical/visual representation of a Boolean function of up to five or six variables. The map of an expression reveals logical adjacencies and opportunities for eliminating a literal from two or more cubes. The columns and rows of the map are arranged so that they are logically adjacent over the space of the

input variables of the function. Each vertex (point) in the Boolean domain of the function is represented by a square in the map. Each cell of the map has an entry to indicate where the vertex is in the *On-Set* (1), *Off-Set* (0), or the *Don't-Care-Set* (x). K-maps facilitate finding the largest possible cubes that cover all 1s without redundancy. Their application requires manual effort.

The K-map of a function of four variables shows all 16 possible vertices. Also, observe the ordering of the rows and columns, and that the topmost and bottommost rows are logically adjacent, and the leftmost and rightmost columns are logically adjacent. Logically adjacent cells that contain a 1 can be combined. A rectangular cluster of cells that are logically adjacent can be combined. Any don't-cares of the function can be used to form prime implicants and create additional possibilities for reduction.

Example 2.23

The K-map in Figure 2-26 shows how its corresponding Boolean function can be reduced by logical adjacency to give $f = bd + b'd'$. The SOP expression corresponding to the minterms at the four corners of the map can be simplified successively as shown below:

$$a'b'c'd' + a'b'cd' + ab'c'd' + ab'cd' = a'b'd' + ab'd'$$

The resulting expression can be reduced to

$$a'b'd' + ab'd' = b'd'$$

Alternatively, the four terms at the corners can be combined to give:

$$a'b'c'd' + ab'c'd' + a'b'cd' + ab'cd' = b'c'd' + b'cd' = b'd'$$

The shaded inner quad of minterms in Figure 2-26 can also be reduced:

$$a'bc'd + a'bcd + abc'd + abcd = bc'd + bcd = bd$$

and so

$$f = b'd' + bd = (b \oplus d)'$$

Note that each corner minterm implies $b'd'$, and that $b'd'$ does not imply another implicant. Therefore, it is a prime implicant. It is also an essential prime implicant. Similarly, bd is an essential prime implicant.

End of Example 2.23

To form a minimal realization from a Karnaugh map, (1) identify all of the essential prime implicants using don't-cares as needed and (2) use the prime implicants to form a cover of the remaining 1s in the map (ignoring don't-cares). In general, the covering set of prime implicants is not unique.

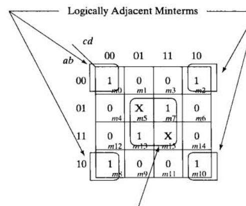


FIGURE 2-26 K-map of $f = a'b'c'd' + a'b'cd' + ab'cd' + ab'c'd' + abc'd + a'bcd$.

Example 2.24

Minimal covers will be found for the Boolean function whose K-map is shown in Figure 2-27.

The prime implicants of f are listed below and those that are essential are identified.

Prime Implicants: $m3, m2, m7, m6 \rightarrow a'c$ (essential)

$m2, m10 \rightarrow b'cd'$ (essential)

$m7, m15 \rightarrow bcd$

$m13, m15 \rightarrow abd$

$m12, m13 \rightarrow abc'$ (essential)

The minimal covers of f are formed by including the essential prime implicants with implicants that cover the remaining vertices.

Minimal Covers: (1) $a'c, b'cd', bcd, abc'$

(2) $a'c, b'cd', abd, abc'$

End of Example 2.24

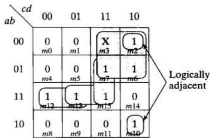


FIGURE 2-27 K-map for $f = abc'd' + abc'd + abcd + a'bcd + a'b'cd + a'b'cd' + a'bcd' + ab'cd'$.

The following steps will form a minimal cover: (1) select an uncovered minterm, (2) identify all adjacent cells containing a 1 or an X, and (3) a single term (not necessarily a minterm) that covers the minterm and all of its neighboring cells having a 1 or an X is an essential prime implicant. Add the term to the set of essential prime implicants. Step 1 is repeated until all of the essential prime implicants have been selected. After Step 1 is complete, find a minimal set of prime implicants that cover the other 1s in the map (do not cover cells containing X). These steps may produce more than one possible minimal cover. Select the cover that has the fewest literals.

2.4.3 Karnaugh Maps (POS Form)

The minimal product of sums form of a Boolean expression is formed by finding a minimal cover of the 0s in the Karnaugh map, then applying DeMorgan's theorem to the result.

Example 2.25

The 0-cells of the K-map shown in Figure 2-28 can be combined through logical adjacency):

$$m_0, m_1, m_4, m_5\text{: } a'b'c'd' + a'b'c'd + a'bc'd' + a'bc'd \rightarrow a'c'$$

$$m_0, m_1, m_8, m_9\text{: } a'b'c'd' + a'b'c'd + ab'c'd' + ab'c'd \rightarrow b'c'$$

$$m_9, m_{11} \quad : ab'c'd + ab'cd \rightarrow ab'd$$

$$m_{14} \quad : abcd'$$

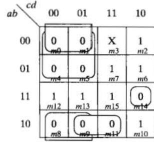


FIGURE 2-28 K-map for a minimal POS expression.

The expression for the 0-cells becomes:

$$f'(a, b, c, d) = a'c' + b'c' + ab'd + abcd'$$

After DeMorgan's law is applied to f', the minimal POS expression for f becomes:

$$f(a, b, c, d) = (a + c)(b + c)(a' + b + d')(a' + b' + c' + d)$$

End of Example 2.25

2.4.4 Karnaugh Maps and Don't-Cares

Don't-cares represent situations in which an input cannot occur or the output does not matter. The general rule is that don't-cares can be used when covering them leads to an improved representation.

Example 2.26

A binary-coded decimal (BCD) word is a 4-bit word whose values correspond to the digits 0, ..., 9. The BCD code, also known as a 8421 code, uses only the first 10 patterns, beginning with 0000_2 and ending with 1001_2. The code for each decimal digit N isobtained by adding 1 to the code of the preceding digit, $N - 1$. Suppose a function f is asserted when the BCD representation of a 4-variable input is 0, 3, 6 or 9 [5]. The K-map in Figure 2-29(a) does not make use of don't-cares (denoted by X).

The function obtained without exploiting don't-cares has 16 literals:

$$f(a, b, c, d) = a'b'c'd' + a'b'cd + a'bcd' + ab'c'd$$

If the don't-cares are included, f has the SOP form $f(a, b, c, d) = a'b'c'd' + a'b'cd + abc'd' + abc'd + abcd + abcd' + ab'c'd + ab'cd$, which has 32 literals. The K-map in Figure 2-29(b) shows how f can be reduced to obtain an SOP form that has 12 literals:

$$f(a, b, c, d) = a'b'c'd' + b'cd + bcd' + ad$$

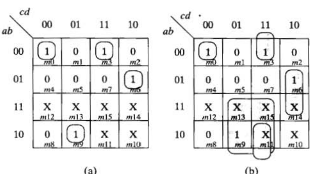


FIGURE 2-29 K-maps (a) without and (b) with don't-cares.

which can be reduced to

$$f(a, b, c, d) = a'b'c'd' + b'cd + ab + ac'd$$

End of Example 2.26

2.4.5 Extended Karnaugh Maps

A 4-variable Karnaugh map can be extended by entering variables to indicate that the represented function is asserted if the variable is asserted. No entry indicates that the function is not asserted if the variable is asserted. The process for finding a minimal representation of a Boolean function using an extended K-map is (1) to find the minimal cover with the extension variables de-asserted, then (2) for each variable, to separately find the minimal sum with all 1s changed to x in the map, and all other variables set to 0, and form the product of the minimal sum and the extension variable. Form the sum obtained by combining (1) with the sum of the results of (2). The result is a minimal representation if the extension variables can be assigned independently.

Example 2.27

The K-map in Figure 2-30(a) indicates where function F asserts independently of variables f and e, and where it asserts with them. In Figure 2-30(b), we consider logical adjacencies with f and e both set to 0, then in Figure 2-30(c), we show f asserted with all

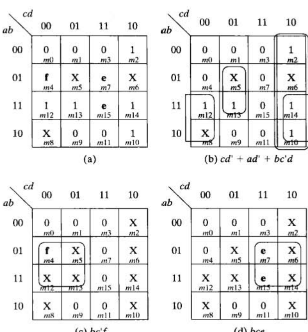


FIGURE 2-30 Extended Karnaugh maps: (a) F asserts independently of f and e, and with them, (b) logical adjacencies with don't-cares and with e and f both 0, (c) f asserted, and x replacing all 1s of the original map, and (d) with e asserted, f de-asserted, and all 1s of the original map set to x.

1s of the original map replaced by an x, and in Figure 2-30(d), we show variable e asserted with f de-asserted, and all 1s of the original map set to x. The sum of the cubes obtained from these steps gives $F = cd' + ad' + bc'd + bc'f + bce$. *Note*: fe is contained in e and f, so the simultaneous assertion of f and e has also been considered.

End of Example 2.27

2.5 Glitches and Hazards

The output of a combinational circuit may make a transition even though the logical values applied at its inputs do not imply a change. These unwanted switching transients are called "glitches." Glitches are a consequence of the circuit structure, the delays of an actual implementation, and the application of patterns that cause the glitch to occur. A circuit in which a glitch may occur under the application of appropriate inputs signals is said to have

1-Hazard 0-Hazard

FIGURE 2-31 Waveforms produced by a circuit with (a) a static 1-hazard, and (b) a static 0-hazard.

a "hazard." If a circuit has a hazard it could exhibit a glitch under certain conditions. There are two types of hazards, *static* and *dynamic*. The term *static* refers to a circuit in which the output should not change under the application of certain inputs, but does.

A circuit has a *static 1-hazard* if an output has an initial value of 1, and an input pattern that does not imply an output transition causes the output to change to 0 and then return to 1. A circuit has a *static 0-hazard* if an output has an initial value of 0, and an input pattern that does not imply an output transition causes the output to change to 1 and then return to 0. The waveforms that result from these hazards are shown in Figure 2-31. Whether a static hazard occurs or not depends on the application of the appropriate input pattern.

Static hazards are caused by differential propagation delays on reconvergent fanout paths. The signal applied at C in Figure 2-32 reconverges at the OR gate whose output is F. Thus, the signal propagates along two different paths to reach the output. The logic that forms the signals that arrive at the inputs of the gate implies that the two inputs are complementary. However, if the propagation delays along the signal paths are different, the output will have a hazard. Hazards might not be significant in a synchronous sequential circuit if the clock period can be extended. A hazard is problematic if the signal serves as the input to an asynchronous subsystem (e.g., a counter or a reset circuit).

The steps that form a "minimal" realization of a circuit do not imply that it will be hazard free. If hazards are problematic, then more work needs to be done to detect and remove them. Static hazards can be eliminated by introducing redundant cubes in the cover of the output expression (the added cubes are called a "hazard cover"). Note that the treatment of hazards assumes that the output glitch is caused by the transition of a single bit of an input signal. Methods that eliminate hazards in two-level and multilevel circuits apply only if this condition is satisfied.

Consider the circuit shown in Figure 2-32, where $F = AC + BC'$. If the initial inputs to the circuit are $A = 1, B = 1, C = 1$, the output is $F = 1$. Next, if the inputs are $A = 1, B = 1, C = 0$, the output should still be $F = 1$. In a physical realization of the circuit (i.e., nonzero propagation delays), the delay of the path to $F1$ will be greater than the delay of the path to $F0$, because the signal travels through an additional gate, causing a change in C to reach $F1$ later than it reaches $F0$ (the path with greater delay is said to be longer), that is, AC de-asserts before BC' asserts. Consequently, when C changes from 1 to 0, the output undergoes a single, momentary transition to 0 and then returns to 1. The presence of a static hazard is apparent in the simulated waveforms in Figure 2-33 and in the Karnaugh map of the output signal shown in Figure 2-34.[2]

[2]We will consider simulation of digital logic in Chapter 4. The simulator used to obtain the results in Figure 2.33 is bundled on the CD-ROM that accompanies this book.

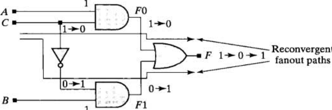


FIGURE 2-32 A circuit with reconvergent fanout and a static 1-hazard.

The Karnaugh map in Figure 2-34 reveals how a change of input C from 1 to 0 in the circuit of Figure 2-32 causes the cube AC to de-assert and the cube BC' to assert. However, AC de-asserts before BC' can assert. In the circuit of Figure 2-32, for example, the hazard occurs because the cube AC is initially asserted, while BC' is not. The hazard can be removed by adding a redundant cube, AB, to cover the adjacent 1s of the adjacent prime implicants associated with the hazard. The redundant cube is referred to as a "hazard cover." It eliminates the dependency of the output on the input C (the boundary between the cubes is now covered). *The cover of a hazard introduces redundant logic and requires additional hardware.*

Example 2.28

The hazard-free cover of the circuit in Figure 2-32 is given by $F = AC + BC' + AB$. The circuit-level realization of the covered function in Figure 2-35 has an additional AND gate.

End of Example 2.28

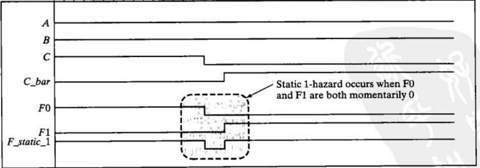


FIGURE 2-33 Results of simulating the circuit shown in Figure 2.32, which has reconvergent fanout and a static 1-hazard.

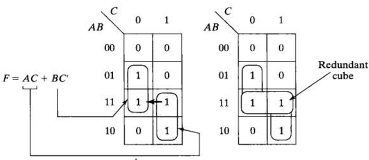


FIGURE 2-34 Karnaugh map of the logic for the circuit shown in Figure 2.32, which has reconvergent fanout and a static 1-hazard. *Note:* The arrow indicates a transition that causes the hazard to occur.

2.5.1 Elimination of Static Hazards (SOP Form)

When a signal changes value at the input of a circuit, a static 1-hazard could occur at the output of the circuit if three conditions are satisfied: (1) an output remains asserted (i.e., its value is 1 before and after the input signal changes value), (2) the cube that is asserted in the SOP expression of the output by the initial value of the signal is different from the cube that is asserted by the final value of the signal, and (3) the cubes that are asserted by the initial and final values of the signal are not covered by the same prime implicant. If the output cubes asserted by the initial and final

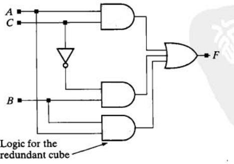


FIGURE 2-35 Circuit modified to remove a static 1-hazard.

values of the input signal are covered by the same prime implicant, a glitch cannot occur when the input signal changes value. Whether a static 1-hazard actually occurs depends on the accumulated delays along the signal propagation path from the inputs to the output.

If a static 1-hazard can be caused by changing the value of a single input signal, the cell that is asserted by the initial value of the input signal must be logically adjacent to the cell that is asserted by the final value of the input signal, because only one input signal is allowed to change. Consequently, the addition of a redundant cube covering both cells will cover the boundary between them, and cover the hazard. So, to eliminate a static 1-hazard caused by changing the value of a single input signal, form an SOP cover that covers every pair of adjacent 1s that reside in adjacent cubes. This guarantees that every single-bit input change is covered by a prime implicant. The set of such prime implicants is a hazard-free cover for a two-level (AND–OR) realization of the circuit, but a better alternative might be found, and should be sought.

Example 2.29

The expression $f = \Sigma\, m(0, 1, 4, 5, 6, 7, 14, 15) = a'c' + bc$, whose K-map is shown in Figure 2-36, has a static 1-hazard, because f is asserted by the cubes $a'c'$ and bc, and with $a = 0$, $b = 1$, and $d = 1$, a glitch can occur as c changes from *1* to *0* or vice versa. Note that adjacent cells of adjacent cubes are asserted, depending on whether $c = 0$ or $c = 1$. The hazard can be removed by adding either the cube $a'bd$ or $a'b$ to the expression. Both are redundant prime implicants, but $a'b$ is chosen to given a minimal resulting expression: $f = a'c' + bc + a'b$. Also, observe that the redundant cube that is

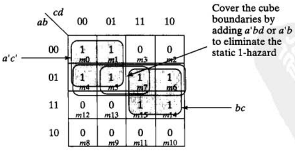


FIGURE 2-36 Cover of a static 1-hazard.

added to the SOP expression to cover the hazard does not depend on the input signal whose change causes the hazard.

There are two approaches to eliminating a static 0-hazard. The first detects where the transition of a single input causes a transition across the boundary between adjacent prime implicants and adds redundant prime implicants to f' as needed. The second method (1) eliminates the static 1-hazards of f, (2) considers whether the implicants of the 0s of the expression that is free of static 1-hazards also cover all adjacent 0s of the original function, and then (3) adds redundant prime implicant factors to the complement of the static 1-hazard-free expression in POS form, as needed.

End of Example 2.29

Example 2.30

The Karnaugh map shown in Figure 2-37 for the function $f = a'c' + bc$ (see Example 2.29) has a static 0-hazard, because f is de-asserted in cubes ac' and $b'c$, and with $a = 1, b = 0$, and $d = 1$, switching c from 1 to 0, or vice versa, crosses a boundary between adjacent 0s in adjacent de-assertion cubes of f. By following the first method for eliminating a static 0-hazard, we consider the 0s of the K-map in Figure 2-37 and apply DeMorgan's law to obtain $f = (a' + c)(b + c')$. We cover the hazard by adding ab' to f', and including the redundant prime implicant product factor $(a' + b)$ in the POS form of f. The factor $ab'd$ would also cover the hazard, but it is not minimal. The resulting, equivalent, hazard-free POS expression is $f = (a' + c)(b + c')(a' + b)$.

Also observe that, in this example, the POS expression that eliminates the static 0-hazard is equivalent to the expression that eliminates the static 1-hazard, for

$$f = (a' + c)(b + c')(a' + b)$$

$$= a'ba' + a'bb + a'c'a' + a'c'b + cba' + cbb + cc'a' + cc'b$$

$$= a'b + a'c' + bc$$

End of Example 2.30

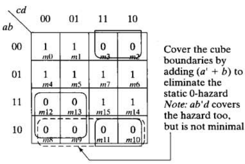


FIGURE 2-37 Cover of the 0s in the K-map of a static 0-hazard in $f = a'c' + bc$.

Example 2.31

The alternative method for eliminating a static 0-hazard from the expression given in Example 2.29 begins with the static 1-hazard-free function: $f = a'c' + bc + a'b$. Now consider the K-map for $f' = (a + c)(b' + c')(a + b')$ and examine it for coverage of the 0s in the K-map of the original function:

$$f' = ab'a + ab'b' + ac'a + ac'b + cb'a + cb'b' + cc'a + cc'b'$$
$$= ab' + ac' + b'c$$

The K-maps of complement of the static 1-hazard-free function, and the original function are shown in Figure 2-38.

All the adjacent 0s of the K-map for $f = a'c' + bc$ are covered by 0s of the complement of the static 1-hazard-free function, $f' = ab' + ac' + b'c$, and no boundary at which a transition could occur between adjacent 0-cubes is uncovered. Therefore there is no static 0-hazard.

In this example, there are no static 0-hazards or static 1-hazards in the expression with the added redundant cube. In general, eliminating the static 1-hazards might not eliminate the static 0-hazards.

End of Example 2.31

2.5.2 Summary: Elimination of Static Hazards in Two-Level Circuits

The methodology for eliminating static 1-hazards in a two-level circuit is (1) to cover with prime implicants all 1s in adjacent cells of adjacent cubes of the K-map of the SOP form of the function and (2) to add redundant prime implicants as needed to complete

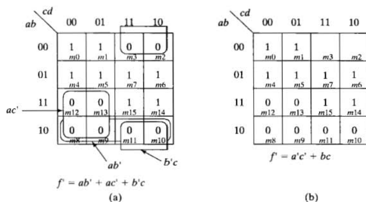


FIGURE 2-38 Karnaugh maps illustrating the 0s of (a) $f' = ab' + ac' + b'c$, and (b) $f = a'c' + bc$.

the cover of the function. To cover a static 0-hazard, we cover all adjacent 0s in the POS form of the static 1-hazard-free function, adding prime implicants in the POS form of the static 1-hazard function as needed to cover any uncovered adjacencies.

2.5.3 Static Hazards in Multilevel Circuits

Like two-level circuits, multilevel circuits are subject to static hazards, but the outputs of multilevel circuits are not written in SOP or POS forms, which have two levels of logic. In a multilevel circuit, there may be multiple paths from an input to an output of the circuit, with each path having a different propagation delay. When a circuit has propagation delays, a Boolean variable and its complement might not change value at exactly the same time. For example, a transition in the variable will precede a transition in its complement, a', by the propagation delay of the inverter whose input is a and whose output is a'. Thus, the Boolean cube aa' has a transient interval over which its value is not 0, and the circuit could have a static 0-hazard. Similarly, the factor $(b + b')$ might have a transient during which the value of the factor is 0, rather than 1.

The Boolean expression for an output of a multilevel circuit can always be flattened into a two-level form by multiplying its product factors. To eliminate a static hazard in a multilevel circuit, we begin by flattening the multilevel description of the output expression into an SOP form, f_{tof}, called the "transient output function" [4], taking care not to eliminate either the product or the sum of a literal with its complement. Each input variable and its complement are treated as independent variables in f_{tof}. For example, we do not cancel aa' from an SOP form and do not cancel a factor like $a + a'$ in a POS form. Preserving factors such as aa' and $(a + a')$ exposes the possible transients in which the indicated variables are not the complement of each other. The presence of a product of a variable and its complement reveals a static 0-hazard in that input; a sum of a variable and its complement indicates a static 1-hazard in that input. After forming the transient output function, check for static 1-hazards in the two-level expression (terms such as aa' can be

ignored in this check), and add redundant prime implicants to cover adjacent 1s in the K-map. Then check to see whether the 0s of the static 1-hazard-free function cover the 0s of the original function. Introduce redundant terms as needed to create a hazard-free cover (terms such as aa' reveal the variable that causes a static 0-hazard).

Example 2.32

Consider the possibility of a static 1-hazard in the multilevel function

$$f = bcd + (a + b)(b' + d') = bcd + ab' + ad' + bb' + bd'$$

with the transient output function:

$$f_{tof} = bcd + ab' + ad' + bd'$$

Note that f_{tof} does not include the cube bb'—it implies a static 0-hazard and has no influence on a possible 1-hazard. The K-map of f_{tof} without the bb' term is shown in Figure 2-39.

The transient output function has three static 1-hazards, that is, three cube boundaries across which the transition of a single input might cause a hazard, depending on the propagation delays of the circuit. The three possibilities are listed below, where $(1111) \leftrightarrow (1011)$ denotes a transition of $(abcd)$ between initial and final values.

$$(a, b, c, d) = (1111) \leftrightarrow (1011)$$
$$(a, b, c, d) = (1111) \leftrightarrow (1110)$$
$$(a, b, c, d) = (0111) \leftrightarrow (0110)$$

By adding two additional cubes, bc and ac, to f, we cover the hazards and form the static 1-hazard-free expression, f_{1HF}:

$$f_{1HF} = bcd + ab' + ad' + bd' + bc + ac$$

The final, minimal, form is obtained by removing the redundant cube bcd:

$$f_{1HF} = ab' + ad' + bd' + bc + ac$$

Next, we illustrate the removal of a static 0-hazard in a multilevel circuit.

End of Example 2.32

Example 2.33

The multilevel function $f = bcd + (a + b)(b' + d')$ has its complement given by

$$f' = [bcd + (a + b)(b' + d')]' = [bcd]' [(a + b)(b' + d')]'.$$

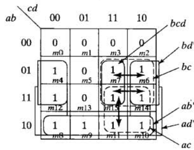


FIGURE 2-39 Karnaugh map of the transient output function $f_{tof} = bcd + ab' + ad' + bd'$.

By applying DeMorgan's law to the above expression we obtain

$$\begin{aligned} f' &= (b' + c' + d')(a'b' + bd) \\ &= a'b' + a'b'c' + bc'd + a'b'd' \\ &= a'b' + bc'd \end{aligned}$$

and

$$f = (a'b' + bc'd)' = (a + b)(b' + c + d')$$

The cubes of the expression for f' indicate where f will be 0. Now consider the 0s in the K-map of f, as shown in Figure 2-40.

The boundary between the cubes of the (logically and physically) adjacent 0s in the map indicates that a static 0-hazard exists when the inputs make the transitions $(a, b, c, d) = (0101) \leftrightarrow (0001)$. We add to f' a redundant cube, $a'c'd$, to cover the hazard and form the complement of a static 0-hazard-free function, f'_{0HF}. The augmented expression becomes

$$f'_{0HF} = a'b' + bc'd + a'c'd$$

and so f_{0HF} has the POS form

$$f_{0HF} = (a + b)(b' + c + d')(a + c + d')$$

The final expression for f_{0HF} is free of static 0-hazards. Also, using the results obtained in Example 2.32, observe that

$$f_{0HF} = ab' + ad' + bd' + bc + ac = f_{1HF}$$

We conclude that f_{0HF} and f_{1HF} are both free of static 0-hazards and static 1-hazards.

End of Example 2.33

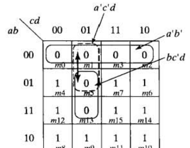


FIGURE 2-40 0s of the K-map of $f = (a'b' + bc'd)'$.

2.5.4 Summary: Elimination of Static Hazards in Multilevel Circuits

Static hazards can be eliminated in multilevel circuits by (1) forming f_{tof}, the transient output function, by collapsing the multilevel logic into an SOP form (while ignoring complement relationships, e.g., aa'), (2) covering every group of adjacent 1s of f_{tof} in the K-map to form f_1, a function free of static 1-hazards, (3) applying DeMorgan's law to f_1 and simplifying with Boolean relationships (treating each variable and its complement as independent variables), and (4) forming f_0 in SOP form by covering any groups of adjacent 0s. If no term of the resulting expression contains the product of a variable and its complement, the expression will be free of static 1-hazards and static 0-hazards.

2.5.5 Dynamic Hazards

A circuit has a dynamic hazard if an input transition is supposed to cause a single transition in an output, but causes it two or more transitions before it reaches its expected value. Typical waveforms of a dynamic hazard are shown in Figure 2-41. Such hazards are a consequence of multiple static hazards caused by multiple reconvergent paths in a multilevel circuit. They are not easy to eliminate, but if a circuit is free of all static hazards, it will be free of dynamic hazards. Consequently, a method for eliminating dynamic hazards is to (1) transform the circuit into a two-level form, and then (2) detect and eliminate all static hazards.

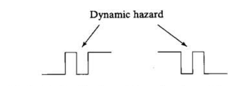


FIGURE 2-41 Waveforms illustrating dynamic hazards.

Example 2.34

The circuit in Figure 2-42 has two nodes where input C reconverges. The output *F_static* has a static hazard, and *F_dynamic* (the location of the second reconvergence) has a dynamic hazard. The simulation results in Figure 2-43 display the effect of the hazards.

The Karnaugh map for *F_static* is shown in Figure 2-44, where it is apparent that the redundant prime implicant, AB, covers the boundary between cubes BC' and AC.

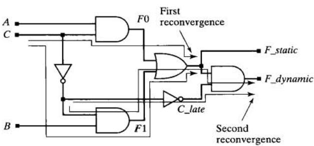


FIGURE 2-42 A circuit having two nodes of reconvergence, a static hazard, and a dynamic hazard.

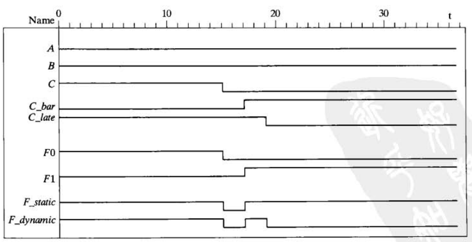


FIGURE 2-43 Simulation results showing the effects of static and dynamic hazards.

The redundant cube eliminates the static 1-hazard and assures that *F_dynamic* will not depend on the arrival of the effect of the transition in *C*. The additional logic for the redundant cube is shown in Figure 2-44. The hazard-free circuit and its simulated waveforms are shown in Figures 2-45 and 2-46, respectively.

End of Example 2.34

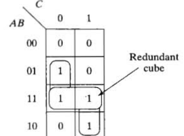


FIGURE 2-44 Karnaugh map of *F_static* in Figure 2-42.

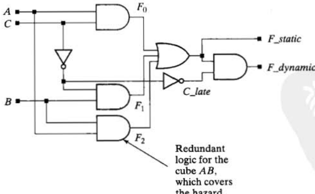


FIGURE 2-45 Hazard-free equivalent of the circuit in Figure 2-42. Redundant logic forming F_2 has been added to the original circuit.

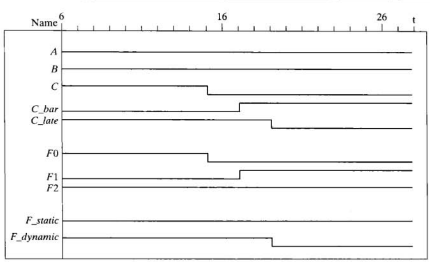


FIGURE 2-46 Simulation results for the hazard-free circuit in Figure 2-45.

2.6 Building Blocks for Logic Design

Combinational logic encompasses a wide range of functionality and circuit structures, but certain structures and circuits are commonly used in many applications, and it is worthwhile to gain familiarity with them.

2.6.1 NAND–NOR Structures

In CMOS technology, AND gates and OR gates are not implemented as efficiently as NAND gates and NOR gates. An SOP form or a POS form can always be converted to a NAND logic structure or a NOR logic structure. The NAND gate and NOR gate are universal logic gates—any Boolean function can be realized from only NAND gates or only NOR gates. DeMorgan's laws provide equivalent structures for NAND and NOR gates, shown in Figure 2-47.

Networks realizing SOP expressions[3] can be transformed by DeMorgan's laws into a circuit that uses only NAND gates and inverters by (1) replacing AND gates by NAND gates in the original AND–OR structure, (2) placing inversion bubbles at the inputs of the OR gates, (3) inserting inverters where needed to match bubbles at the inputs of the OR gates, and (4) substituting a NAND gate for a NOR gate that

[3]The output of the network must be the output of an OR gate.

Gate	DeMorgan equivalent
a, b → y $y = (a \cdot b)'$	a, b → y $y = a' + b'$
a, b → y $y = (a + b)'$	a, b → y $y = a' \cdot b'$

FIGURE 2-47 Equivalent circuits that result from DeMorgan's laws.

has inversion bubbles at its inputs. A circuit in POS form[4] can be transformed into an equivalent circuit that uses only NOR gates and inverters by (1) replacing OR gates by NOR gates, (2) placing inversion bubbles at the inputs of the AND gates, (3) inserting inverters where needed to match bubbles at the inputs of the AND gates, and (4) substituting a NOR gate for a NAND gate that has inversion bubbles at its inputs.

Example 2.35

The function $Y = G + EF + AB'D + CD$ has the two-level circuit realization shown in Figure 2-48(a), which can be transformed into the circuit in Figure 2-48(b) by placing bubbles at the inputs to the OR gate that forms Y and an inverter at input G to match the bubble at the input to the OR gate driven by G. Then DeMorgan's law is applied to replace the OR gate with input bubbles by the NAND gate shown in Figure 2-48(c).

To verify that that circuit of (c) is equivalent to that of (a), we note that

$$\begin{aligned} Y &= [(G')(EF)'(AB'D)'(CD)']' \\ &= (G')' + [(EF)']' + [(AB'D)']' + [(CD)']' \\ &= G + EF + AB'D + CD \end{aligned}$$

End of Example 2.35

[4]The output of the network must be the output of an AND gate.

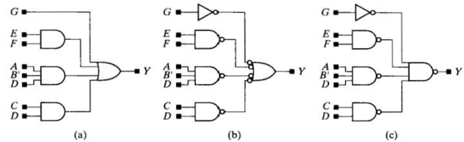


FIGURE 2-48 Circuit transformations to obtain a NAND/Inverter realization of an SOP expression.

Example 2.36

Now consider the POS expression $Y = D(B + C)(A + E + F')(A + G)$. The circuit in Figure 2-49(b) is formed by replacing OR gates with NOR gates, adding inversion bubbles to the input of the AND gate, and adding an inverter to D to match its input bubble. Then the NAND gate with inversion bubbles at its inputs is replaced by an equivalent NOR gate to form the circuit in Figure 2-49(c).

A check reveals that the transformed circuit is equivalent to the original circuit:

$$\begin{aligned} Y &= [D' + (B + C)' + (A + E + F')' + (A + G)']' \\ &= (D')'[(B + C)']'[(A + E + F')']'[(A + G)']' \\ &= D(B + C)(A + E + F')(A + G) \end{aligned}$$

A circuit whose structure does not consist of alternating AND gates and OR gates can still be transformed into an equivalent structure that uses only NAND gates and inverters or a structure that uses only NOR gates and inverters. To transform such a circuit into a NAND structure, (1) replace all AND gates by NAND gates (Figure 2-50(a), (2) place inversion bubbles at the inputs of all OR gates (Figure 2-50(b)), and (3) replace

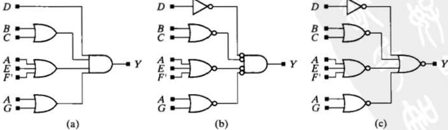


FIGURE 2-49 Circuit transformations to obtain a NOR/Inverter realization of a POS expression.

OR gates that have inversion bubbles on their inputs with DeMorgan-equivalent NAND gates (Figure 2-50(c)). If, after these changes have been made, the output of a NAND gate drives the input of another NAND gate, place an inverter at the input of the driven NAND gate (Figure 2-50(d)); if the output of an OR gate having bubbles at its inputs drives another OR gate that has inversion bubbles at its inputs, place an inverter on the path connecting them (see Figure 2-50(e)). Then replace OR gates with inversion bubbles at their inputs by equivalent NAND gates. These steps ensure that inversions caused by the replacement of gates will be matched and that the final circuit will be equivalent to the original circuit.

Alternatively, to transform the circuit into a NOR structure, (1) replace OR gates by NOR gates (Figure 2-51(a)), (2) place inversion bubbles at the inputs of any AND gates (Figure 2-51(b)), and (3) replace AND gates with bubble inputs by DeMorgan-equivalent

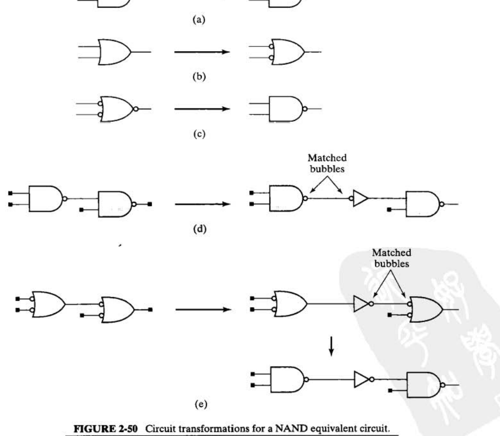


FIGURE 2-50 Circuit transformations for a NAND equivalent circuit.

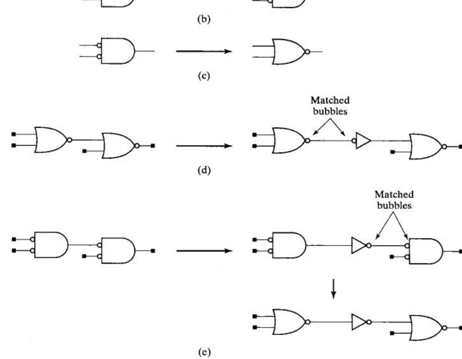


FIGURE 2-51 Circuit transformations for a NOR equivalent circuit.

NOR gates (Figure 2-51(c)). If, after these changes have been made, the output of a NOR gate drives the input of another NOR gate, place an inverter at the input of the driven NOR gate (Figure 2-51(d)); if the output of an AND gate with bubbles at its inputs drives another AND gate that has inversion bubbles at its inputs, place an inverter on the path connecting them (see Figure 2-51(e)). Then replace the AND gate that has inversion bubbles at its inputs with an equivalent NOR gate.

These rules ensure that inversion bubbles will be matched and guarantee that the transformed (NAND or NOR) circuit is equivalent to the original circuit.

End of Example 2.36

2.6.2 Multiplexers

Multiplexer circuits are used to steer data through functional units of computers and other digital systems. For example, a multiplexer can be used to steer the contents of a particular storage register to the inputs of an arithmetic and logic unit (ALU) and to steer the output of the ALU to the same or a different register. A gate-level schematic of a two-channel multiplexer is shown in Figure 2-52. When $sel = 0$ the data at input a passes through the circuit (with some propagation delay) to *y_out*; likewise, if $sel = 1$ the data at input b goes to *y_out*. The Boolean expression describing the function of the circuit is given by: $y_out = sel' \cdot a + sel \cdot b$.

In general, a multiplexer has n datapath input channels and a single output channel. An m-bit address determines which input channel is connected to the output channel. The input channel selected by the multiplexer shown symbolically in Figure 2-53 is governed by $Data_Out = Data_In$ [Address[k]], where k is an index into the address space.

Multiplexers can also be used to implement combinational logic. The values of a Boolean function can be assigned to the input lines and decoded by the select lines. This implementation might be inefficient because the mux must fully decode the truth table of all of the input bits.

Example 2.37

The truth table in Figure 2-54 describes a 4-bit majority function, which asserts its output if a majority of its inputs are asserted. The schematic shows how to implement the function with a 16-input mux, using its four select lines to decode the possible bit patterns of the inputs to the function.

End of Example 2.37

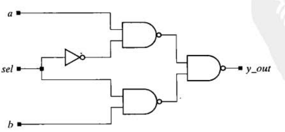


FIGURE 2-52 Gate-level schematic for a two-channel multiplexer circuit.

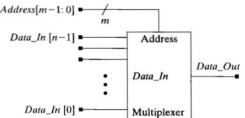
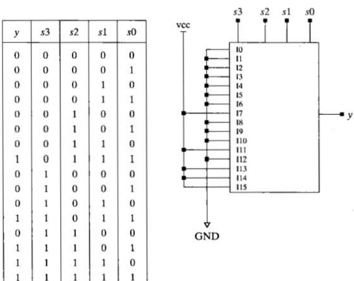


FIGURE 2-53 Schematic symbol for an n-channel multiplexer with an m-bit channel selector address.



y	$s3$	$s2$	$s1$	$s0$
0	0	0	0	0
0	0	0	0	1
0	0	0	1	0
0	0	0	1	1
0	0	1	0	0
0	0	1	0	1
0	0	1	1	0
1	0	1	1	1
0	1	0	0	0
0	1	0	0	1
0	1	0	1	0
1	1	0	1	1
0	1	1	0	0
1	1	1	0	1
1	1	1	1	0
1	1	1	1	1

FIGURE 2-54 Truth table and circuit for a 16-input mux implementation of a 4-bit majority function.

2.6.3 Demultiplexers

A demultiplexer circuit implements the reverse functionality of a multiplexer. It has a single input datapath, n output datapaths, and an input m-bit address that determines which of the n outputs is connected to the input. The output channel selected by the demultiplexer shown in Figure 2-55 is determined by *Data_Out* $[n - 1{:}0]$ = *Data_In* [Address[k]], where k is an index into the address space.

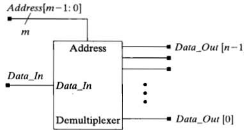


FIGURE 2-55 Gate-level schematic for an n-output demultiplexer circuit having an m-bit destination address.

2.6.4 Encoders

Multiplexer and demultiplexer circuits dynamically establish connectivity between datapaths in a system. A data pattern that passes through a multiplexer or a demultiplexer is not altered by the circuit. On the other hand, an encoder circuit acts to transform an input data word into a different output data word. Input data words are typically wide in comparison to the encoded output word, so encoders serve to reduce the size of a datapath in a system. An encoder assigns a unique bit pattern to each input line. Usually, a device whose output code is smaller than its input code is referred to as an encoder. If the size of the output word is larger than the size of the input word, the circuit is referred to as a decoder. One typical application of an encoder is in a client–server polling circuit whose output code indicates which of n clients requesting service from a server is to be granted service.

An encoder has n inputs and m outputs, with $n = 2^m$. An encoder could transform up to 2^m different input words into unique output codes, treating the remaining input patterns as don't-care conditions, but ordinarily only one of the inputs is asserted at a time, and a unique output bit pattern (code) is assigned to each of the n inputs. The asserted output is determined by the index of the asserted bit of the n-bit binary input word. Block diagram symbols for encoders are shown in Figure 2-56.

Example 2.38

A 5:3 encoder having a 5-bit input word is to generate a 3-bit output code indicating the number of bits that are asserted in the input word. The input words and encoded output bit patterns are shown in Figure 2-57. Boolean logic equations can be derived for each bit of the output word.

End of Example 2.38

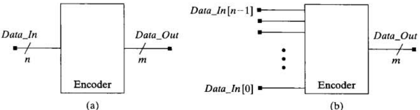


FIGURE 2-56 Schematic symbols for an encoder: (a) an encoder with an n-bit input bus and (b) an encoder with individual bit-line inputs.

Input	Output	Input	Output
00000	000	10000	001
00001	001	10001	010
00010	001	10010	010
00011	010	10011	011
00100	001	10100	010
00101	010	10101	011
00110	010	10110	011
00111	011	10111	100
01000	100	11000	010
01001	010	11001	011
01010	010	11010	011
01011	011	11011	100
01100	010	11100	011
01101	011	11101	100
01110	011	11110	100
01111	100	11111	101

FIGURE 2-57 Input–output words for a 5:3 encoder that indicates the number of asserted bits in the input word.

2.6.5 Priority Encoder

A priority encoder allows multiple input bits to be asserted simultaneously and uses a priority rule to form an output bit pattern. A priority encoder in a client–server system would identify the client that has the highest priority among multiple clients requesting service.

Example 2.39

The input–output patterns for an eight-client priority encoder are shown in Figure 2-58. (*Note*: *X* denotes a don't-care condition.) The client corresponding to the leftmost 1-bit of the input word has highest priority. This is a combinational scheme; a sequential machine could impose some rule providing all clients with some level of service.

Input word	Output word
1xxxxxxx	000
01xxxxxx	001
001xxxxx	010
0001xxxx	011
00001xxx	100
000001xx	101
0000001x	110
00000001	111

FIGURE 2-58 Input–output words for an 8 : 3 priority encoder.

End of Example 2.39

2.6.6 Decoder

A binary decoder interprets an input pattern of bits and forms a unique output word in which only 1 bit is asserted. Decoders are commonly used to extract the opcode from an instruction in a digital computer; row and column address decoders are used to locate a word in memory from its address. Figure 2-59 shows block diagram symbols for a decoder.

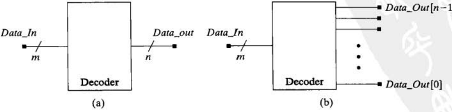


FIGURE 2-59 Block diagram symbols for decoders: (a) decoder with input/output busses and (b) decoder with an expanded output bus.

A binary decoder has m inputs and n outputs, with $n = 2^m$. There are many different possible mappings between input words and output words. An encoder can be built from combinational logic that forms the input–output mapping. (Sequential encoders and decoders are widely used in communication and video transmission circuits.)

Example 2.40

The input–output patterns for an eight-client decoder are shown in Figure 2-60. The arrangement of bits in the output words identifies the client that will be served. This decoder does not resolve contention between multiple clients, and it assumes that only one client at a time will request service.

A binary decoder generates all of the minterms of its inputs. All of the output lines are available to form as many functions of the same inputs as are needed by an application. Binary decoders can be used for small implementations with multiple outputs, but the number of outputs precludes their use in applications with a large number of inputs.

Input word	Output word
000	10000000
001	01000000
010	00100000
011	00010000
100	00001000
101	00000100
110	00000010
111	00000001

FIGURE 2-60 The input–output patterns for an eight-client decoder.

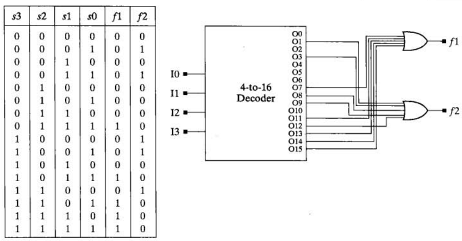
End of Example 2.40

Example 2.41

A decoder can be used to implement multiple Boolean functions of the same inputs. The truth table in Figure 2-61 describes $f1$, a majority function, along with some other function, $f2$. Additional logic is used with the decoder to combine its outputs to form the two functions.

End of Example 2.41

PDG



s3	s2	s1	s0	f1	f2
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	0	0
0	1	1	0	0	0
0	1	1	1	1	0
1	0	0	0	0	1
1	0	0	1	0	1
1	0	1	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	0	1	1	0
1	1	1	0	1	0
1	1	1	1	1	0

FIGURE 2-61 A truth table for two functions implemented by a single 4-to-16 decoder.

2.6.7 Priority Decoder

A priority decoder can be used in applications in which multiple input codes might imply contention.

Example 2.42

The input code for a client–server system that is to serve eight clients will contain a 1 in any bit position that corresponds to a request from a client for service. The server must determine which of multiple clients is to be served. One simple rule assigns a unique priority to each client. The input–output codes in Figure 2-62 assign the highest priority to the client associated with the leftmost bit of the input code. The table accounts for all possible input patterns. A sequential decoder circuit could base service on other considerations, such as whether a client has been blocked from service by higher-priority clients for too long.

End of Example 2.42

Input Word	Output Word
1xxxxxxx	10000000
01xxxxxx	01000000
001xxxxx	00100000
0001xxxx	00010000
00001xxx	00001000
000001xx	00000100
0000001x	00000010
00000001	00000001

FIGURE 2-62 The input–output patterns for an eight-client priority decoder.

REFERENCES

1. Breuer MA, Friedman AD. *Diagnosis and Design of Reliable Digital Systems*. Rockville, MD: Computer Science Press, 1976.
2. Fabricius ED. *Introduction to VLSI Design*. New York: McGraw-Hill, 1990.
3. Bryant RE. "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers, C-35*, 677–691, 1986.
4. Tinder RF. *Engineering Digital Design*, 2nd ed. San Diego, CA: Academic Press, 2000.
5. Katz RH. *Contemporary Logic Design*, 2nd ed. Upper Saddle River, NJ: Prentice-Hall, 2004.
6. McCluskey EJ. "Minimization of Boolean Functions," *Bell Systems Technical Journal, 35*, 1417–1444, 1956.
7. McCluskey EJ. *Introduction to the Theory of Switching Circuits*. New York: McGraw-Hill, 1965.
8. McCluskey EJ. *Logic Design Principles*, Upper Saddle River, NJ: Prentice-Hall, 1986.
9. Wakerly JF. *Digital Design Principles and Practices*, 4th ed. Upper Saddle River, NJ: Prentice-Hall, 2006.
10. Brayton RK. et al. *Logic Minimization Algorithms for VLSI Synthesis*. Boston, MA: Kluwer, 1984.

PROBLEMS

1. Find the canonical SOP form of the following Boolean function: $F(a, b, c) = \Sigma\, m(1, 3, 5, 7)$.
2. Find the canonical POS form of the following Boolean function: $F(a, b, c, d) = \Pi\, M(0, 1, 2, 3, 4, 5, 12)$.
3. Express the function $F = a'b + c$ as a sum of minterms.
4. Express the function $F = a'bcd' + a'bcd + a'b'c'd' + a'b'c'd$ as (a) a sum of minterms and (b) a product of maxterms.

5. Express the function $G = (a'bcd' + a'bcd + a'b'c'd' + a'b'c'd)'$ as a sum of minterms.
6. Find a NAND circuit realization of the function $f = ac' + bcd + a'd$.
7. Find a NOR circuit realization of the function $f = (b + c + d)(a' + b + c)(a' + d)$.
8. Find the complement of the following expressions:
   a. $ab' + a'b$
   b. $b + (cd' + e)a'$
   c. $(a' + b + c)(b' + c')(a + c)$
9. Simplify the following Boolean functions to a minimum number of literals:
   a. $F = a + a'b$
   b. $F = a(a' + b)$
   c. $F = ac + bc' + ab$
10. Using Karnaugh maps, simplify the following Boolean functions:
    a. $F(a, b, c) = \Sigma\, m(0, 2, 4, 5, 6)$
    b. $F(a, b, c) = \Sigma\, m(2, 3, 4, 5)$
    c. $F(a, b, c) = bc' + ac' + a'bc + ab$
    d. $F(a, b, c, d) = \Sigma\, m(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$
    e. $F(a, b, c, d) = a'b'c' + b'cd' + a'bcd' + ab'c'$
11. Find a NAND gate realization of the following Boolean function: $F(a, b, c) = \Sigma\, m(0, 6)$.
12. Using the K-map diagram below:
    a. Draw a K-map for $f = \Sigma m(0, 4, 6, 8, 9, 11, 12, 14, 15)$
    b. Identify the prime implicants of f.
    c. Identify the essential prime implicants of f.
    d. Find all minimal expressions of f and identify those that use only essential prime implicants.

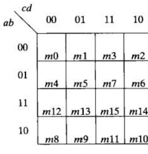


FIGURE P2-12

13. Design a two-level circuit that implements a 4-bit majority function, that is, the output is a 1 if three or more inputs are asserted.
14. Example 2.37 showed how to implement a 4-bit majority function with a 16-input mux. Show that it is also possible to implement this function with a 8-input mux.

CHAPTER 3

Fundamentals of Sequential Logic Design

Computers and other digital systems that have memory or that execute a sequence of operations under the direction of stored information are referred to as *sequential machines*, and their circuitry is modeled by sequential logic. Sequential machines do not follow combinational logic because the outputs of a sequential machine depend on the history of the applied inputs as well as on their present value.

The history of the inputs applied to a sequential machine is represented by the state of the machine and requires hardware elements that store information; that is, it requires memory to store the state of the machine as an encoded binary word. For example, a machine whose output is the running count of the number of 1s encountered by a receiver of a serial bit stream must have storage elements to hold the value of the count. Today's electronic systems rely on transistor circuits to store information. Transistors are small, are easy to fabricate, operate reliably, and they have two states, on or off, which can be used to develop voltages representing logical 0 and logical 1.

Sequential machines can be deterministic or probabilistic, and synchronous or asynchronous. We will consider only synchronous, deterministic machines. A common clock acts as a synchronizing signal for the operations of a synchronous sequential machine. This establishes fixed, predictable, intervals for propagating signals through the circuit, leading to a more reliable design and a simpler design methodology. Today's synthesis tools support only synchronous circuits.

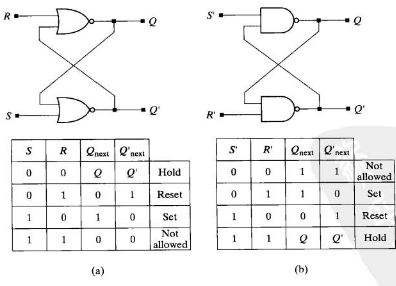
3.1 Storage Elements

Storage elements store information in a binary format—that is, as a pattern of 0s and 1s. For example, the opcode for addition in a simple microprocessor might be the

binary word pattern 0010. The circuits that store information can be level-sensitive, edge-sensitive, or a combination of both. Level-sensitive storage elements are commonly referred to as latches, and edge-sensitive storage elements are referred to as flip-flops. The outputs of a level-sensitive sequential circuit are immediately affected by a change in the value of one or more inputs, as long as an enabling signal is asserted. The outputs of an edge-sensitive circuit are sensitive to the values of the inputs, but may change value only when a synchronizing signal makes either a rising or falling edge transition.

3.1.1 Latches

The circuits in Figure 3-1 implement basic S-R (set–reset) latches. Their feedback structure of cross-coupled (a) NOR gates or (b) NAND gates enables the output (Q) of the circuits to have two stable states, 0 and 1, depending on the value of the set (S) and reset (R) inputs. Once the input conditions establish an output value, it will remain until it is changed by new input conditions. The truth tables shown with the circuits describe the new (next) state that results from a given state when an input pattern is applied while the latch is in a known state (only one input is allowed to change at a time). In practice, we avoid applying 11 to a NOR latch because the outputs of the latch will not be logical complements of each other and because (in a physical circuit)



S	R	Q_{next}	Q'_{next}	
0	0	Q	Q'	Hold
0	1	0	1	Reset
1	0	1	0	Set
1	1	0	0	Not allowed

(a)

S'	R'	Q_{next}	Q'_{next}	
0	0	1	1	Not allowed
0	1	1	0	Set
1	0	0	1	Reset
1	1	Q	Q'	Hold

(b)

FIGURE 3-1 Feedback circuit structures implementing latches: (a) cross-coupled NOR gates and (b) cross-coupled NAND gates.

a race condition occurs if the inputs are changed from 11 to 00. This makes the output unpredictable.[1] Likewise, we avoid applying 00 to a NAND latch because the outputs of the latch will not be logical complements of each other and because (in a physical circuit) a race condition occurs if the inputs are changed from 00 to 11.

3.1.2 Transparent Latches

Latches are level-sensitive storage elements; the action of data storage is dependent on the level (value) of the input clock (or enable) signal. The output of a transparent latch changes in response to the data input only while the latch is enabled; that is, changes at the input are visible at the output. A transparent latch is also called a D-latch or a data latch.

A transparent latch results from a minor change to a basic unclocked S-R latch. The latch circuit in Figure 3-2 has additional NAND gates and uses a clock signal to gate the inputs; that is, *Enable* determines whether S' and R' will have an effect on the circuit. When *Enable* is de-asserted, the circuit is not affected by the values of S' and R'. An S-R latch with gated inputs is also called a "clocked latch" and a "gated latch." The modified circuit in Figure 3-3(a) retains the signal *Enable* but passes complementary values of *Data* to the S' and R' inputs of the latch. This ensures that an unstable condition will not occur (00 will not be applied to the S-R stage) and that the value of *Q_out* will follow the value of *Data* while *Enable* is asserted. When *Enable* is de-asserted, the feedback loop ensures that the value of *Q_out* becomes fixed at its current value and is said to be latched. It remains latched until *Enable* is asserted again. The waveforms in Figure 3-3(b) illustrate the latching behavior of the circuit.

3.2 Flip-Flops

Flip-flops are edge-sensitive storage elements; the action of data storage is synchronized to either the rising or falling edge of a signal, which is commonly referred to as a clock signal. The value of data that is stored depends on the data present at the data input(s) when the clock makes a transition at its active (rising or falling) edge; at all other times the value and transitions of the data are ignored. There are various kinds of flip-flops,[2] depending on the action of additional input signals that control the storage of data, such as a reset signal [1–4].

3.2.1 D-Type Flip-Flop

A D-type flip-flop is the simplest type; at each active edge of the clock, it stores the value that is present at its D input, independently of the present stored value. A block diagram symbol and truth table for a D-type flip-flop are shown in Figures 3-4(a) and 3-4(b), respectively. The truth table includes an entry for the present state of the flip-flop (Q) and the state (Q_{next}) that will result at the next active edge of the clock signal (*clk*) for a given value of the data input (D). The waveforms shown in

[1] The race condition also leads to an indeterminate result in simulation.

[2] Only fundamental-mode flip-flops will be considered here—those in which only one input may change at a time.

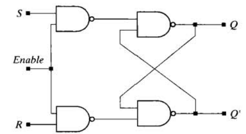


FIGURE 3-2 An S-R latch with an enabling input signal.

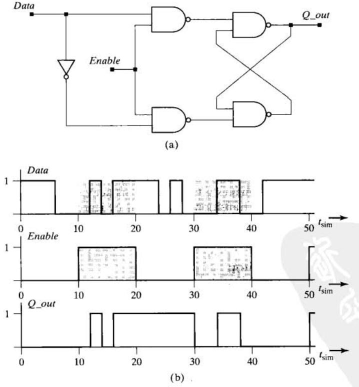
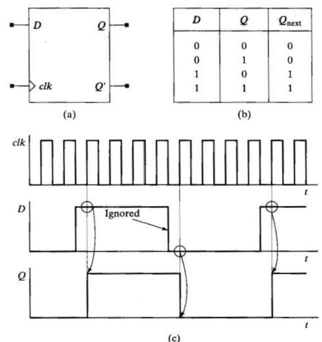


FIGURE 3-3 A transparent latch: (a) circuit schematic and (b) input–output waveforms.



D	Q	Q_{next}
0	0	0
0	1	0
1	0	1
1	1	1

FIGURE 3-4 A positive-edge-triggered D-type flip-flop: (a) block diagram symbol, (b) truth table, and (c) sample input–output waveforms.

Figure 3-4(c) illustrate how data present at D is stored, for this example, on the rising edge of *clk*, and how transitions in D are ignored over the interval between the active edges of *clk*. However, D must be stable for a sufficiently long time prior to the active edge of *clk*; otherwise, the device may not operate properly. The Boolean logic describing a D-type flip-flop obeys the following so-called characteristic equation [2], $Q_{next} = D$. AD-type flip-flop may also have other (level-sensitive) inputs, such as *set* and *reset* signals, to override the synchronous behavior and initialize the output.

3.2.2 Master–Slave Flip-Flop

A D-type flip-flop can be implemented by a master–slave configuration of two data latches, as shown in Figure 3-5. The transparent latch of the master stage samples the input during the half-cycle beginning at the inactive edge of the clock; the sampled value will be propagated to the output of the latch of the slave stage at the next active edge during the so-called slave cycle of the circuit. The output of the master stage must settle before the enabling edge of the slave stage. The master stage is enabled on the inactive edge of the clock, and the slave stage is enabled on the active edge. Setup and

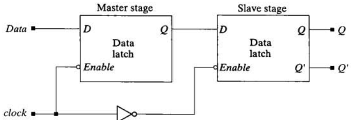


FIGURE 3-5 A master–slave implementation of a negative-edge-triggered D-type flip-flop.

hold conditions apply at the active edge of the clock (see Chapter 10). These specify conditions for stability of the data relative to the clock in order to ensure proper operation of the device.

In complementary metal-oxide semiconductor (CMOS) technology [5,6] a D-type flip-flop is commonly implemented with transmission gates. D-type flip-flops are popular because they have fewer input signal paths, and circuits using D-type flip-flops are simpler to design. A transmission gate is formed by a parallel connection of an *n*-channel transistor and *p*-channel transistor, shown in Figure 3-6 with a circuit symbol for a transmission gate. Transmission gates have symmetric noise margins in either direction of transmission and support bidirectional signal transmission.

The transmission gates and "glue logic" shown in Figure 3-7 form a master–slave circuit that has the functionality of a positive-edge-triggered D-type flip-flop with an additional signal, *Clear_bar*, that forces output *Q* to be de-asserted when *Clear_bar* is 0. The master stage is active while *clock* is low, and the slave stage is active while *clock* is high. While *clock* is low, the master stage charges to a value determined by *Data*; when *clock* goes high, the output of the master stage is passed through to the slave stage. The waveforms in Figure 3-7(b) show that *Q* gets the value of *Data* at the rising edges of *clock*.

Figure 3-8 shows the signal paths (a) during the master cycle and (b) during the slave cycle. The output node of the master stage, *w*2, is charged by the input during the master cycle (with *clock* low) and sustained by the feedback loop during the slave

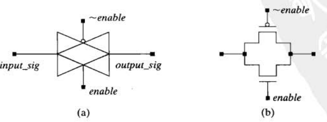


FIGURE 3-6 CMOS transmission gate: (a) circuit symbol and (b) transistor-level schematic.

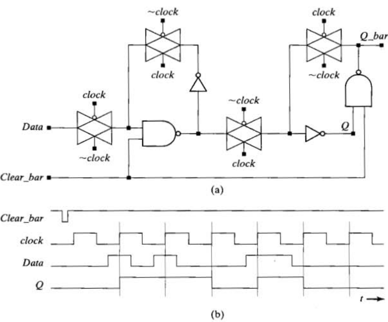


FIGURE 3-7 CMOS master–slave circuit of a D-type flip-flop: (a) circuit schematic and (b) sample waveforms.

cycle, that is, while *clock* is high. The output of the slave stage is sustained by its feedback loop while the master stage is charging. At the active edge of the flip-flop, the output of the master stage is sustained by its feedback loop, and it charges the output of the slave stage during the slave cycle (clock is high).

3.2.3 J-K Flip-Flops

J-K Flip-flops are also edge-sensitive storage elements; data storage is synchronized to an edge of a clock. The value of the data stored is conditional, depending on the data that is present at the J and K inputs when the clock makes a transition at its active edge. The characteristic equation describing the next state of the flip-flop is: $Q_{\text{next}} = JQ' + K'Q$. A J-K flip-flop can be implemented by a D-type flip-flop combined with input logic that forms the data input as $D = JQ' + K'Q$. A block diagram symbol, truth table, and sample waveforms of a J-K flip-flop are shown in Figure 3-9.

3.2.4 T Flip-Flop

The output of a T flip-flop will be complemented at the active edge of the clock if the T (toggle) input is asserted. Otherwise, the output remains unchanged. A T flip-flop can be efficient in implementing a counter. The characteristic equation of a T flip-flop is given by $Q_{\text{next}} = QT' + Q'T = Q \oplus T$. This type of flip-flop can be implemented

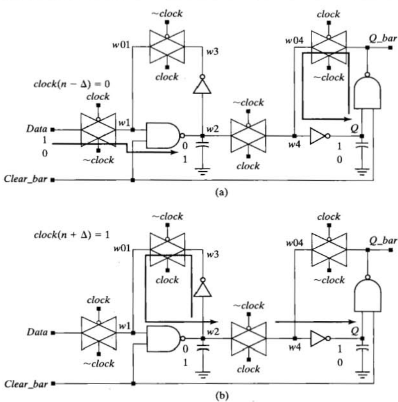
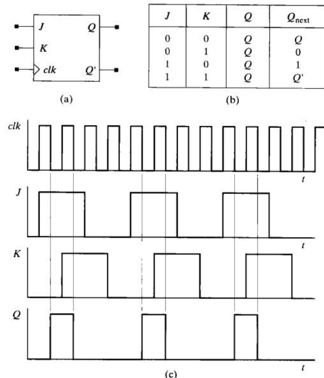


FIGURE 3-8 Signal paths in a CMOS master–slave D-type flip-flop: (a) master cycle signal paths and (b) slave cycle signal paths.

by connecting the *T*-input to the *J* and *K* inputs of a J-K flip-flop. Figure 3-10 shows the schematic symbol, truth table, and sample waveforms for a T flip-flop. Note that the frequency of toggles in *Q* are one-half those of *clk*.

3.3 Busses and Three-State Devices

Busses are multiwire signal paths that connect multiple functional units in a system. They are the highways for information flow. For example, a personal computer has an address bus that carries the source and destinations of data to be retrieved from or stored in memory, and a data bus that carries data being exchanged between functional units, registers, and memory. By sharing the physical resource of a bus, the overall physical resources and



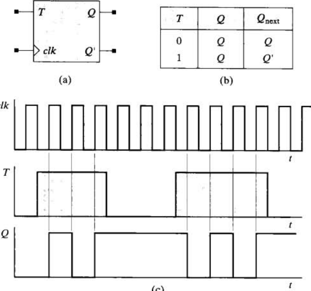
J	K	Q	Q_{next}
0	0	Q	Q
0	1	Q	0
1	0	Q	1
1	1	Q	Q'

FIGURE 3-9 J-K flip-flop: (a) Block diagram symbol, (b) truth table, and (c) sample waveforms.

board space supporting the architecture of a system can be reduced, compared to a circuit with dedicated signal paths. The trade-off is that access to the bus must be managed to avoid conflicts. Bus management makes use of hardware and software.

At the hardware level, three-state devices provide a dynamic interface between a bus and a circuit, acting as signal paths when enabled, but are otherwise an open circuit. Multiple drivers can be connected to a common bus, each with its own set of three-state buffers or inverters that interface to the bus. The output of a three-state device is a function of its data input while the controlling input is asserted. Otherwise, the output is said to be in the high-impedance state or it is disconnected from the circuit. Figure 3-11 shows logic symbols and truth tables for various three-state circuit elements that can buffer or invert an input signal ("Hi-Z" represents a high-impedance state.)

Three-state devices are commonly used to isolate subcircuits from a bus, as shown in Figure 3-12. When *send_data* is high the content of the register is placed on the external bus, *data_to_or_from_bus*; when *rcv_data* is high, data from the external bus is passed into the circuit via *inbound_data*.



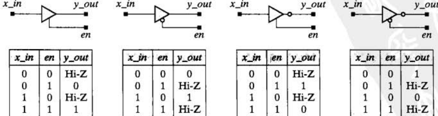
T	Q	Q_{next}
0	Q	Q
1	Q	Q'

FIGURE 3-10 T (toggle) flip-flop: (a) Block diagram, (b) truth table, and (c) sample waveforms.

Busses may operate in a synchronous or an asynchronous manner. At the software level, hand-shaking protocols are used to establish and support coherent transmission of data. Busses include an arbitration scheme that resolves contention issues between multiple requesters for bus service.

Example 3.1

The registers in Figure 3-13 are connected by a 4-bit bidirectional data bus. Each register can send data to any other register. The signal waveforms shown in Figure 3-14



x_in	en	y_out
0	0	Hi-Z
0	1	0
1	0	Hi-Z
1	1	1

x_in	en	y_out
0	0	0
0	1	Hi-Z
1	0	1
1	1	Hi-Z

x_in	en	y_out
0	0	Hi-Z
0	1	1
1	0	Hi-Z
1	1	0

x_in	en	y_out
0	0	1
0	1	Hi-Z
1	0	0
1	1	Hi-Z

FIGURE 3-11 Circuit symbols and truth tables for three-state devices.

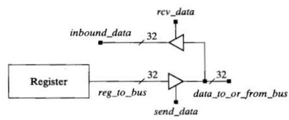


FIGURE 3-12 Bus isolation with three-state devices.

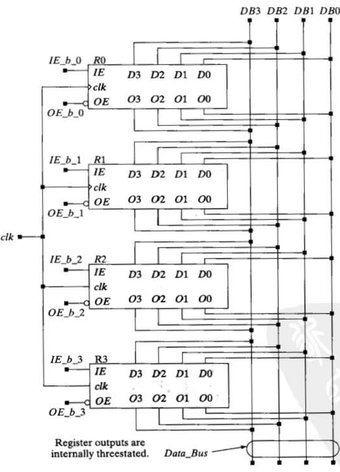


FIGURE 3-13 A register bank with a 4-bit-wide data bus.

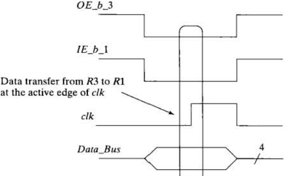


FIGURE 3-14 Bus isolation and data transfer with three-state devices.

would establish datapaths connecting the output of register *R*3 to the inputs of register *R*1 via active-low three-state buffers that are built into the register circuit. To connect the output of *R*3 to the input of *R*1, both *OE_3_b* and *IE_b_1* must be low. The other registers are not affected by the bus activity.

End of Example 3.1

3.4 Design of Sequential Machines

Unlike combinational logic, whose output is an immediate function of only its present inputs, sequential logic depends on the history of its inputs. This dependency is expressed by the concept of "state." The future behavior of a sequential machine is completely characterized by its input and its present state. At any time, the state of a system is the minimal information that, together with the inputs to the system, is sufficient to determine the future behavior of the system. For example, knowing the number of 1s that will appear at the input of a machine that counts 1s in a serial bit stream is not enough information to determine the count at any time in the future. The present count must also be known. Thus, the state of the counter is its present count.

Sequential machines are widely used in applications that require prescribed sequential activity. For example, the outputs of a sequential machine control the synchronous datapath and register operations of a computer. All sequential machines have the general feedback structure shown in Figure 3-15, in which the next state of the machine is formed from the present state and the present input. Combinational logic forms the next state (NS) from the primary inputs and the stored value of the present state (PS). A state register (memory) holds the value of the PS, and the value of the next state is formed from the inputs and the content of the state register. In this structure, the state transitions are asynchronous.

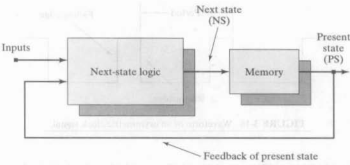


FIGURE 3-15 Block diagram of a sequential machine.

The state transitions of an asynchronous sequential machine are unpredictable. Most application-specific integrated circuits (ASICs) are designed for fast synchronous operation because race conditions are very problematic for asynchronous machines, and they get worse as the physical dimensions of devices and signal paths shrink. Synchronous machines overcome race issues by having a clock period that is sufficient to stabilize the signals in the circuit. In an edge-triggered clocking scheme, the clock isolates a storage register's inputs from its output, thereby allowing feedback without race conditions. In fact, synchronous machines are widely used because timing issues are reduced to (1) ensuring that setup and hold timing constraints[3] are satisfied at flip-flops (for a given system clock), (2) ensuring that clock skew[4] induced by the physical distribution of the clock signal to the storage elements does not compromise the synchronicity of the design, and (3) providing synchronizers at the asynchronous inputs to the system [2].

The state transitions of an edge-triggered flip-flop-based synchronous machine are synchronized by the active edge (i.e., rising or falling) of a common clock. State changes give rise to changes in the outputs of the combinational logic that determines the next state and the outputs of the machine. Clock waveforms may be symmetric or asymmetric. Figure 3-16 illustrates features of an asymmetric clock waveform, that is, the length of the interval in which the clock is low is not equal to the length of the interval in which the clock is high. Register transfers are all made at either the rising or the falling edge of the clock, and input data are synchronized to change between the active edges.

The period of the clock must be long enough to allow all transients activated by a transition of the clock to settle at the outputs of the next-state combinational logic before the next active edge occurs. This establishes a minimum cycle time (period) of the clock of a sequential machine. The inputs to the state register's flip-flops must

[3]Setup constraints require the data to be stable in an interval before the active edge of the clock; hold constraints require the data to be stable in an interval after the active edge.
[4]*Clock skew* refers to the condition that the active edge of the clock does not occur at exactly the same time at every flip-flop.

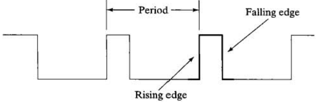


FIGURE 3-16 Waveform of an asymmetric clock signal.

remain stable for a sufficient interval before and after the active edge of the clock. The constraint imposed before the clock establishes an upper bound on the longest path through the circuit, which constrains the latest allowed arrival of data. The constraint imposed after the clock established a lower bound on the shortest path through the combinational logic that is driving the storage device, by constraining the earliest time at which data from the previous cycle could be overwritten. Together, these constraints ensure that valid data are stored. Otherwise, timing violations may occur at the inputs to the flip-flops and cause a condition of metastability, with the result that invalid data are stored.[5]

The set of states of a sequential machine is always finite, and the number of possible states is determined by the number of bits that represent the state. A machine whose state is encoded as an n-bit binary word can have up to 2^n states. We will use the term *finite-state machine* to refer to a clocked sequential machine that has one of the two structures shown in Figure 3-17. Synchronous (i.e., clocked) finite-state machines (FSMs) have widespread application in digital systems, for example, as datapath controllers in computational units and processors. Synchronous FSMs are characterized by a finite number of states and by clock-driven state transitions.

There are two fundamental types of FSMs: Mealy and Moore. The next state and the outputs of a Mealy machine depend on the present state and the inputs; the next state of a Moore machine depends on the present state and the inputs, but the output depends on only the present state. In both machines, the next state and outputs are formed by combinational logic.

3.5 State-Transition Graphs

FSMs can be described and designed systematically with the aid of timing diagrams [2], state tables, state graphs [3], and algorithmic state machine charts ASM charts [1]. Timing diagrams can be used to specify relationships between assertions and transitions of signals in a system and at its interface to its environment. For example, the write cycle of a static random access memory can be specified by a timing chart that indicates when the address of a memory cell must be asserted prior to assertion of a

[5]We will consider metastability in Chapter 5.

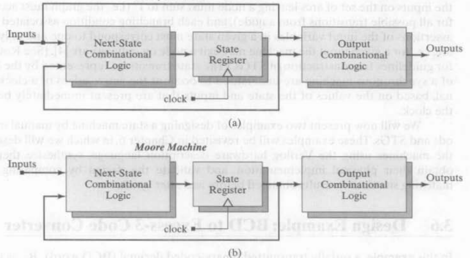


FIGURE 3-17 Block diagram structures of finite state machines: (a) a Mealy machine and (b) a Moore machine.

write-enable signal. In a synthesis-oriented design methodology, timing specifications are incorporated as constraints on the circuit that must be realized by the design tools. Our attention here will focus on state tables, state-transition graphs, and ASM charts. Chapter 11 will consider timing analysis.

State tables, or state-transition tables, display in tabular format the next state and output of a state machine for each combination of present state and input. A state-transition graph (STG), or diagram, of an FSM is a directed graph in which the labeled nodes, or vertices, correspond to the machine's states, and the directed edges, or arcs, represent possible transitions under the application of an indicated input signal when the system is in the state from which the arc originates. The vertices of the STG of a Mealy machine are labeled with the states. The edges of the graph are labeled with (1) the input that causes a transition to the indicated next state and (2) the output that is asserted in the present state for that input. The graph for a Moore-type machine is similar, but its outputs are indicated in each state vertex, instead of on the arcs.

Given an STG for a synchronous machine, the design task is to determine a circuit that implements the next-state and output logic. If the state of the machine is represented by a binary word, its value can be stored in flip-flops. At each active edge of the clock, the inputs to the state-holding flip-flops become the state for the next cycle of the clock. The design of the machine specifies the logic that forms the inputs to the flip-flops from the state and the external inputs to the machine. This logic will be combinational, and it should be minimized, if possible. To be a valid STG, each of its vertices must represent a unique state, each arc must represent a transition from a given state to a next state under the action of the indicated input, and each arc leaving a node

must correspond to a unique input. In general, the Boolean conditions associated with the inputs on the set of arcs leaving a node must sum to 1[6] (i.e., the graph must account for all possible transitions from a node), and each branching condition associated with assertions of the input variables in a given state must correspond to one and only one arc, i.e., for a given input the machine may exit a node on only one arc [4]. (See Roth [3] for guidelines for construction of STGs.) The state transitions represented by the STG of a synchronous machine are understood to occur at the active edges of a clock signal, based on the values of the state and inputs that are present immediately before the clock.

We will now present two examples of designing a state machine by manual methods and STGs. These examples will be revisited in Chapter 6, in which we will describe the machines using the Verilog hardware description language, synthesize them to obtain their physical implementation, and validate the design by comparing and matching simulation results obtained before and after synthesis.[7]

3.6 Design Example: BCD to Excess-3 Code Converter

In this example, a serially transmitted binary-coded decimal (BCD word), B_{in}, is to be converted into an Excess-3 encoded serial bit stream, B_{out}. An Excess-3 code word is obtained by adding 3_{10} to the decimal value of the BCD word and taking the binary equivalent of the result. Table 3-1 shows the decimal digits, their 4-bit BCD code words, and their Excess-3 encoded counterparts. An Excess-3 code is self-complementing [2,4,7], that is, the 9s complement[8] of an Excess-3 encoded word is obtained in hardware by complementing the bits of the word (i.e., by taking the 1s complement of the word). For example, the Excess-3 code for 6_{10} is 1001_2; its bitwise complement is 0110_2, which is the Excess-3 code for 3_{10}. This feature of the Excess-3 code makes it possible to easily implement a diminished radix[9] complement scheme for subtracting numbers that are encoded in a BCD form. This is similar to subtraction of signed binary words by adding the 2s complement of the minuend to the subtrahend. The 2s complement is formed by adding 1 to the 1s (diminished radix) complement of the minuend. Thus, the 10s complement of 6_{10} can be obtained by bitwise complementing 1001_2, the Excess-3 code of 6_{10}, and adding 1 to the result: $0110_2 + 0001_2 = 0111_2$, which decodes to 7_{10}.

A BCD to Excess-3 code converter for a serial bit stream can be implemented as a Mealy FSM. Figure 3-18 shows a serial bit stream, B_{in}, entering the converter and the corresponding serial stream of Excess-3 encoded bits, B_{out}, leaving the machine. Note that the bits of B_{in} are transmitted in sequence, with the least significant bit (LSB) first. Consequently, care must be taken to interpret the waveforms of B_{in} and B_{out} correctly.

[6]The chart can be simplified by showing only the transitions that leave a state by omitting arcs that begin and end at the same state and by omitting return arcs that are activated by a reset signal.

[7]See sections 6.6.1 to 6.6.3.

[8]The 9s complement of a binary number a is the binary value a' such that $a + a' = 9$.

[9]The radix 9 is the diminished radix for a base 10 (decimal) system.

TABLE 3-1 BCD and Excess-3 code words.

Decimal Digit	8-4-2-1 Code (BCD)	Excess-3 Code
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

The order of the bits in the waveforms is shown progressing from right to left (with increasing t), with the LSB at the left and the most significant bit (MSB) at the right. The pattern of bits in the waveform must be reversed, as shown in Figure 3-18, to form the binary values of the transmitted and received words.

The STG[10] of a serial code converter that implements the code in Table 3-1 is shown in Figure 3-19(a), with an asynchronous reset signal that transfers the machine from any state to state *S_0* whenever it is asserted, independently of the clock. The machine's action commences in *S_0* with the first clock edge after reset and continues indefinitely, repeating the addition of 0011_2 to successive 4-bit slices of the input stream. The LSB of the word is the first bit in the sequence of input samples, and the first bit generated for the output word. The state table in Figure 3-19(b) summarizes the same information as the state transition graph, but in tabular format. The notation "-/-" indicates an unspecified or impossible condition.

Systematic design of a D flip-flop realization of an FSM consists of the following steps: (1) construct an STG for the machine, (2) eliminate equivalent states, (3) select a state code (e.g., a binary code), (4) encode the state table, (5) develop Boolean equations describing the inputs of the D flip-flops that hold the state bits, and (6) using K-maps, optimize the Boolean equations. In general, the number of flip-flops used to represent the state of the machine must be sufficient to accommodate a binary representation of the number of states—that is, a machine that has 12 states requires at least four flip-flops.

[10]The STG of a completely specified machine with n inputs must have 2^n arcs leaving each node, and the number of its states must be a power of 2. Otherwise, some bit patterns will be unused in the hardware implementation.

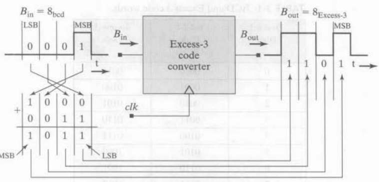
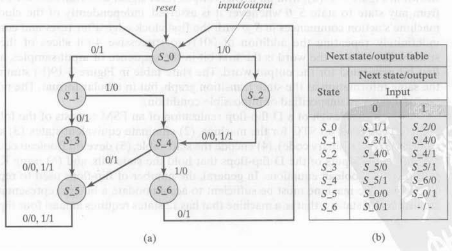


FIGURE 3-18 Input–output bit streams in a BCD to Excess-3 serial code converter.

For a given set of flop-flops, it is then necessary to assign a unique binary code to each state. This problem is difficult because the number of possible codes grows exponentially with the number of available flip-flops. The choice ultimately matters, because it can have an impact on the complexity of the logic required to implement the machine. We will consider this topic in more detail in Chapter 6. The codes for state assignment in our example are shown in Figure 3-20, where a simple (sequential) 3-bit binary code has been used to encode the seven states of the machine. The encoded next state and output table are also shown.



Next state/output table		
	Next state/output	
State	Input	
	0	1
S_0	S_1/1	S_2/0
S_1	S_3/1	S_4/0
S_2	S_4/0	S_4/1
S_3	S_5/0	S_5/1
S_4	S_5/1	S_6/0
S_5	S_0/0	S_0/1
S_6	S_0/1	-/-

FIGURE 3-19 BCD to Excess-3 serial code converter implemented as a Mealy-type FSM: (a) state transition graph and (b) the machine's state table.

State assigment	
$q_2 q_1 q_0$	State
000	S_0
001	S_1
010	S_6
011	S_4
100	
101	S_2
110	S_5
111	S_3

(a)

Encoded next state/output table						
	State	Next state		Output		
	$q_2 q_1 q_0$	$q_2^+ q_1^+ q_0^+$				
		Input		Input		
		0	1	0	1	
S_0	000	001	101	1	0	
S_1	001	111	011	1	0	
S_2	101	011	011	0	1	
S_3	111	110	110	0	1	
S_4	011	110	010	1	0	
S_5	110	000	000	0	1	
S_6	010	000	–	1	–	
	100	–	–	–	–	

(b)

FIGURE 3-20 BCD to Excess-3 code converter implemented as a Mealy-type FSM: (a) state assignment and (b) encoded next state and output table.

Our next step will be to develop Karnaugh maps for each bit of the encoded state and output, as functions of the present state bits and the input (B_{in}). These maps are shown in Figure 3-21 with their corresponding Boolean equations. The unspecified entries in the table are treated as don't-care conditions. Each equation has been minimized individually, although this does not necessarily produce the optimal realization (speed vs. area) of the logic. We will consider optimization of a set of Boolean equations in our discussion of logic synthesis in Chapter 6.

The Boolean equations for q_2^+ and B_{out} can be converted into the following NAND gate structure, where, for clarity, we use the symbol "·" to indicate the Boolean AND operator:

$$q_2^+ = q_1'q_0'B_{in} + q_2'q_0B_{in}' + q_2q_1q_0$$

$$\overline{q_2^+} = \overline{q_1'q_0'B_{in} + q_2'q_0B_{in}' + q_2q_1q_0}$$

$$\overline{q_2^+} = \overline{q_1'q_0'B_{in}} \cdot \overline{q_2'q_0B_{in}'} \cdot \overline{q_2q_1q_0}$$

$$q_2^+ = \overline{\overline{q_1'q_0'B_{in}} \cdot \overline{q_2'q_0B_{in}'} \cdot \overline{q_2q_1q_0}}$$

and

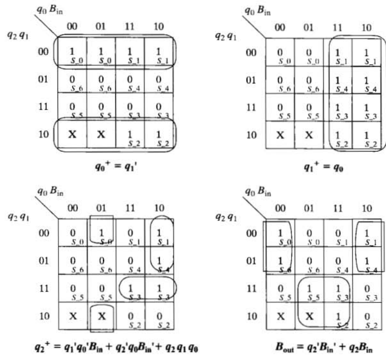


FIGURE 3-21 Karnaugh maps for the encoded state bits and output bit (B_{out}) of a BCD to Excess-3 code converter implemented as a Mealy-type FSM with input bit (B_{in}).

$$
\begin{aligned}
B_{out} &= q_2'B_{in}' + q_2B_{in} \\
B_{out}' &= \overline{q_2'B_{in}' + q_2B_{in}} \\
B_{out}' &= (\overline{q_2'B_{in}'}) \cdot (\overline{q_2B_{in}}) \\
B_{out} &= \overline{(\overline{q_2'B_{in}'}) \cdot (\overline{q_2B_{in}})}
\end{aligned}
$$

The schematic of the code converter is shown in Figure 3-22, with three positive edge-triggered flip-flops storing the state bits. The simulation results in Figure 3-23 illustrate the input–output waveforms and the state transitions of the machine. The annotation of the displayed waveforms shows the bit stream of the encoded word produced by the converter for $B_{in} = 0100_2$, where the LSB is asserted first, and the MSB is last in the time sequence. Since the output of a Mealy machine depends on the input as well as the state, the transitions of B_{in} affect the waveform of B_{out}. We have aligned the transitions of B_{in} to occur on the inactive edge of the clock. This is a recommended practice, which ensures that the data are stable before the active edge of the clock. Since the input of a Mealy machine can cause the output of the machine to change its value, the valid output of a Mealy machine is taken to be the value of the output immediately before the active

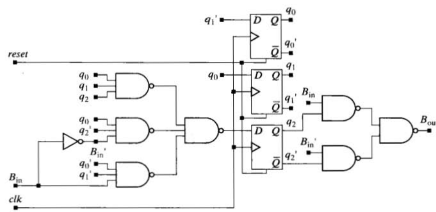


FIGURE 3-22 Circuit for a Mealy-type FSM implementing a BCD to Excess-3 code converter.

edge of the clock. The value of B_{out} immediately before an active edge of the clock depends on the value of B_{in} immediately before the clock, and it is the valid output of this machine.[11] Thus, transmitting the input bit stream 0100_2 (LSB first) generates the output bit stream 0111_2. The waveforms of B_{in} and B_{out} in Figure 3-23 are annotated with bubbles to mark corresponding values of the BCD and Excess-3 encoded bits.

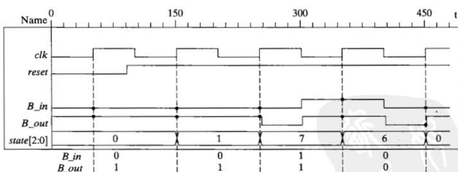


FIGURE 3-23 Simulation results for a BCD to Excess-3 code converter implemented as a Mealy-type FSM, with annotation marking corresponding input and output values.[12]

[11] A physical circuit implementing the machine would have to perform the machine's addition fast enough to have the result ready for the active edge of the clock. Performance is an issue for synthesis and will be addressed in Chapter 10.

[12] Note: *B_in* and *B_out* in the simulator output represent B_{in} and B_{out}, respectively.

3.7 Serial-Line Code Converter for Data Transmission

Line codes are used in data transmission or storage systems to reduce the effects of noise in serial communications channels and/or to reduce the width of channel datapaths [2]. For example, in data transmission in which bits of a word are encoded and transmitted synchronously over a channel, the receiver of the data must be able to operate synchronously with the sending unit, identify the boundaries between words (frames), and distinguish the transmitted bits from each other. One scheme for data recovery after transmission requires three signals: a clock to define the boundaries of the data bits, a synchronizing signal to define word boundaries, and a data stream. Other implementations using fewer signal channels are possible. For example, a phone system or a disk read/write head will have a single channel for the data and use a coding scheme to enable clock recovery and synchronization. Code converters transform the data stream into a format that has been encoded to enable the receiver to recover the data. A phase lock loop (PLL [3]) can recover the clock from the line data (i.e., synchronize itself to the clock of the data) if there are no long series of 1s or 0s in a data stream with a non–return-to-zero (NRZ) format; the clock can be recovered from a data stream with a non–return-to-zero invert-on-ones (NRZI) format or return-to-zero (RZ) code format if the data has no long string of 0s. Manchester encoders are attractive because they can recover the clock independently of the pattern of the data, but they require higher bandwidth.

Figure 3-24 shows bit data and encoded bit patterns for four common encoding/transmission schemes. The patterns show only the relationship between the bits of the data and the bits of the encoded signal. The figure does not show latency between the actual input and output bit times, but the actual phase (timing) relationships might differ from those shown.

- **NRZ Code**: The signal waveform of the line value formed by an NRZ code generator duplicates the bit pattern of the input signal, as shown in Figure 3-24. The output waveform makes no transition between two identical successive bits. A Moore machine implementing an NRZ code samples the data at the active edge of *clock_1* and changes state accordingly (in the middle of the bit time); the transitions of the data are synchronized to the inactive edge of *clock_1*.
- **NRZI Code**: If the input to an NRZI code converter in the present bit time is 0, the sequential output of the converter remains at it previous value (i.e., the output in the previous bit time). If the input is a 1, the output in the present bit time is the complement of the output in the previous bit time. So the output remains constant as long as the input remains at 0, and the output toggles if the input is held at 1, as shown in Figure 3-24. The asserted value is held for the entire bit time.

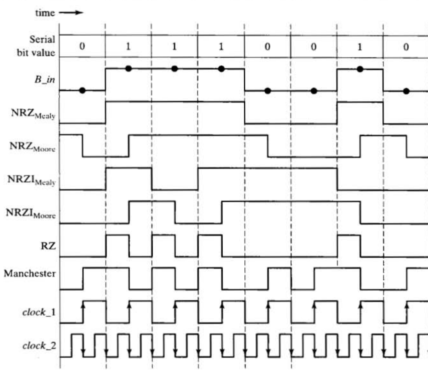


FIGURE 3-24 Serial line code formats and sample waveforms.

- **RZ Code**: A 0 in the input bit stream of RZ code generator is transmitted as a 0 for the entire bit time. A 1 in the bit stream is transmitted as a 1 for the first half of the bit time, and 0 for the remaining bit time (typically).
- **Manchester Code:** A 0 in the bit stream of a Manchester code generator is transmitted as a 0 for the lead half of the bit time and as a 1 for the remaining half. A 1 in the bit stream is transmitted as a 1 for the leading half of the bit time and as a 0 for the remaining bit time.

Note that the clock frequency (*clock_2*) of the Mealy-type machines implementing RZ and Manchester encoders must operate at twice the clock frequency of the bit stream generator (*clock_1*) in order to assert the line value of a full bit time without latency.

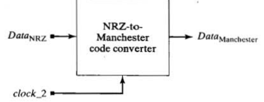


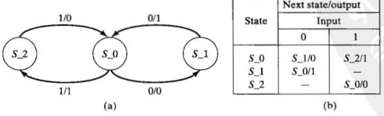
FIGURE 3-25 Input–output datapaths for a NRZ Manchester code converter.

3.7.1 Design Example: A Mealy-Type FSM for Serial Line-Code Conversion

A serial line-code converter can be implemented by an FSM in which the inbound bit stream controls the actions of the machine to produce an encoded outbound bit stream. As an example, a Mealy-type FSM will be designed to convert a data stream in NRZ format, Data_{NRZ}, to a data stream in Manchester code format, $\text{Data}_{\text{Manchester}}$, as represented by the block diagram in Figure 3-25.

The STG and state table for the machine are shown in Figure 3-26, and the state assignment and encoded state table are shown in Figure 3-27. The table shows the state label and code for each state, the codes corresponding to the bits of the next state for each possible value of B_{in}, and the output that will be asserted in the indicated state for each possible value of B_{in}. An asserted bit in the input will be asserted for two clocks of the output. Thus, the arcs leaving *S_1* with an input of 1 and from *S_2* with an input of 0 are not shown because those inputs sequences cannot occur. The corresponding entries in the next-state table are marked as don't-cares.

The Karnaugh maps and Boolean equations for the line converter are derived from the STG, and are shown in Figure 3-28. The circuit schematic shown in Figure 3-29 implements the Boolean equations of the machine and stores its state in two negative edge-triggered D-type flip-flops. The simulation results in Figure 3-30 show the input (B_{in}) and output (B_{out}) waveforms for a sample of data and also show the state transitions of the machine that accomplishes the code conversion. Note that because the machine's output is generated as a Mealy-type output, there is no latency between



State	Next state/output	
	Input	
	0	1
S_0	S_1/0	S_2/1
S_1	S_0/1	—
S_2	—	S_0/0

(b)

FIGURE 3-26 Mealy-type NRZ-to-Manchester encoder: (a) the state transition graph and (b) the next-state table.

	q_0	
q_1	0	1
0	S_0	S_1
1	S_2	

(a)

State		Next state		Output	
$q_1\ q_0$		$q_1^+\ q_0^+$			
		Input		Input	
		0	1	0	1
S_0	00	01	10	0	1
S_1	01	00	00	1	—
S_2	10	00	00	—	0

(b)

FIGURE 3-27 Mealy-type NRZ-to-Manchester encoder: (a) the state assignment table and (b) the encoded next-state/output table.

the waveform of the inbound bit stream and that of the output; that is the bit times of the input and the output coincide.

3.7.2 Design Example: A Moore-Type FSM for Serial Line-Code Conversion

In general, the output of a Mealy machine might have glitches when the input bit stream changes. If this cannot be tolerated, a Moore-type machine should be used. A simplified STG (without impossible arcs) and a state table for a Moore machine NRZ-to-Manchester encoder are shown in Figure 3-31, and the state assignment and encoded state table are shown in Figure 3-32. Note that the data transitions are synchronized by the negative-edge transitions of *clock_1*, and the state transitions of the encoder and the sampling of *B_in* are synchronized by the negative-edge transitions of *clock_2*.

The Karnaugh maps and Boolean equations for the converter are derived from the STG, and are shown in Figure 3-33, with don't-care conditions denoted by "–" (for impossible patterns). The circuit schematic shown in Figure 3-34 implements the

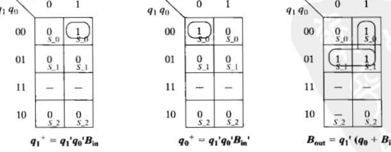


FIGURE 3-28 The Karnaugh maps and Boolean equations for the encoded state bits and output bit (B_{out}) of an NRZ-to-Manchester encoder with input bit B_{in}.

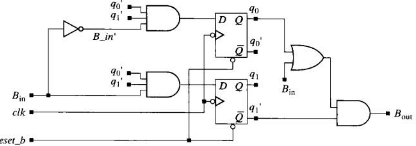


FIGURE 3-29 The circuit schematic of a Mealy-type NRZ-to-Manchester encoder.

Boolean equations of the machine with negative-edge triggered flip-flops. The simulation results in Figure 3-35 show the input (B_{in}) and output (B_{out}) waveforms for a sample of data and also show the state transitions of the state machine that accomplishes the code conversion. Note that the Manchester encoder must run at twice the frequency of the incoming data and that the bit stream of the output of the Moore machine lags the input bit stream by one-half the cycle time input clock. The transitions of the input data stream are made at the falling edges of *clock_1*, and the state

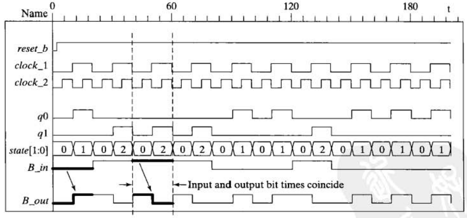
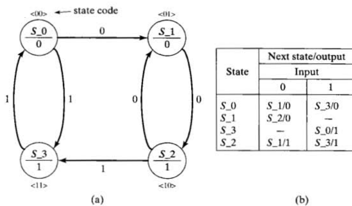


FIGURE 3-30 Simulation results for the Mealy-type NRZ-to-Manchester encoder.[13]

[13]In general, it is not advisable to switch an input at the same time that a sequential machine makes a state transition, but in this circuit the samples that are made when the input is making a transition occur in states (i.e., *S_1* and *S_2*) in which the input is treated as a don't-care. So the input will not affect the transition from *S_1* or *S_2* to *S_0*. The scheme here effectively samples B_{in} in the middle of its bit time, and eliminates a cycle of latency in the output of the converter.



State	Next state/output	
	Input	
	0	1
S_0	S_1/0	S_3/0
S_1	S_2/0	–
S_3	–	S_0/1
S_2	S_1/1	S_3/1

FIGURE 3-31 Moore machine NRZ-to-Manchester encoder: (a) the state-transition graph and (b) the next-state/output table.

machine samples the input at the rising edges of *clock_1*—that is, in the middle of the bit time of the input, which corresponds to every other falling edge of *clock_2*. The transitions of the state of the machine coincide with the falling edge of *clock_2*. The latency between the output and the input is a consequence of the fact that the outputs of a Moore machine are dependent on only the state. Thus, a change in the input must first cause a transition to a state before the encoder can assert an output that corresponds to the sampled/detected input.

3.8 State Reduction and Equivalent States

Two states of a sequential machine are equivalent ($\equiv$) if they have the same output sequence for all possible input sequences. Such states of the machine cannot be distinguished from each other on the basis of observed outputs. Equivalent states can be

	q_0	
q_1	0	1
0	S_0	S_1
1	S_2	S_3

(a)

State		Next State		Output
$q_1\ q_0$		$q_1^+\ q_0^+$		
		Input		
		0	1	
S_0	00	01	11	0
S_1	01	10	–	0
S_3	11	–	00	1
S_2	10	01	11	1

(b)

FIGURE 3-32 Moore-type NRZ-to-Manchester encoder: (a) the state-assignment table and (b) encoded next-state/output table.

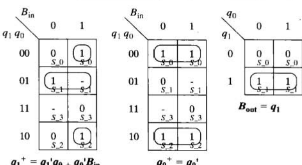


FIGURE 3-33 The Karnaugh maps and Boolean equations for the encoded state bits and output bit (B_{out}) for a NRZ-to-Manchester encoder implemented as a Moore machine with input bit B_{in}.

combined without changing the input–output behavior of the machine. Identifying and combining equivalent states usually simplifies the state table and the STG of the machine and leads to a reduction in hardware (because the equivalent states do not have to be decoded) without compromising functionality [8]. In general, for every FSM, there is a unique equivalent machine that is minimal.

Example 3.4

The state machine whose next-state table is shown in Figure 3-36 has two states that are equivalent: *S_4* = *S_5*. Both *S_4* and *S_5* have the same next states and outputs under the actions of the inputs. If the machine is in *S_4* and an input sequence is applied, the

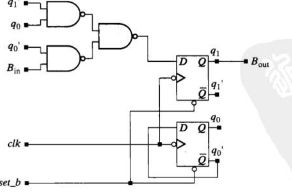


FIGURE 3-34 The circuit schematic of a Moore machine NRZ-to-Manchester encoder.

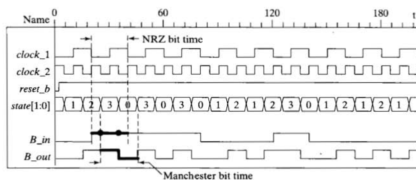


FIGURE 3-35 Simulation results for the Moore machine NRZ-to-Manchester encoder.

outputs will be the same as if the machine was in *S_5* and the same input sequence is applied. The machine's STG, given in Figure 3-37(a), shows how *S_4* and *S_5* map into the same next state, and that they also have the same outputs for all assertions of the input.

Two states of a sequential machine are equivalent if their associated rows in the machine's state table are identical. The STG of a machine can be pruned by removing all but one of the states that are equivalent and redirecting the affected arcs to the remaining equivalent state. Likewise, the next-state table can be simplified by eliminating one of

	Next state		Output	
	Input		Input	
State	0	1	0	1
S_0	S_6	S_3	0	0
S_1	S_1	S_6	0	1
S_2	S_2	S_4/S_5	0	1
S_3	S_7	S_3	0	1
S_4	S_7	S_2	0	0
S_5	S_7	S_2	0	0
S_6	S_0	S_1	0	0
S_7	S_4	S_3	0	0

Equivalent states — Replace label — Delete

FIGURE 3-36 Next-state and output table with equivalent states *S_4* and *S_5*. To simplify the table, replace the label *S_5* by the label *S_4*, and delete the row for *S_5*.

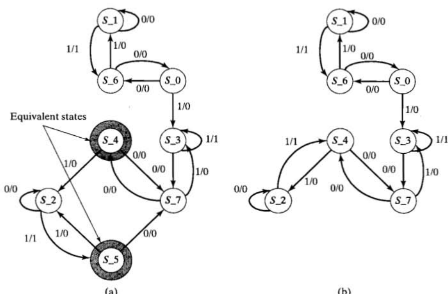


FIGURE 3-37 State-transition graph with equivalent states: (a) graph with *S_4* and *S_5* and (b) state-transition graph after removal of *S_5* and redirection of arcs.

the rows and replacing the labels of the deleted state by the label of its equivalent state. But be careful—do not conclude that two states are not equivalent if their state-table rows are different. The condition of matching rows of the next-state table is a sufficient condition for the associated states to be equivalent, but is not a *necessary* condition to ensure their equivalence. Merely comparing the rows of a state table is not a foolproof way to identify equivalent states. Other states that are equivalent might not be detected by such an approach.

A more general procedure for eliminating equivalent states relies on a recursive definition of equivalence: Two states are equivalent if they have the same output for each input value, and the states they reach for the same input value are equivalent. The procedure (1) forms a triangular array (see Figure 3-38) showing the possible pairwise combinations of distinct states, and (2) considers the conditions for the pair of states to be equivalent (we already know that *S_4* and *S_5* of the original table are equivalent, so we do not consider *S_5*). For example, *S_0* and *S_4* can be equivalent only if the next states they reach are equivalent and if they have the same output under the action of each possible input value. The table entries in Figure 3-36 indicate that *S_0* and *S_4* have the same outputs, but are equivalent only if *S_6* and *S_7* are equivalent, and if *S_2* and *S_3* are equivalent. Both conditions must be satisfied—we do not know in advance whether *S_2* and *S_3* are equivalent and whether *S_6* and *S_7* are equivalent.

S_1						
S_2		S_6 - S_4				
S_3		S_1 - S_7 S_6 - S_3	S_2 - S_7 S_4 - S_3			
S_4	S_6 - S_7 S_3 - S_2					
S_6	S_3 - S_1				S_7 - S_0 S_2 - S_1	
S_7	S_6 - S_4				S_2 - S_3	S_0 - S_4 S_1 - S_3
	S_0	S_1	S_2	S_3	S_4	S_6

FIGURE 3-38 Triangular array showing possible pairwise combinations of states.

Within a given cell corresponding to a row and column of the table, list the pairs of states that are reached from the states whose labels correspond to the row and column. For example, if the machine is in *S_1*, its next state is *S_1* or *S_6* if the input is 0 or 1, respectively. Likewise, if the machine is in *S_3*, it reaches *S_7* and *S_3* under the same applied input values. Thus, for *S_1* and *S_3* to be equivalent, it is necessary and sufficient that *S_1* and *S_7* be equivalent states, and that *S_6* and *S_3* be equivalent states, provided of course, that the outputs are the same. The cells of Figure 3-38 are annotated with the pairs of states that must be equivalent in order for the states corresponding to the row and column to be *equivalent*. If a pair of states has a different output for some input, the pair is eliminated and the cell is shaded to signify that the states cannot be equivalent. For example, *S_1* and *S_4* cannot be equivalent because their output assertions do not match. Next, any cells containing the labels of a state pair corresponding to a shaded state pair are lined out, as shown in Figure 3-39. For example, the cell containing

S_1						
S_2		S_6 - S_4				
S_3		~~S_1 - S_7 S_6 - S_3~~	~~S_2 - S_7 S_4 - S_3~~			
S_4	S_6 - S_7 S_3 - S_2					
S_6	S_3 - S_1				S_7 - S_0 S_2 - S_1	
S_7	S_6 - S_4				S_2 - S_3	S_0 - S_4 S_1 - S_3
	S_0	S_1	S_2	S_3	S_4	S_6

FIGURE 3.39 Triangular array showing possible pairwise combinations of states.

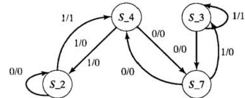


FIGURE 3-40 Completely pruned state-transition graph.

the pairs (*S_1, S_7* and *S_3, S_6*) is lined out because *S_1* and *S_7* cannot be equivalent. When this process is completed, the remaining labeled cells identify equivalent states. The results in Figure 3-39 indicate that $S_4 = S_5$, $S_0 = S_7$, $S_2 = S_2$, and $S_4 = S_6$. This leads to the simplified STG shown in Figure 3-40, having only four states, instead of eight.

End of Example 3.4

REFERENCES

1. Katz RH. *Contemporary Logic Design*. Redwood City, CA: Benjamin Cummings, 1994.
2. Wakerly JF. *Digital Design Principles and Practices*, 4th ed. Upper Saddle River, NJ: Prentice-Hall, 2006.
3. Mano M, Ciletti M. Digital Design, 4th ed. Upper Saddle River, NJ: Prentice-Hall, 2007.
4. Tinder RF. *Engineering Digital Design*, 2nd ed. San Diego, CA: Academic Press, 2000.
5. Weste N, Eshraghian K., Smith, JM. *Principles of CMOS VLSI Design—A Systems Perspective with Verilog/VHDL Manual*. Reading, MA: Addison-Wesley, 2000.
6. Smith MJS. *Application-Specific Integrated Circuits*. Reading, MA: Addison-Wesley, 2008.
7. Breeding KJ. *Digital Design Fundamentals*, 2nd ed. Upper Saddle River, NJ: Prentice-Hall, 1997.
8. Hachtel GD, Somenzi F. *Logic Synthesis and Verification Algorithms*. Boston, MA: Kluwer, 1996.

PROBLEMS

1. Using D-type flip-flops, design a synchronous Moore finite-state machine that monitors two inputs, *A* and *B*, and asserts a scalar output if the number of 1s observed on the inputs is a multiple of 4.
2. Using D-type flip-flops, design a Moore machine whose output indicates the even parity of its serial bit stream of inputs since reset.

3. Using D-type flip-flops, design a 3-bit counter that will count in the cyclic pattern of even numbers: 0_{10} 2_{10} 4_{10} 6_{10}.
4. Draw the state-transition graph of a Mealy machine that samples a serial bit stream and asserts an output if the last three samples are 1. Assume that the samples are made at the inactive edges of the clock. Design the machine with D-type flip-flops.
5. Draw the state-transition graph of a Moore machine that samples a serial bit stream and asserts an output if the last three samples are 1. Assume that the samples are made at the inactive edges of the clock. Design the machine with D-type flip-flops.
6. Examine the Mealy machine described by the K-maps in Figure 3-28 and identify possible glitches that could occur when the input makes a transition.
7. Draw the state-transition graph of a Moore-type state machine that receives a serial bit stream and generates a serial output in which the output (1) is asserted for one clock cycle if the input pattern 0111 is recognized, (2) remains de-asserted until the pattern 0111 is recognized again, and (3) re-asserts on detecting the second occurrence of 0111_2, and so forth, continuously. (a) Assume that the LSB (right-most bit of the binary value) of the indicated pattern (0111_2) arrives first in the data stream. (b) Alternatively, assume that the pattern indicates the left-to-right time sequence of arriving bits, and that 0 is the LSB and arrives first.
8. Draw the state-transition graph of a machine that behaves like the machine in Problem 7, but terminates its activity after it has detected six occurrences of 0111, until a reset condition is asserted.
9. Draw the state-transition graph of a Moore machine that converts a NRZ bit stream into a NRZI bit stream.
10. Draw the state-transition graph (STG) of an NRZI line encoder implemented as (a) a Mealy machine and (b) a Moore machine. Using the STGs, draw output waveforms for the waveform of *B_in* shown in Fig. 3-24. Identify and compare latency and valid bit times of the output waveforms of the Mealy and Moore machines. *Caution*: Avoid the trap of attempting to read the data bits on the clock edge that synchronizes their transitions.
11. Repeat Problem 7 using an ASM chart instead of an STG.
12. Repeat Problem 8 using an ASM chart instead of an STG.
13. Draw the STG of a Moore machine that implements a BCD to Excess-3 code converter.
14. In the STG shown below, (a) determine whether states *s_0* and *s_2* are equivalent, and (b) determine whether states *s_1* and *s_3* are equivalent.

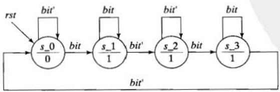


FIGURE P3-14

15. Find the equivalent states of the STG shown below, and draw the STG of the reduced machine.

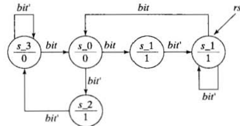


FIGURE P3-15

PDG

CHAPTER 4

Introduction to Logic Design with Verilog

Designers of application-specific integrated circuits (ASICs) use hardware description languages (HDLs) at key steps in the design flow of a circuit. An HDL model is developed; synthesized into a physical circuit; and verified for functionality, timing, and fault coverage. HDLs have some similarity to ordinary, general-purpose languages like C, but they have additional features for modeling and simulating the functionality of combinational and sequential circuits. Two languages are in widespread use: Verilog and VHDL. Both are Institute of Electrical and Electronic Engineers (IEEE) standards, and both have enthusiastic users. Rather than compare, contrast, and even argue the relative merits of the languages, our focus will be on digital design with the Verilog HDL.

Designers using an HDL (1) write a text-based description (model) of a circuit, (2) compile the description to verify its syntax, and (3) simulate the model to verify the functionality of the design. Verification by simulation requires that designers write a testbench containing descriptions of stimulus waveforms that are applied to exercise the functionality of the circuit. Simulators display waveforms by exercising the circuit's behavior, and some can analyze the waveforms to detect functional errors.

Verilog gives designers several alternative ways to describe a circuit. Familiar tools such as schematics, truth tables, and Boolean equations have simple Verilog counterparts. Some designs can be entered easily as a structural description in which logic gates are connected to other gates, just as they would be on a schematic. Truth tables and Boolean equations are also commonly used, but other styles of design can be more abstract, such as an algorithm that describes the impulse response of a lowpass digital filter. We will see that Verilog accommodates abstract models of behavior too.

The Verilog HDL serves as a vehicle for designing, verifying, and synthesizing a circuit [1]. It is also a medium for exchanging designs between designers. Intellectual property encapsulated as a Verilog description can be exported and embedded in other designs.

Today's designs are so large that they must be designed with a top-down methodology that systematically partitions a complex circuit into smaller functional units that can be designed and verified individually and more easily, before reintegrating them and re-verifying the aggregate. Verilog lets designers create a design hierarchy of functional units. In today's global design environment, the partitioned design can be distributed across multiple teams located in the same lab or anywhere on the World Wide Web.

Chapter 2 summarized three common descriptions of combinational logic: schematic/gates, truth tables, and Boolean equations. Verilog includes constructs for each of these, and abstract models too. This chapter will present Verilog constructs corresponding to gate-level and truth-table descriptions of combinational logic.

4.1 Structural Models of Combinational Logic

A Verilog model of a circuit encapsulates a description of its functionality as a structural or behavioral view of its input–output (I/O) relationship. A structural view could be a netlist of gates or a high-level architectural partition of the circuit into major functional blocks, such as an arithmetic and logic unit (ALU). A behavioral view could be a simple Boolean equation model, a register transfer level (RTL) model, or an algorithm. We will begin our introduction to HDLs by considering how Verilog supports structural design. We will introduce basic concepts here before moving to the abstract models.

Structural design is similar to creating a schematic. Figure 4-1 shows a gate-level schematic of a half adder circuit, along with its Verilog description. A schematic consists of icons (symbols) of logic gates, lines representing wires that connect gates, and labels of relevant signal names at I/O pins and internal nodes. Similarly, an HDL *structural model* consists of a list of *declarations*, or statements, that specify the inputs and outputs of the unit and list any gate primitives (e.g., **xor**, **and**) that are interconnected to implement the desired functionality. The primitives that are listed in a declaration within a module are said to be *instantiated* in the design.

4.1.1 Verilog Primitives and Design Encapsulation

Verilog includes a set of 26 predefined functional models of common combinational logic gates, called primitives. *Primitives* are the most basic functional objects that can be used to compose a design. Their functionality is built into the language by means of

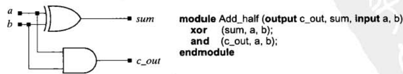


FIGURE 4-1 Schematic and Verilog description of a half adder.

TABLE 4-1 Verilog primitives for modeling combinational logic gates.

n-Input	*n*-Output, 3-state
and	buf
nand	not
or	bufif0
nor	bufif1
xor	notif0
xnor	notif1

internal truth tables that define the relationship between the scalar (i.e., a single bit) output(s) of each primitive and its scalar input(s). Table 4-1 shows an abbreviated list of the predefined primitives and their reserved keywords.[1] Their names suggest their functionality (the ***not*** primitive corresponds to an inverter). The primitives in Table 4-1 are called *n*-input primitives because the same model keyword (e.g., ***nand***) automatically accommodates any number of scalar inputs, rather than only a pair of inputs. The *n*-output primitives (e.g., the buffer primitive, ***buf***) have a single input but can have multiple scalar outputs (to model a gate that has fanout to more than one location). The primitives ***bufif0*** and ***bufif1*** are three-state buffers; ***notif0*** and ***notif1*** are three-state inverters.[2] There are no pre-defined sequential primitives in Verilog.

MODELING TIP

The output port of a primitive must be first in the list of ports. The instance name of a primitive is optional.

Each primitive has ports (corresponding to hardware pins/terminals) that connect to its environment. Figure 4-2 shows a 3-input ***nand*** primitive and a Verilog statement that would imply the use of this primitive in a circuit in which it is connected to input signals *a, b*, and *c* and has output signal *y*. The list of ports is placed immediately to the right of the primitive name as a comma-separated list enclosed by parentheses and terminated by a semicolon (;). The output port(s) of a primitive must be first in the list of ports, followed by the primitive's input port(s). A primitive is instantiated within a module by a statement declaring its keyword name, followed by an optional instance name and by a parentheses-enclosed list of its terminals.[3] All identifiers (names) in Verilog have a scope (i.e., domain of definition) that is local to the module, function, task, or named block in which they are declared. They have meaning within that scope and cannot be referenced directly from outside it.[4]

[1]Keywords have a fixed, predefined meaning and may not be used for any other purpose.
[2]The complete set of primitives is described in Appendix A.
[3]See Appendix F for a formal description of the language syntax.
[4]Verilog has a mechanism for hierarchical dereferencing of an identifier in a design. See Appendix G.

a
b
c
y

```
...
nand (y, a, b, c);
...
```

FIGURE 4-2 A three-input ***nand*** gate and an example of its instantiation as a Verilog primitive.

A simulator uses built-in truth tables to form the outputs of primitives during simulation. The values of the inputs attached to a primitive are determined by the circuitry external to the primitive, just as the output of a combinational logic gate is determined by the values of its inputs. Figure 4-3 shows a list of instantiated primitives that are connected by wires[5] to have the functionality of a five-input and-or-invert (AOI) circuit.

A list of primitives describes the functionality of a design, with a fixed set of signal names. By itself, the list would require that the model be used only in an environment that has the same signal names. The list does not have a name either, but it would be helpful if it had a descriptive name by which we could identify its functionality. To circumvent these limitations, the functionality of a design and its interface to the environment are encapsulated in a Verilog ***module***. A ***module*** is declared by writing text that describes (1) its functionality, (2) its ports to/from the environment, and (3) its timing and other attributes of the design, such as the physical silicon area, that would be needed to implement it in on a chip.

Verilog is a standard HDL (IEEE 1364-1995, 2001, 2005). As such, it undergoes regular scrutiny by an industry-based committee to incorporate enhancements. The revisions adopted in 2001 incorporated ANSI-style syntax to reduce verbosity and increase clarity of descriptions. Fortunately, the evolved versions of the language all accommodate the syntax of their predecessors. Verilog modules using the 1995 syntax have the text format shown in Figure 4-4. The keywords ***module*** and ***endmodule*** encapsulate the text that describes the module having type-name *my_design*. A module's type-name is user-defined and distinguishes the module from others. The ports of the module are listed beside *my_design* in a comma-separated list enclosed by parentheses in Figure 4-4. Unlike primitives, modules do not have a restriction on the relative ordering of I/O ports in the list. The inputs and outputs can be listed in any order. The circuit in Figure 4-1 is an example of this syntax. You will notice that the declarations of the ports require redundant information – the list in parentheses contains the names of

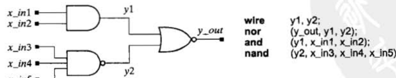


FIGURE 4-3 A list of declarations of wire-connected primitives having the functionality of a five-input AOI gate.

[5]We will see that the data-type ***wire*** in Verilog is used to establish connectivity in a design, just as a physical wire establishes connectivity between gates.

```
module my_design (module_ports);
... // Declarations of ports go here
... // Functional details go here
endmodule
```

FIGURE 4-4 The format of a Verilog module, with ***module . . . endmodule*** keyword encapsulation.

ports, and a separate list contains a declaration of the mode and name of each port. The ANSI-style syntax approved by IEEE in 2001 has the form shown below and eliminates redundant text.

```
module my_design ( /* Declarations of port mode and name go here*/);
// Functional details go here
endmodule
```

This edition of the text will use the 2001 syntax in most of its examples. Other changes in syntax will be presented and discussed where appropriate. The standard adopted by IEEE in 2005 made some clarifications but did not make additional changes to the syntax of the language. The declarations placed within a module determine whether the module will be structural, behavioral, or a combination of these. All the examples in this chapter will be structural models.

4.1.2 Verilog Structural Models

A structural model of a logic circuit is declared and encapsulated as a named Verilog module, consisting of (1) a module name accompanied by its ports, (2) a declaration of the operational modes of the ports (e.g., input), (3) an optional list of internal wires and/or other variables used by the model, and (4) a list of interconnected primitives and/or other modules, just as one would place and connect their physical counterparts on a PC board or on a schematic. The *primary* inputs and outputs of a physical circuit connect to its environment, and are the named ports of the model. In operation and in simulation, signals applied at the primary inputs interact with the internal gates to produce internal signals and the signals at the primary outputs. A designer could apply signal generators to the inputs of the actual circuit and observe the inputs and outputs on an oscilloscope or logic analyzer. A declared module can be referenced (instantiated) within the declaration of some other module to create more elaborate and complex structural models.

A complete Verilog structural model of a five-input AOI circuit is listed in Figure 4-5 in the 1995 and the 2001/2005 syntaxes to illustrate some Verilog terminology. (*All keywords in the text are shown in boldface italic type.*) The keywords ***module*** and ***endmodule*** enclose (encapsulate) the description.[6] The text between these keywords declares (1) the interface between the model and its environment (by declaring the port list and the mode of each port), (2) the wires that are used to connect the logic gates that model the circuit (by declaring *y*1 and *y*2 to have type ***wire***), and (3) the logic gate primitives and the configurations of the port signals that connect them together to form the circuit (by listing the gates and their ports).

[6] Appendix B contains a list of Verilog keywords.

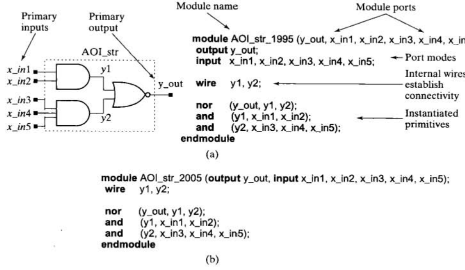


FIGURE 4-5 An AOI circuit and its Verilog model: (a) IEEE 1364-1995 syntax and (b) IEEE 1364-2002, 2005 syntax.

4.1.3 Module Ports

The ports of a module define its interface to the environment in which it is used. The *mode* of a port determines the direction that information (signal values) may flow through the port. A port's mode may be unidirectional (***input***, ***output***) or bidirectional (***inout***). Signals in the environment of a module are made available through its ***input*** ports; signals generated within a module are made available to the environment through its ***output*** ports. Bidirectional (***inout***) ports accommodate a flow of information in either direction. The mode of a module port is declared explicitly and is not determined by the order in which a port appears in the port list (but recall that outputs are the leftmost entries in the port list of a primitive).

The environment of a module interacts with its ports, but does not have access to the internal description of the module's functionality. Those details are hidden from the surrounding circuitry. The listing of declarations within a Verilog module tells a simulator how to form the output of the circuit from the values of its inputs. We will discuss simulation in more detail below.

4.1.4 Some Language Rules

Verilog is a case-sensitive language, so it matters whether you refer to a signal as *C_out_bar* or *C_OUT_BAR*. Verilog treats these as different names. An identifier (name) in Verilog is composed of a case-sensitive, space-free sequence of upper and lower case letters from the alphabet, the digits (0, 1, . . . , 9), the underscore (_), and the

$ symbol. The name of a variable may not begin with a digit or $, and may be up to 1024 characters long.[7] White space may be used freely, except in an identifier.

Usually, each line of text in a Verilog description must terminate with a semicolon (;). An exception is the terminating ***endmodule*** keyword. Comments may be imbedded in the source text in two ways. A pair of slashes, //, forms a comment from the text that follows it on the same line; the symbol-pair /* initiates a multi-line comment, and must be matched by the symbol-pair */ to terminate the scope of the comment. Multi-line comments may not be nested.

4.1.5 Top-Down Design and Nested Modules

Top-down design refers to the practice of systematically and repeatedly partitioning a complex system into simpler functional units whose design can be managed and executed successfully. A high-level partition and organization of the design is sometimes referred to as an *architecture*. The individual functional units that result from the partition are easier to design and simpler to test than larger, equivalent aggregates. The divide-and-conquer strategy of top-down design makes possible the design of circuits with several million gates. Top-down design is used in the most modern and sophisticated design methodologies that integrate entire systems on a chip (SoC) [2]. The instantiation of a module within the declaration of a different[8] module is referred to as a nested module. Nested modules are the Verilog mechanism supporting top-down design because nesting automatically creates a partition of the design.

MODELING TIP

Use nested module instantiations to create a top-down design hierarchy.

Example 4.1

A binary full adder circuit can be formed by combining two half adders and an OR gate as shown in the schematic in Figure 4-6(a). The Verilog hierarchical model of the partitioned design (Figure 4-6(b)) contains two instances of the module *Add_half*.

Modules may be nested within other modules, but not in a recursive manner. When a module is referenced by another module (i.e., when a module is listed inside the declaration of another module), a structural hierarchy is formed of the nesting/nested design objects. The hierarchy establishes a partition and represents relationships between the referencing and the referenced modules. The referencing module is called a *parent* module; the referenced module is called a *child* module. The module in which a child module is instantiated is a parent module. The two instances of *Add_half* within *Add_full* are child modules of *Add_full*. Note: Primitives are basic design objects. Although modules may have other modules and primitives nested within them, nothing can be instantiated (nested) within a primitive.

[7] The names of predefined system tasks begin with $.

[8] Do not attempt recursive declarations.

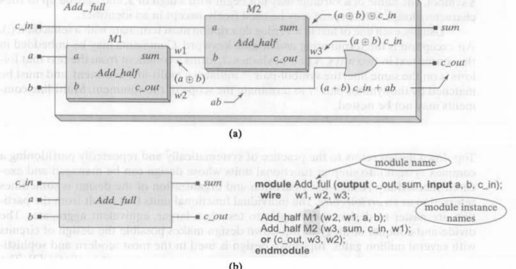


FIGURE 4-6 Hierarchical decomposition of a full adder: (a) gate-level schematic and (b) Verilog model.

A module hierarchy can have arbitrary depth. It is limited only by the capacity of the host computer's memory. Each instance of an instantiated module must be accompanied by a module instance name that is unique within its parent module. An instantiated primitive may be given a name, but that is not required.

End of Example 4.1

MODELING TIP

The ports of a module may be listed in any order.
An instantiated module must have an instance name.

Example 4.2

A 16-bit ripple-carry adder can be formed by cascading four 4-bit ripple-carry adders in a chain in which the carry generated by a unit is passed to the carry input port of its neighbor, beginning with the least significant bit. Each 4-bit adder is declared as a cascade of full adders. Figure 4-7 shows the partition and the port signals that are associated with each unit of *Add_rca_16*.

The complete text for *Add_rca_16* is given below.

```
module Add_rca_16 (output c_out, output [15: 0] sum, input [15: 0], a, b, input c_in);
  wire          c_in4, c_in8, c_in12, c_out;
  Add_rca_4 M1  (c_in4,     sum[3: 0],   a[3:0],    b[3:0],    c_in);
  Add_rca_4 M2  (c_in8,     sum[7:4],    a[7:4],    b[7:4],    c_in4);
  Add_rca_4 M3  (c_in12,    sum[11:8],   a[11:8],   b[11:8],   c_in8);
  Add_rca_4 M4  (c_outsum,  sum[15:12],  a[15:12],  b[15:12],  c_in12);
                c_out, a, b,
                c_in);
endmodule

module Add_rca_4 (output c_out, output [3: 0] sum, input [3 0] a, b, input c_in);
  wire          c_in2, c_in3, c_in4;
  Add_full      M1    (c_in2,    sum[0],    a[0], b[0], c_in);
  Add_full      M2    (c_in3,    sum[1],    a[1], b[1], c_in2);
  Add_full      M3    (c_in4,    sum[2],    a[2], b[2], c_in3);
  Add_full      M4    (c_out,    sum[3],    a[3], b[3], c_in4);
endmodule

module Add_full (output c_out, sum, input a, b, c_in)
  wire          w1, w2, w3;
  Add_half      M1 (w2, w1, a, b);
  Add_half      M2 (w3, sum, c_in, w1);
  or            M3 (c_out, w2, w3);
endmodule

module Add_half (output c_out, sum, input a, b);
  xor           M1 (sum, a, b);
  and           M2 (c_out, a, b);
endmodule
```

End of Example 4.2

4.1.6 Design Hierarchy and Source-Code Organization

The hierarchical model for *Add_rca_16* illustrates how Verilog supports top-down structured design by nesting modules within modules. Figure 4-8 shows the hierarchy for *Add_rca_16*. The top-level functional unit is encapsulated in *Add_rca_16*, and it contains instantiations of other functional units of lesser complexity, and so on. The lowest level of the hierarchy consists of primitives and/or modules that have no underlying hierarchical detail. All of the modules that compose a design must be placed in one or more text files that, when compiled together, completely describe the functionality of the top-level module. It does not matter how the modules are distributed across

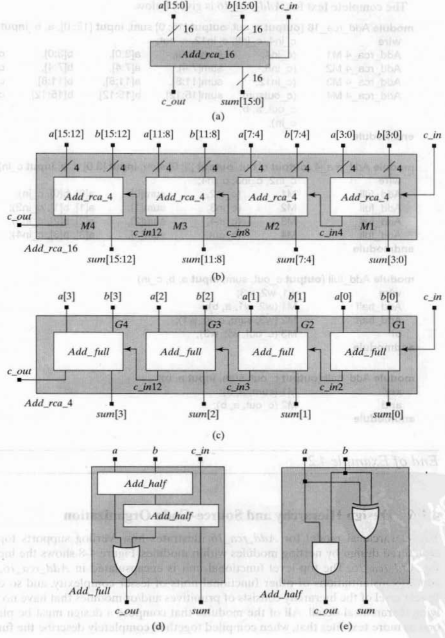


FIGURE 4-7 Hierarchical decomposition of a 16-bit, ripple-carry adder into a chain of four 4-bit-slice adders, each formed by a chain of full adders: (a) top-level schematic symbol, (b) decomposition into four 4-bit adders, (c) interior detail of a 4-bit adder, (d) a full adder, and (e) gate level schematic of a half adder.

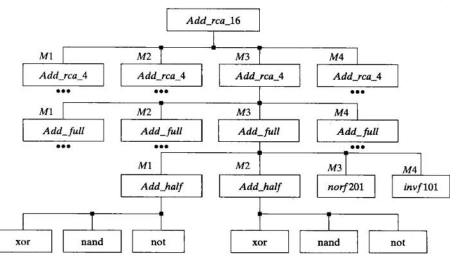


FIGURE 4-8 Design hierarchy of a 16-bit ripple-carry adder.

multiple source code files, as long as their individual descriptions reside in a single file. A simulator will compile designated source files and extract the modules that it needs in order to integrate a complete description of any design hierarchy that is implied by the port structures and references to nested/instantiated modules.

4.1.7 Vectors in Verilog

A vector in Verilog is denoted by square brackets, enclosing a contiguous range of bits, e.g., *sum[3:0]* represents four bits from *sum*, which was declared in *Add_rca_1* as a 16-bit signal. *The language specifies that, for the purpose of calculating the decimal equivalent value of a vector, the leftmost index in the bit range is the most significant bit, and the rightmost is the least significant bit.* An expression can be the index of a part-select. If the index of a part-select is out of bounds the value **x** is returned by a reference to the variable.[9] For example, if an 8-bit word *vect_word[7:0]* has a stored value of decimal 4, then *vect_word[2]* has a value of 1; *vect_word[3:0]* has a value of 4; and *vect_word[5:1]* has a value of 2.

[9]In Verilog's logic system, the symbol **x** represents an ambiguous (unknown) value, not a don't-care.

4.1.8 Structural Connectivity

Wires in Verilog establish connectivity between design objects. They connect primitives to other primitives and/or modules, and connect modules to other modules and/or primitives. By themselves, wires have no logic. The variable type *wire* is a member of a family of nets, all of which establish connectivity in a design.[10]

> **MODELING TIP**
>
> Use nets to establish structural connectivity.

The logic value of a ***wire*** (net) is determined dynamically during simulation by what is connected to the wire. If a *wire* is attached to the output of a primitive (module), it is said to be driven by the primitive (module), and the primitive (module) is said to be its driver. For example, in Figure 4-5, *y_out* is driven by a ***nor*** gate (primitive). The logic of the gate and the values of its inputs determine *y_out*. In that example we explicitly declared *y*1 and *y*2 to have type ***wire***, but did not have to do so. *According to the 2001/2005 syntax, any identifier that is referenced without having a type declaration is by default of type **wire**.*[11,12] Consequently, the input and output ports have default type ***wire*** too, unless we specifically declare them to have a different type (e.g., we will see that a variable may have type ***reg***).

> **MODELING TIP**
>
> An undeclared identifier is treated by default as a ***wire***.

The ports of an instantiated module must be connected in a manner that is consistent with the declaration of the module, but the names of the connecting signals need not be the same. In Example 4.2, the formal name of the second port of *Add_half* (i.e., the name given in the declaration of *Add_half*) is *Sum*, but in instance *M*1 the actual name of the port is *w*1. The actual ports were associated with the formal ports by their position in the port list. This mechanism works well in models that have only a few ports, but when the list of ports is large, it is easier and safer to associate ports by their names using the following convention in the port list: *.formal_name(actual_name)*. This connects *actual_name* to *formal_name*, regardless of the position of this entry in the list. The *formal_name* is the name given in the declaration of the instantiated module, and *actual_name* is the name used in the instantiation of the module. Association by name is more readable and less error prone than association by position.

---

[10] The Verilog family of predefined nets is described in Appendix C.

[11] Caution: The 1995 syntax is more restrictive and in some cases requires that an identifier be explicitly declared to have the type of a net (see [5]).

[12] The default net type can be changed by a compiler directive.

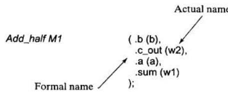


FIGURE 4-9 Formal and actual names for port association by name in module *Add_full.*

Example 4.3

The first ($M1$) instantiation of *Add_half* in *Add_full* can be written using port name association as shown in Figure 4-9.

End of Example 4.3

Our next example of a structural model will be used as a point of comparison with other examples to illustrate alternative styles of design with Verilog.

Example 4.4

A 2-bit comparator compares two 2-bit binary words, A and B, and asserts outputs indicating whether the decimal equivalent of word A is less than, greater than, or equal to that of word B. The functionality of the comparator is described by the set of Boolean equations[13] below, where $A1$ and $A0$ are the bits of A, and $B1$ and $B0$ are the bits of B.

$$A_lt_B = A1'\,B1 + A1'\,A0'\,B0 + A0'\,B1\,B0$$

$$A_gt_B = A1\,B1' + A0\,B1'\,B0' + A1\,A0\,B0'$$

$$A_eq_B = A1'\,A0'\,B1'\,B0' + A1'\,A0\,B1'\,B0 + A1\,A0\,B1\,B0 + A1\,A0'\,B1\,B0'$$

The Karnaugh map methods of Chapter 2 can be used to eliminate redundant logic from these equations and produce the generic gate-level description of the comparator shown in Figure 4-10. This gate-level, combinational logic implementation of the comparator can be modeled by a structural interconnection of Verilog primitives. Their aggregate behavior is that of the comparator circuit.

[13]+ denotes logical OR;' denotes logical complement. A1 A0 denotes the logical AND of A1 and A0.

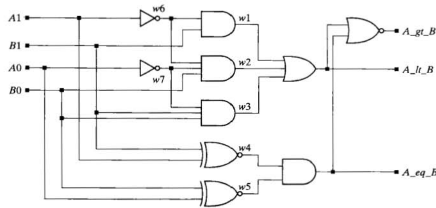


FIGURE 4-10 Schematic of a 2-bit binary comparator.

A structural Verilog description corresponding directly to the schematic of the 2-bit comparator is shown below. Notice that two of the instantiations of the ***and*** gate have three inputs, and the others have two inputs. This feature of the built-in primitives allows the same primitive to be used freely to accommodate whatever number of inputs are required by the context of its usage.

```
module Comp_2_str (output A_gt_B, A_lt_B, A_eq_B, input A0, A1, B0, B1);
  nor        (A_gt_B, A_lt_B, A_eq_B);
  or         (A_lt_B, w1, w2, w3);
  and        (A_eq_B, w4, w5);
  and        (w1, w6, B1);
  and        (w2, w6, w7, B0);
  and        (w3, w7, B0, B1);
  not        (w6, A1);
  not        (w7, A0);
  xnor       (w4, A1, B1);
  xnor       (w5, A0, B0);
endmodule
```

We will see in Chapter 6 that a synthesis tool can automatically optimize a gate-level description, remove redundant logic, and draw the resulting schematic. Next, we will use the 2-bit comparator as a building block to form a structural model of a 4-bit comparator.

End of Example 4.4

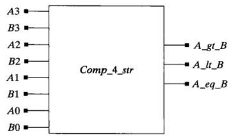


FIGURE 4-11 Block diagram symbol of a 4-bit comparator.

Example 4.5

A 4-bit comparator has the block diagram symbol shown in Figure 4-11. The comparator compares two 4-bit binary words and asserts outputs indicating their relative size. It would be cumbersome to write the Boolean equations for the outputs, so we will connect the outputs of two 2-bit comparators with additional logic to generate the appropriate outputs that result from comparing 4-bit words. The logic for connecting the 2-bit comparators is based on the observation that a strict inequality in the higher order bit-pair determines the relative magnitudes of the 4-bit words; on the other hand, if the higher-order bit-pairs are equal, the lower-order bit-pairs determine the output. The hierarchical structure shown in Figure 4-12 implements the 4-bit comparator, and the simulation results in Figure 4-13 display the assertions of the outputs for some values of the bits of the datapaths, with the vectors $A_bus = \{A3, A2, A1, A0\}$ and $B_bus = \{B3, B2, B1, B0\}$ formed in the testbench for the purpose of simulation.

The source code for module *Comp_4_str* is given below. It contains two instantiations of the module *Comp_2_str* that was declared in Example 4.4.

```
module Comp_4_str (
 output A_gt_B, A_lt_B, A_eq_B,
 input A3, A2, A1, A0, B3, B2, B1, B0);

 wire                 w1, w0;
 Comp_2_str M1 (A_gt_B_M1, A_lt_B_M1, A_eq_B_M1, A3, A2, B3, B2);
 Comp_2_str M0 (A_gt_B_M0, A_lt_B_M0, A_eq_B_M0, A1, A0, B1, B0);
 or                   (A_gt_B, A_gt_B_M1, w1);
 and                  (w1, A_eq_B_M1, A_gt_B_M0);
 and                  (A_eq_B, A_eq_B_M1, A_eq_B_M0);
 or                   (A_lt_B, A_lt_B_M1, w0);
 and                  (w0, A_eq_B_M1, A_lt_B_M0);
endmodule
```

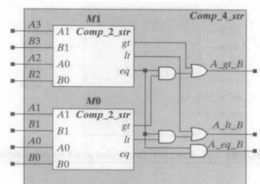


FIGURE 4-12 Hierarchical structure of a 4-bit binary comparator composed of 2-bit comparators and glue logic.

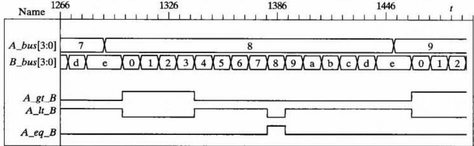


FIGURE 4-13 Simulation results for a 4-bit binary comparator.

End of Example 4.5

4.2 Logic System, Design Verification, and Test Methodology

Language-based models of a circuit must be verified to assure that their functionality conforms to the specification for the design. Two methods of verification are used: logic simulation and formal verification. Logic simulation applies stimulus patterns to

a circuit and monitors its simulated behavior to determine whether it is correct. Formal verification uses elaborate mathematical proofs to verify a circuit's functionality without having to apply stimulus patterns. Formal methods are used by industry to verify large complex circuits; we will consider only logic simulation.

4.2.1 Four-Value Logic and Signal Resolution in Verilog

Verilog uses a four-valued logic system having the symbols: 0, 1, x, and z. The language's abstract modeling constructs and the truth tables of its built-in primitives are defined for all four values of a primitive's inputs.[14] A logic simulator creates symbolic input waveforms in this four-value logic system and generates the internal and symbolic output signals for a circuit.

In Verilog's four-value logic system, the values 1 and 0 correspond, respectively, to assertion (True) or de-assertion (False) of a signal. The symbols for a signal in an actual circuit will have only these values, but simulators can accommodate additional symbolic logic values. The value ***x*** represents a condition of ambiguity in which the simulator cannot determine whether the value of the signal is 0 or 1. This happens, for example, when a net is driven by two primitives that have opposing output values. The primitive gates that are built into Verilog are able to model automatically this kind of contention between signals. (Verilog also has models for open collector and emitter follower logic in which signal contention is resolved by the technology itself to form either wired-and or wired-or structures, respectively [1].)

MODELING TIP

The logic value ***x*** denotes an unknown (ambiguous) value.
The logic value ***z*** denotes a high impedance.

The logic value ***z*** denotes a three-state condition in which a wire is disconnected from its driver. Figure 4-14 shows the waveforms that a simulator would produce in simulating an ***and*** primitive that is driven by signals that range over all of the possible input values. The waveforms in Figure 4-15 demonstrate how a Verilog simulator resolves multiple drivers on a net. Note that the three-state primitives[15] produce an output value of ***z*** when they are not enabled, and that the value of ***x*** is produced when a ***wire*** is driven by opposing values. In practice, great care is taken to ensure that a bus does not have contending multiple drivers active at the same time.

[14] Appendix A describes Verilog's built-in primitives and their truth tables in Verilog's four-value logic system.
[15] See Appendix A.

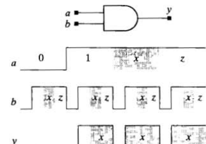


FIGURE 4-14 Four-value logic waveforms for the ***and*** primitive.

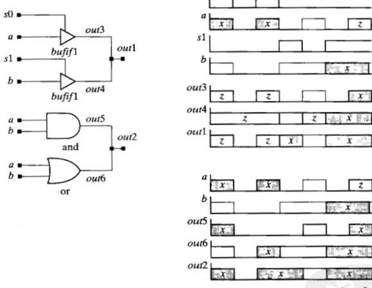


FIGURE 4-15 Resolution of multiple drivers on a net.

4.2.2 Test Methodology

A large circuit must be tested and verified systematically to ensure that all of its logic has been exercised and found to be functionally correct, i.e., that the model actually describes the functionality that we intend to implement. Later, in Chapter 11, testing verifies that the timing specifications as well as the functionality, of the design are met, and that the circuit has no process-induced faults (defects).

A haphazard test methodology makes debugging very difficult, creates a false sense of security that the model is correct, and risks enormous loss should a product fail as a result of an untested area of logic. In practice, design teams write an elaborate test plan that specifies the features that will be tested and the process by which they will be tested. The plan must also identify features that will not be tested.

Example 4.6

The partition of a 16-bit ripple-carry adder into four bit-slices of 4 bits each was shown in Figure 4-7. Each of the four-bit units was built as a chain of full adders, and each full adder was composed of a simple hierarchy of half-adders and glue logic. An attempt to verify the 16-bit adder by applying 16-bit stimulus patterns to its datapaths and an additional bit for its carry-in bit requires checking the results of applying 2^{33} input patterns! It would be foolhardy to attempt this if a simpler strategy can be found. Applying 2^{33} patterns consumes a lot of processing time, and any error that might be detected would be difficult to associate with the underlying circuit. A more clever strategy is to verify that the half adder and full adder each work correctly. Then, the 4-bit slice unit can be verified exhaustively by applying only 2^9 patterns! Once these simpler design units have been verified, the 16-bit adder can be tested to work correctly for a carefully chosen set of patterns that check the connectivity between the four units. This dramatically reduces the number of patterns and focuses the debugging effort on a much smaller portion of the overall circuitry. (See the problems section at the end of the chapter.)

End of Example 4.6

We saw that modeling begins with a complex functional unit and partitions it in a top-down fashion to enable the design of simpler units. Systematic verification proceeds in the opposite direction, beginning with the simpler units and moving to the more complex units above them in the design hierarchy. A basic methodology for verifying the functionality of a digital circuit consists of building a testbench that applies stimulus patterns to the circuit and displays the waveforms that result. The user, or software, can verify that the response is correct. The testbench is a separate Verilog module that has the basic organization shown in Figure 4-16. It resides at the top of a new design hierarchy that contains a stimulus generator, a response monitor, and an instantiation of the unit under test (UUT). The stimulus generator uses Verilog statements to define the patterns that are to be applied to the circuit. During simulation the response monitor selectively gathers data on signals within the design and displays them in a text or graphical format. Testbenches can be very complex, containing a variety of pattern generators and additional software to perform analysis on the gathered data to detect and report functional errors.

A simulator performs several essential tasks: it (1) checks the source code, (2) reports any violations of the language's syntax,[16] (3) determines whether the number of ports

[16] Appendix F presents the formal syntax of the Verilog language.

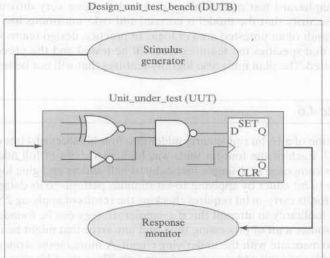


FIGURE 4-16 Organization of a testbench for verifying a unit under test.

specified by a model are actually connected to it in an instantiation, and (4) simulates the behavior of the circuit under the application of input signals that are defined in the testbench. Syntax errors must be eliminated before a simulation can run, but be aware that absence of syntax errors does not imply that the functionality of the model is correct.

Example 4.7

The module below, *t_Add_half*, has the basic structure of a testbench for verifying *Add_half* in a simulator having a graphical user interface.[17] Notice that it contains an instantiation of the UUT, *Add_half*. The waveforms that are to be applied to the UUT are not generated by hardware. Instead, they are generated abstractly by a single-pass Verilog behavior, declared by the keyword ***initial*** and accompanied by statements enclosed by the block statement keyword pair ***begin*** ... ***end***. In this simple example, the user serves as the response monitor by comparing the output waveforms to their expected values. Constructs used in a testbench are introduced in *t_Add_half*, and will be explained in the next section.

```
module t_Add_half();
  wire                  sum, c_out;
  reg                   a, b;
  Add_half M1 (c_out, sum, a, b);        // UUT
```

[17]See "Selected System Tasks and Functions" at the companion web sites (www.pearsonhighered.com/ciletti) for built-in system tasks, such as ***$monitor***, which can be used to display simulation results in a text format.

```
  initial begin                          // Time Out
    #100 $finish;
  end
  initial begin                          // Stimulus patterns
    #10 a = 0; b = 0;
    #10 b = 1;
    #10 a = 1;
    #10 b = 0;
  end
endmodule
```

End of Example 4.7

4.2.3 Signal Generators for Testbenches

A Verilog *behavior* is a group of statements that execute during simulation to assign value to simulation variables as though they were driven by hardware. The keyword ***initial*** declares a single-pass behavior that begins executing when the simulator is activated, at $t_{sim} = 0$ (we use t_{sim} to denote the time base of the simulator). The statements that are associated with the behavior are listed within the ***begin*** ... ***end*** block keywords, and are called *procedural* statements. When each procedural statement (e.g., $b = 1$;) in the testbench *t_Add_half* in Example 4.7 executes, it assigns a value to a variable. Using the assignment operator, =, such statements are called "procedural assignments."

MODELING TIP

Use procedural assignments to describe stimulus patterns in a testbench.

The time at which a procedural assignment statement in a ***begin*** ... ***end*** block executes depends on its order in the list of statements, and on the delay time preceding the statement (e.g., #10). The statements execute sequentially from top to bottom and from left to right across lines of text that may contain multiple statements. In this example, each line is preceded by a time delay (e.g., 10 simulator time units) that is prescribed with a *delay control operator* (#) and a delay value, for example, #10. A delay control operator preceding a procedural assignment statement suspends its execution and, consequently, suspends the execution of the subsequent statements in the behavior until the specified time interval has elapsed during simulation. A single-pass behavior expires when the last statement has executed, but (in general) the simulation does not necessarily terminate, because other behaviors might still be active.

The general structure of a testbench consists of one or more behaviors that generate waveforms at the inputs to the UUT, monitor the simulation data, and control the overall sequence of activity. Note that the inputs to the UUT in *t_Add_half* are

assigned value abstractly by a single-pass behavior and are declared by the keyword ***reg***, which indicates that the variables (signals) *a* and *b* are getting their values from execution of procedural statements (just like in an ordinary procedural language, such as C, but interpreted as waveforms that evolve under the control of the simulator). Since hardware is not driving the value of an abstractly generated input, the type declaration ***reg*** ensures that the value of the variable will exist from the moment it is assigned by a procedural statement until execution of a later procedural statement changes it. The outputs of the UUT are declared as wires. Think of them as providing the ability to observe the output ports of the UUT.

The waveforms that result from simulating *t_Add_half* are shown in Figure 4-17. In Verilog, nets are assigned the value *z* when simulation begins,[18] and they then inherit the value implied by their driver. An undriven net will remain at a value of *z*. All variables that have type ***reg*** are given the value ***x*** when the simulation begins, and they hold that value until they are assigned a different value. The cross-hatched waveform fill pattern in Figure 4-17 denotes the value ***x***, which results from having the inputs (of ***reg*** type) holding their initial value of ***x*** until time 10, when the first explicit assignment is made. The indicated simulation time steps are dimensionless, unless a timescale directive is used to associate physical units with numerical values.[19] Note the correspondence between the waveforms specified by the stimulus generator in the testbench and the simulated waveforms applied to the circuit. Execution of each statement is delayed by 10 time steps, so the final statement in the stimulus generator executes at time step 40 (i.e., in this example, the delays accumulate). The testbench includes a separate behavior for a stopwatch that terminates the simulation after 100 time steps. The stopwatch executes a built-in Verilog system

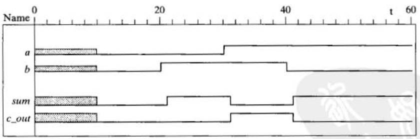


FIGURE 4-17 Waveforms produced by a simulation of *Add_half*, a 0-delay binary half adder.

[18] There is an exception, the so-called **trireg** net, which defaults to an initial value of x. This type of net has relevance in switch-level modeling, and will not be considered in this text.

[19] See Verilog's compiler directives at the companion web site (www.pearsonhighered.com/ciletti).

task, ***$finish***, which surrenders control to the operating system of the host machine when time step 100 is reached. A stopwatch is optional and not needed here, but, in general, it will prevent endless execution if the testbench otherwise fails to terminate the simulation.

> **MODELING TIP**
>
> A Verilog simulator assigns an *initial* value of **z** to all nets, which then inherit the value of their drivers; a simulator assigns an initial value of **x** to variables that have type ***reg***.

4.2.4 Event-Driven Simulation

A change in the value of a signal (variable) during simulation is referred to as an "event." The one-pass behavior in *t_Add_half* programmed the inputs to *Add_half* to change at prescribed times, independently of the signals in the circuit. However, the events of the outputs of the UUT, *sum* and *c_out*, depend on the occurrence of events at the inputs to the device, just as the signals in a physical circuit change in response to the signals at the input. By knowing (1) the schedule of events at the inputs and (2) the structure of a circuit, we can create a schedule for the events at the outputs [3].

The simulators used for logic simulation are said to be *event-driven* because their computational activity is driven by the propagation of events in a circuit. Event-driven simulators are inactive during the interval between events. When an event occurs at the input to the UUT, the simulator schedules updating events for the internal signals and outputs of the UUT. Then, while time continues to advance, the simulator rests until the next triggering event occurs at the input. All of the gates and abstract behaviors are active concurrently [4, 5], and it is the simulator's job to detect events and schedule any new events that result from their occurrence.

4.2.5 Testbench Template

Testbenches are an important tool in the design flow of an ASIC. Much effort is expended to develop a thorough testbench because it is an insurance policy against failure. A test plan should be developed *before* the testbench is written. The plan, at a minimum, should specify what functional features will be tested (e.g., the net circuit's three-state outputs) and how they will be tested in the testbench. It is also important for a test plan to identify what features are not tested by the testbench (e.g., simulation with **x** and **z** values applied at the inputs to the circuit). Testbenches are customized to the UUT, but the simple structure given below can be used to develop testbenches for the problems at the ends of the chapters.

```
module t_DUTB_name ( );       // substitute the name of the UUT
  reg ...;                    // Declaration of register variables for primary
                              // inputs of the UUT
  wire ...;                   // Declaration of primary outputs of the UUT
```

```
parameter                        time_out = // Provide a value
UUT_name M1_instance_name (UUT ports go here);
initial $monitor ( );            // Specification of signals to be monitored and
                                 // displayed as text20
initial #time_out $finish;       // Stopwatch to assure termination of simulation
initial                          // Develop one or more behaviors for pattern
                                 // generation and/or error detection
 begin
                                 // Behavioral statements generating waveforms
                                 // to the input ports, and comments documenting
                                 // the test. Use the full repertoire of behavioral
                                 // constructs for loops and conditionals.
 end
endmodule
```

4.2.6 Sized Numbers

The values assigned to the stimulus waveforms in the testbench in Example 4.7 are sized numbers. *Sized numbers* specify the number of bits that are to be stored for a value. For example, 8'*ha* denotes an 8-bit stored value corresponding to the hexadecimal number *a*; the binary value in memory will be 0000_1010.[21] Unsized numbers are stored as integers having length determined by the host simulator (at least 32 bits). Four formats are available: binary (b), decimal (d), octal (o), and hexadecimal (h). The format specifier is not case-sensitive, and by default a number is interpreted to be a decimal value.

4.3 Propagation Delay

Physical logic gates have a propagation delay between the time that an input changes and the time that the output responds to the change. The primitives in Verilog have a default delay of 0, meaning that the output responds to the input immediately, but a nonzero delay can also be associated with a primitive. Timing verification ultimately depends on realistic values of the propagation delays in a circuit, but simulation is commonly done with 0 delay to verify the functionality of a model quickly. Simulation with a unit delay is often done, too, because it exposes the time sequence of signal activity, which can be masked by 0-delay simulation.

MODELING TIP

All primitives and nets have a default propagation delay of 0.

[20]Only one **$monitor** statement can be active at a time.

[21]In Verilog, the underscore may be inserted in the representation of a number to make it more readable.

Example 4.8

The primitives within *Add_full_unit_delay* and *Add_half_unit_delay* are shown below with annotation that assigns to them a unit delay. The delay notation #1 has been inserted before the instance name of each instantiated primitive (# denotes the delay control operator). The effect of the delay is apparent in the simulated transitions of *sum* and *c_out* shown in Figure 4-18. Notice that the 0-delay simulation results in Figure 4-17 do not reveal whether *c_out* is formed before or after *sum*. Both, in fact, appear to change as soon as the inputs change. The waveforms of the unit-delay model reveal the time-ordering of the signal activity in Figure 4-18.

```
module Add_full_unit_delay (output c_out, sum, input a, b, c_in);
  wire                  w1, w2, w3;
  Add_half_unit_delay   M1 (w2, w1, a, b);
  Add_half_unit_delay   M2 (w3, sum, w1, c_in);
  or                    #1 M3 (c_out, w2, w3);
endmodule
module Add_half_unit_delay  (output c_out, sum, input a, b);
  xor                   #1 M1 (sum, a, b);
  and                   #1 M2 (c_out, a, b);
endmodule
```

ASICs are fabricated by assembling onto a common silicon die the logic cells from a standard-cell library. The library cells are predesigned and precharacterized so that their Verilog model includes accurate timing information, and synthesis tools use this information to optimize the performance (speed) of a design. Our focus in this book will be on synthesizing gate-level structures from technology-independent[22]

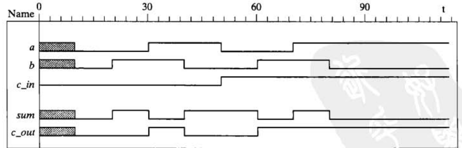


FIGURE 4-18 Results of unit-delay simulation of a 1-bit full adder.

[22]Technology-independent behavioral models describe only the functionality of a circuit, not its propagation delays.

behavioral models of circuits, using either standard cells or field-programmable gate arrays (FPGAs). The timing characteristics of the latter are embedded within the synthesis tools for FPGAs; the timing characteristics of the former are embedded within the model of the cell and are used by a synthesis tool to analyze the timing of a circuit in conjunction with selecting parts from a cell library to realize specified logic. Circuit designers do not attempt to create accurate gate-level timing models of a circuit by manual methods. Instead, they rely on a synthesis tool to implement a design that will satisfy timing constraints. We will address this topic in Chapter 10.[23]

End of Example 4.8

Example 4.9

The models of *Add_half_ASIC* and *Add_full_ASIC* shown below use parts (*norf201*, *invf101*, *xorf201*, and *nanf201*) from a standard-cell library.[24] Complementary metal-oxide semiconductor (CMOS) *and* gates are commonly implemented by combining a *nand* gate with an inverter; likewise, an *or* gate is implemented by a *nor* gate and an inverter. The Verilog timescale directive '**timescale** 1ns / 1ps at the first line of the source file directs the simulator to interpret the numerical time variables within the standard cells as having units of nanoseconds, with a resolution of picoseconds.[25]

```
'timescale 1 ns / 1 ps
module Add_full_ASIC
  (output c_out, sum, input a, b, c_in);

  wire           w1, w2, w3;
  wire           c_out_bar;
  Add_half_ASIC  M1 (w2, w1, a, b);
  Add_half_ASIC  M2 (w3, sum, w1, c_in);
  norf201        M3 (c_out_bar, w2, w3);
  invf101        M4 (c_out, c_out_bar);
endmodule
```

```
module Add_half_ASIC
  (output c_out, sum, input a, b);
  wire     c_out_bar;
  xorf201  M1 (sum, a, b);
  nanf201  M2 (c_out_bar, a, b);
  invf101  M3 (c_out, c_out_bar);
endmodule
```

The models for *norf201*, *invf101*, *xorf201*, and *nanf201* include propagation delays based on characterization of the physical standard cells. The effect of the realistic propagation delays is apparent in the waveforms produced by simulating *Add_full_ASIC*, as shown in Figure 4-19.

[23] See "Additional Features of Verilog" at the companion web site for additional information about modeling propagation delay.
[24] *norf201* is a two-input nor gate standard cell; *invf101* is an inverter, *xorf201* is a two-input excusive-or gate standard cell; and *nanf201* is a two-input nand standard cell.
[25] See "Compiler Directives" at the companion Web site for a description of time scales.

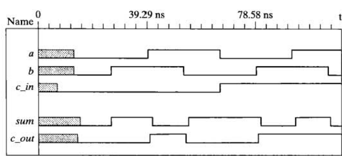


FIGURE 4-19 Results of simulating a 1-bit full adder implemented with ASIC cells having technology-dependent propagation delays.

End of Example 4.9

4.3.1 Inertial Delay

Logic transitions in a digital circuit correspond to transitions in voltage levels caused by the accumulation or dissipation of charge at a physical node/net. The physical behavior of a signal transition is said to have inertia, because every conducting path has some capacitance, as well as resistance, and charge cannot accumulate or dissipate instantly. HDLs must be able to model these effects.

The propagation delay of the primitive gates in Verilog obeys an inertial delay model. This model accounts for the fact that charge must accumulate or dissipate in the physical circuit before a voltage level can be established corresponding to a 0 or a 1. If an input signal is applied to a gate and then removed before sufficient charge has accumulated the output signal will not reach a voltage level corresponding to a transition. For example, if all the inputs to a ***nand*** gate are at value 1 for a long time before one of them is changed momentarily to 0, the output will not change to 1 unless the input is held to 0 for a long enough time. The amount of time that the input pulse must be constant in order for the gate to make a transition is the inertial delay of the gate.

Verilog uses the propagation delay of a gate as the minimum width of an input pulse that could affect the output; that is the value of propagation delay is also used as the value of the inertial delay. The width of a pulse must be at least as long as the propagation delay of the gate. The Verilog simulation engine detects whether the duration of an input has been too short, and then deschedules previously scheduled outputs that were triggered by the leading edge of a pulse. Inertial delay has the effect of suppressing input pulses whose duration is shorter than the propagation delay of the gate.

Example 4.10

The input to the inverter in Figure 4-20 changes at $t_{sim} = 3$. Because the inverter has a propagation delay of 2, the effect of this change is to cause the output to be scheduled to change at $t_{sim} = 5$. However, for pulsewidth $\Delta = 1$, the input changes back to the initial value at $t_{sim} = 4$. The simulator cannot anticipate this activity. The effect of the two successive changes is to create a narrow pulse at the input to the inverter. Because the pulsewidth is less than the propagation delay of the inverter, the simulator deschedules the previously scheduled output event corresponding to the leading edge of the narrow input pulse, and does not schedule the event corresponding to the trailing edge of the pulse. Descheduling is required because the simulator cannot anticipate the falling edge and must wait until it occurs. On the other hand, the pulse with $\Delta = 6$ persists sufficiently long for the output to be affected.

In physical circuits, the propagation delay of a logic gate is affected by its internal structure and by the circuit that it drives. The internal delay is referred to as the intrinsic delay of the gate. In a circuit, the driven gates and the metal interconnect of their fanin nets create additional capacitive loads on the output of the driving gate and affect its timing characteristics. The slew rate of the input signal, which represents the slope of a signal's transition between logic values, can also have an affect on the transitions of the output signal. Accurate standard-cell models of gates account for all these effects.

End of Example 4.10

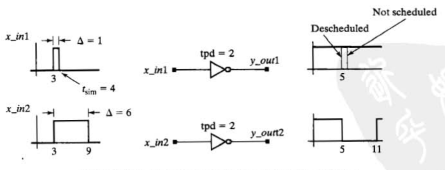


FIGURE 4-20 Event descheduling caused by an inertial delay.

4.3.2 Transport Delay

The time-of-flight of a signal traveling a wire of a circuit is modeled as a *transport delay*. With this model narrow pulses are not suppressed, and all transitions at the driving end of a wire appear at the receiving end after a finite time delay. In most ASICs, the physical distances are so small that the time-of-flight on wires can be ignored, because at the speed of light the signal takes only 0.033 ns to travel 1 cm. However, Verilog can assign delay to individual wires in a circuit to model transport delay effects in circuits where it cannot be neglected, such as in a multichip hardware module or on a printed circuit board. Wire delays are declared with the declaration of a wire. For example, ***wire*** *#2 A_long_wire* declares that *A_long_wire* has a transport delay of two time steps.

4.4 Truth Table Models of Combinational and Sequential Logic with Verilog

Verilog supports truth-table models of combinational and sequential logic. Although Verilog's built-in primitives correspond to simple combinational logic gates, and do not include sequential parts, the language has a mechanism for building user-defined primitives (UDPs), which use truth tables to describe sequential behavior and/or more complex combinational complex logic. UDPs are widely used in ASIC cell libraries because they simulate faster and require less storage than modules.

Example 4.11

The text below declares *AOI_UDP*, a truth table version of the five-input AOI circuit that was introduced in Section 4.1.2.

```
primitive AOI_UDP (output y, input x_in1, x_in2, x_in3, x_in4, x_in5);   // Note: Verilog
2001/2005 syntax
   table
   // x1 x2 x3 x4 x5 : y
      0 0 0 0 0 : 1;
      0 0 0 0 1 : 1;
      0 0 0 1 0 : 1;
      0 0 0 1 1 : 1;
      0 0 1 0 0 : 1;
      0 0 1 0 1 : 1;
      0 0 1 1 0 : 1;
      0 0 1 1 1 : 0;
      0 1 0 0 0 : 1;

      0 1 0 0 1 : 1;
      0 1 0 1 0 : 1;
```

```
        0 1 0 1 1 : 1;
        0 1 1 0 0 : 1;
        0 1 1 0 1 : 1;
        0 1 1 1 0 : 1;
        0 1 1 1 1 : 0;

        1 0 0 0 0 : 1;
        1 0 0 0 1 : 1;
        1 0 0 1 0 : 1;
        1 0 0 1 1 : 1;
        1 0 1 0 0 : 1;
        1 0 1 0 1 : 1;
        1 0 1 1 0 : 1;
        1 0 1 1 1 : 0;

        1 1 0 0 0 : 0;
        1 1 0 0 1 : 0;
        1 1 0 1 0 : 0;
        1 1 0 1 1 : 0;.
        1 1 1 0 0 : 0;
        1 1 1 0 1 : 0;
        1 1 1 1 0 : 0;
        1 1 1 1 1 : 0;
    endtable
endprimitive
```

UDPs are declared in a source file in the same way that a module is declared, but with the encapsulating keyword pair ***primitive*** ... ***endprimitive***. They can be instantiated just like built-in primitives, with or without propagation delay. A UDP has only a single, scalar (single-bit), output port; also, the input ports of a UDP must be scalars.

End of Example 4.11

MODELING TIP

The output of a UDP must be a scalar.

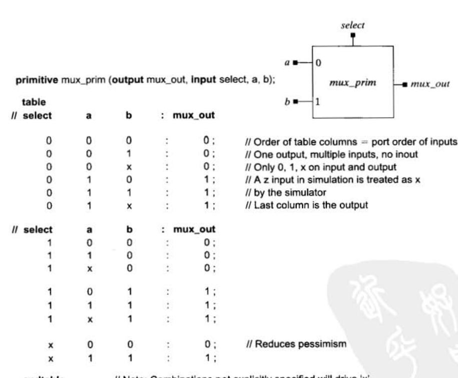
The truth table for a UDP consists of a section of columns, one for each input, followed by a colon and a final column that specifies the output. The order of the input columns must conform to the order in which the input ports are listed in the declaration.[26] A simulator references the table whenever one of its inputs changes and proceeds from the top toward the bottom of the table searching for a match. The search terminates at the first match, ignoring the remaining rows of the table.

[26]See "Rules for UDPs" at the companion web site (www.pearsonhighered.com/ciletti) for additional features and rules of use for UDPs.

Example 4.12

The Verilog UDP *mux_prim*, shown in Figure 4-21 describes a two-input multiplexer and includes comments citing some basic rules for UDP models.

A simulator will automatically assign the (default) value x to the output of a UDP if its inputs have values that do not match a row of the table. An input value of z is treated as x by the simulator. The last two rows of the table describing the behavior of the multiplexer in Figure 4-21 reduce the pessimism that might result during simulation. If both data inputs have the same value the output has that value, regardless of the value of the *select* input. When the value of *select* is x (ambiguous), the output is 0 if both inputs are 0, and 1 if both inputs are 1. If the UDP table overlooks this additional detail, a simulator will propagate a value of x to the output when *select* is x. It is generally




```
primitive mux_prim (output mux_out, input select, a, b);

 table
// select    a    b   : mux_out

      0      0    0   :    0;    // Order of table columns = port order of inputs
      0      0    1   :    0;    // One output, multiple inputs, no inout
      0      0    x   :    0;    // Only 0, 1, x on input and output
      0      1    0   :    1;    // A z input in simulation is treated as x
      0      1    1   :    1;    // by the simulator
      0      1    x   :    1;    // Last column is the output

// select    a    b   : mux_out
      1      0    0   :    0;
      1      1    0   :    0;
      1      x    0   :    0;

      1      0    1   :    1;
      1      1    1   :    1;
      1      x    1   :    1;

      x      0    0   :    0;    // Reduces pessimism
      x      1    1   :    1;

 endtable        // Note: Combinations not explicitly specified will drive 'x'
 endprimitive    // under simulation.
```

FIGURE 4-21 UDP for a two-input multiplexer.

PDG

desirable to reduce the situations under which a primitive propagates a value of *x* because ambiguity may reduce the amount of useful information that can be derived from a simulation.

End of Example 4.12

The entries for the inputs in a truth table can be reduced by using a shorthand notation. The **?** symbol allows an input to take on any of the three values, 0, 1, and *x*. This allows one table row to effectively replace three rows.

Example 4.13

When the *select* input of a two-input multiplexer is 0 and the *a* channel is 0, the output is 0, regardless of the value of the input at the *b* channel. When *select* is a 1 and the *b* channel is a 0, the output is a 0 when the *a* channel is a *0, 1*, or *x*. The ? shorthand notation substitutes 0, 1, and *x* in the table row and effectively implements a don't-care condition on the associated input.

```
table
// Shorthand notation:
// ? represents iteration of the table entry over the values 0,1,x.
// i.e., don't care on the input
//      select    a      b      : mux_out
//      0         0      ?      :    0 ; //? = 0,1,x shorthand notation.
//      0         1      ?      :    1 ;
//      1         ?      0      :    0 ;
//      1         ?      1      :    1 ;
//      ?         0      0      :    0 ;
//      ?         1      1      :    1 ;
endtable
```

End of Example 4.13

Hardware devices can exhibit two basic kinds of sequential behavior: level-sensitive behavior (e.g., transparent latch), which is conditioned by an enabling input signal, and edge-sensitive behavior (e.g., a D-type flip-flop), which is synchronized by an input signal. Level-sensitive devices respond to any change of an input while the enabling input is asserted; edge-sensitive devices ignore their inputs until a synchronizing edge occurs. Sequential hardware devices may have the behavior of one or the other, or a combination of both.

A truth table that describes sequential behavior has input columns followed by a colon (:) and a column for the present state of the device, and another colon and a

column for its next state, that is, the state that will be caused by the present inputs. In addition, the output of the UDP must be declared to have type ***reg*** because the value of the output is produced abstractly, by a table, and must be held in memory during simulation.

MODELING TIP

The output of a sequential UDP must be declared to have type ***reg***.

Example 4.14

A truth-table description of a transparent latch is given below by *latch_rp*. It describes transparent behavior and latching behavior, and also deals with the possibility that, under simulation, the input *enable* might acquire an ***x*** value.

```
primitive latch_rp (output reg q_out, input enable, data);
  table
//            enable    data        state       q_out/next_state
              1         1       :   ?       :   1 ;
              1         0       :   ?       :   0 ;
              0         ?       :   ?       :   - ;
// Above entries do not deal with enable = x.
// Ignore event on enable when data = state:
              x         0       :   0       :   - ;
              x         1       :   1       :   - ;
// Note: The table entry '-' denotes no change of the output.
   endtable
endprimitive
```

The transparent latch modeled by *latch_rp* exhibits level-sensitive behavior; that is, the output can change any time that an input changes, depending on the value of the inputs. The value that is scheduled for the output is determined only by the value of the *enable* and *data* inputs. In contrast, a truth table describing edge-sensitive behavior will be activated whenever an input has an event, but whether the output changes depends on whether a synchronizing input has made an appropriate transition. For example, a flip-flop that is sensitive to the rising edge of its clock would have represent an edge transition of the clock by having an entry of (01) in the corresponding column of the table. UDPs can describe behavior that is sensitive to either the positive or negative edge (transition) of a clock signal, with built-in semantics for positive (***posedge***) and negative (***negedge***) signal transitions. A falling edge (***negedge***) transition is denoted by the following signal value pairs: (10), (1x), and (x0); rising edges are denoted by (01), (0x), and (x1).

End of Example 4.14

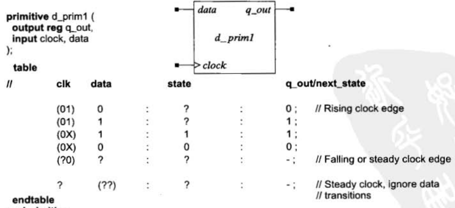
Example 4.15

The UDP *d_prim1* in Figure 4-22 describes the behavior of an edge-sensitive D-type flip-flop. The input signal *clock* synchronizes the transfer of *data* to *q_out*.

The table-entry notation for a sequential behavior uses parentheses to enclose the defining logic values of a signal whose transition affects the output (i.e., the synchronizing input signal). In the table in Figure 4-22, the (01) entry in the column for *clock* denotes a low-to-high transition of the signal *clock*—that is a value change. Note, also, that the row corresponding to the entry of (?0) for *clock* actually denotes 27 input possibilities and replaces 27 rows of entries, as there are two more symbols ? in that row. For example, (?0) represents (00), (10) and (x0). Each of these is combined with three possibilities for the data; each of the resulting nine possibilities is combined with three possibilities for the state. In effect, this row explicitly specifies that the output should not change in any of these situations. Since the model represents the physical behavior of a rising-edge sensitive behavior, the output should not change on a falling edge, or if there is no edge at all (00). Were this row omitted the model would propagate an x value under simulation. Remember, it is desirable that the UDP table be as complete and unambiguous as possible.

End of Example 4.15

A truth table can include both level-sensitive behavior and edge-sensitive behavior to model synchronous behavior with asynchronous set and reset conditions. Because a simulator will search the truth table from top to bottom, the level-sensitive behavior should precede the edge-sensitive behavior in the table.




```
primitive d_prim1 (
  output reg q_out,
  input clock, data
);
  table
//        clk    data      state        q_out/next_state

          (01)   0     :    ?     :     0 ;   // Rising clock edge
          (01)   1     :    ?     :     1 ;
          (0X)   1     :    1     :     1 ;
          (0X)   0     :    0     :     0 ;
          (?0)   ?     :    ?     :     - ;   // Falling or steady clock edge

          ?      (??)  :    ?     :     - ;   // Steady clock, ignore data
                                              // transitions
  endtable
endprimitive
```

FIGURE 4-22 Truth-table model of a D-type flip-flop.

Example 4.16

A J-K flip-flop having asynchronous *preset* and *clear* with edge-sensitive sequential behavior is described in Figure 4-23. The *preset* and *clear* inputs are active-low, and the output is sensitive to the rising edge of the clock. The signal *clock* synchronizes the changes of *q_out*. Depending on the values of the *preset* and *clear* signals when the clock edge occurs, *q_out*

```
primitive jk_prim (output reg q_out, input clk, j, k, preset, clear);
 table
 //   clk    j    k    pre   clr       state   q_out/next_state
 // Preset logic
      ?      ?    ?    0     1     :   ?    :   1;
      ?      ?    ?    *     1     :   1    :   1;

 // Clear logic
      ?      ?    ?    1     0     :   ?    :   0;
      ?      ?    ?    1     *     :   0    :   0;

// Normal clocking
//    clk    j    k    pre   clr       state   q_out/next_state

      r      0    0    0     0     :   0    :   1;
      r      0    0    1     1     :   ?    :   -;
      r      0    1    1     1     :   ?    :   0;
      r      1    0    1     1     :   ?    :   1;
      r      1    1    1     1     :   0    :   1;
      r      1    1    1     1     :   1    :   0;
      f      ?    ?    ?     ?     :   ?    :   -;

 // j and k cases
 //   clk    j    k    pre   clr       state   q_out/next_state
      b      *    ?    ?     ?     :   ?    :   -;
      b      ?    *    ?     ?     :   ?    :   -;

 // Reduced pessimism

      p      0    0    1     1     :   ?    :   -;
      p      0    ?    1     ?     :   0    :   -;
      p      ?    0    ?     1     :   1    :   -;
      (?0)   ?    ?    ?     ?     :   ?    :   -;
      (1x)   0    0    1     1     :   ?    :   -;
      (1x)   0    ?    1     ?     :   0    :   -;
      (1x)   ?    0    ?     1     :   1    :   -;
      x      *    0    ?     1     :   1    :   -;
      x      0    *    1     ?     :   0    :   -;
 endtable
endprimitive
```

FIGURE 4-23 UDP for a J-K flip-flop.

PDG

does not change ($j = 0, k = 0$), *q_out* gets a value of 0, ($j = 0, k = 1$), *q_out* gets a value of 1, ($j = 1, k = 0$), or *q_out* is toggled ($j = 1, k = 1$).[27]

End of Example 4.16

REFERENCES

1. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*, Language Reference Manual, IEEE Standard 1364–1995. Piscataway, NJ: Institute of Electrical and Electronic Engineers, 1996.
2. Chang H, et al., *Surviving the System on Chip Revolution*. Boston, MA: Kluwer, 1999.
3. Thomas R, Moorby P. *The Verilog Hardware Description Language*, Boston: Kluwer, 2008.
4. Ciletti MD. *Modeling, Synthesis and Rapid Prototyping with the Verilog HDL*. Upper Saddle River, NJ: Prentice-Hall, 1999.
5. Ciletti MD. *Starter's Guide to Verilog 2001*. Upper Saddle River, NJ: Prentice-Hall, 2004.

PROBLEMS

For all of the problems requiring development and verification of a design, a test plan, a testbench (see Example 4-7 for an example of a testbench), and simulation results are to be provided with the solution. Supporting material, such as a state-transition graph or a truth table should be included too.

1. Using Verilog gate-level primitives, develop and verify a structural model for the circuit shown in Figure P4-1. Use the following name for your testbench, the model, and its ports: *t_Combo_str(), Combo_str (Y, A, B, C, D)*. *Note*: The testbench will have no ports. Simulate the circuit exhaustively (i.e., for all values of the inputs) and provide graphical output demonstrating that the model is correct. *Note*: If text output is desired, consider using the ***$monitor*** and ***$display*** tasks.

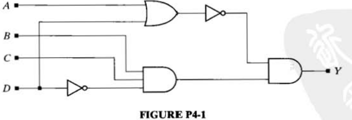


FIGURE P4-1

[27] The symbol * in a UDP table denotes all transitions of an input.

2. Repeat Problem 4.1 and develop *Combo_prim*, a user-defined primitive implementing a truth-table model of the logic. Then instantiate the UDP in a module, *Combo_UDP*. Verify that the model is correct and that its response matches that of *Combo_str* in Problem 1.
3. Develop a testbench that contains patterns exercising the structural connectivity of the 4-bit units in *Add_rca_16* (see Example 4.2). Verifying the structural connectivity assumes that the internal modules (i.e., *Add_rca_4*) are correct and do not need to be re-verified. However, to assure structural connectivity, it is necessary to verify that the inputs and outputs to *Add_rca_4* are connected correctly, and that the carry chains between the internal modules are connected correctly.
4. Modify the UDP *d_prim1* (see Figure 4-22) to develop a new UDP for a D-type flip-flop having an active-high reset input, then develop and verify a Verilog structural model for a circuit that converts a serial bit stream (LSB first) for a binary coded decimal digit into an Excess-3 encoded digit (see Figure 3-22).
5. Develop and exercise a testbench to verify the functionality of *d_prim1* (see Example 4.15). Include a written test plan describing (1) the functional features that will be tested, (2) how they will be tested, and (3) features that will not be tested by your testbench. Annotate your testbench with comments to associate its elements with the test plan.
6. Develop and exercise a testbench to verify the functionality of *jk_prim* (see Example 4.16). Include a written test plan describing (1) the functional features that will be tested and (2) how they will be tested. Annotate your testbench with comments to associate its elements with the test plan.
7. Write a gate-level model describing the following Boolean functions: $Y1(A, B, C, D) = \Sigma\, m\,(4, 5, 6, 7, 11, 12, 13)$, $Y2(A, B, C, D) = \Sigma m(1, 2, 4, 5)$.
8. Develop and verify a UDP for a negative-edge triggered D-type flip-flop that has active-low reset.
9. Develop and exercise a testbench that verifies the functionality of *AOI_str* in Section 4.1.2.
10. Develop and exercise a testbench that verifies the functionality of *latch_rp* in Example 4.14.
11. Develop a small set of test patterns that will (1) test a half-adder circuit, (2) test a full-adder circuit, (3) exhaustively test a 4-bit ripple-carry adder, and (4) test a 16-bit ripple-carry adder by verifying that the connectivity between the 4-bit slices are connected correctly, given that the 4-bit slices themselves have been verified.
12. Develop and exercise a testbench (including a test plan) to verify a gate-level model of a full adder.
13. DevelopD and exercise a testbench (including a test plan) to verify a gate-level model of an S-R (set–reset) latch.
14. Develop and verify a Verilog module that produces 4-bit output code indicating the number of 1s in an 8-bit input word.
15. Develop and verify a structural (gate-level) description of the circuit shown in Figure P4-15.

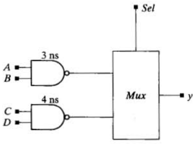


FIGURE P4-15

16. After modifying the circuit shown in Figure P4-15 to have its output exhibit three-state behavior, develop and verify a structural (gate-level) model of the circuit.
17. Develop and verify a structural (gate-level) Verilog module of a bidirectional 4-bit ring counter capable of counting in either direction, beginning with the first active clock edge after reset.
18. Develop and verify a structural (gate-level) Verilog module of a decade counter—that is, a counter whose count sequence is 0, 1, 2, ..., 9, 0, 1, 2 ... repeatedly. Hint: consider a user-defined primitive of a D-type flip-flop.
19. The schematic shown in Figure P4-19 is for *Divide_by_11*, a frequency divider that divides *clk* by 11 and asserts its output for one cycle. The unit consists of a chain of toggle-type flip-flops with additional logic to form an output pulse every 11th pulse of *clk*. The asynchronous signal *rst_b* is active-low and drives *Q* to 1. Develop and verify a model of *Divide_by_11*. Note: you will need to develop a model of a T-type flip-flop.

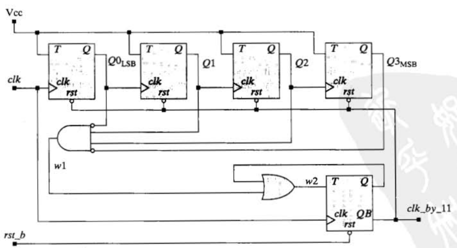


FIGURE P4-19

CHAPTER 5

Logic Design with Behavioral Models of Combinational and Sequential Logic

In Chapter 2 we presented schematics, Boolean equations, and truth table descriptions of combinational logic, and Chapter 3 reviewed fundamental models and manual methods for designing sequential logic. Chapter 4 presented Verilog counterparts of schematics and truth tables. This chapter will present Verilog models based on Boolean equations and introduce a more general and abstract style of modeling for combinational and sequential logic. Algorithmic state machine (ASM) charts will be introduced for developing behavioral models of finite-state machines. More general sequential machines consisting of a datapath and a control unit will be designed by algorithmic state machine and datapath (ASMD) charts, which link a state machine to the datapath that it controls. We will also present some results of synthesizing behavioral models of circuits.

5.1 Behavioral Modeling

Verilog supports structural and behavioral modeling. Structural modeling connects primitive gates and/or functional units to create a specified functionality (for example, an adder) just as parts are connected on a chip or a circuit board. But gate-level models are not necessarily the most convenient or understandable models of a circuit, especially when the design involves more than a few gates. Many modern application-specific integrated circuits (ASICs) have several million gates on a single chip! Also, truth tables become unwieldy when the circuit has several inputs and/or multiple outputs, limiting

the utility of Verilog's user-defined primitives. In today's methodologies for designing ASICs and field-programmable gate arrays (FPGAs), large circuits are partitioned to form an architecture—that is, a configuration of functional units that communicate through their ports—such that each unit is described by a behavioral model of its functionality. Even though the architecture consists of simpler functional units than the whole, designers do not form gate-level implementations of the simpler units directly. Instead, they write so-called behavioral models, which can be synthesized automatically into a gate-level circuit (structure).

Traditionally, designers have designed at the gate (structural) level, using schematic-entry tools to connect gates to form a circuit, but modern design methodology uses a hardware description language (HDL) to decompose and represent a design at a meaningful level of abstraction. Synthesis tools automatically translate and map the HDL into a physical technology, such as an ASIC standard-cell library, or a library of programmable parts, such as FPGAs.

Behavioral modeling is the predominant descriptive style used by the industry, enabling the design of massive chips. Behavioral modeling describes the functionality of a design—that is, *what* the designed circuit will do, not *how* to build it in hardware. Behavioral models specify the input–output model of a logic circuit and suppress details about its low-level internal structure (architecture) and physical implementation. Propagation delays are not included in the behavioral model of the circuit, but the propagation delays of the cells in the target technology are considered by the synthesis tool when it imposes timing constraints on the physical realization of the logic. Behavioral modeling encourages designers to (1) rapidly create a behavioral prototype of a design (without binding it to hardware detail), (2) verify its functionality, and then (3) use a synthesis tool to optimize and map the design into a selected physical technology, subject to constraints on timing and/or area. If the behavioral model has been written in a synthesis-ready style,[1] the synthesis tool will remove redundant logic, analyze tradeoffs between alternative architectures and/or multilevel equivalent circuits, and ultimately achieve a design that is compatible with area or timing specifications/constraints. By focusing the designer's attention on the functionality that is to be implemented, rather than on individual logic gates and their interconnections, behavioral modeling provides the freedom to explore alternatives and tradeoffs before committing a design to production.

Aside from its importance in synthesis, behavioral modeling provides flexibility to a design project by allowing portions of the design to be modeled at different levels of abstraction and completeness. The Verilog language accommodates mixed levels of abstraction, so that portions of the design that are implemented at the gate level (i.e., structurally) can be integrated and simulated with other portions of the design that are represented by behavioral descriptions.

[1]Chapter 6 will consider how to write synthesis-ready models of combinational and sequential logic in Verilog.

5.2 A Brief Look at Data Types for Behavioral Modeling

Before considering behavioral models, we must first understand how information is represented and used in a Verilog model. All computer programs represent information as variables (e.g., integers and real numbers) that can be retrieved, manipulated, and stored in memory. A variable may represent a number used in computation (such as a loop index governing a repetitive sequence of steps), a value of data (such as a binary word), or a computed value (such as the sum of two numbers). In Verilog, a variable can also represent a binary-encoded logic signal in a circuit. A Verilog model describes how the waveforms of variables will evolve in a simulation environment. For example, the *sum* and *c_out* bits of the half adder described in Chapter 4 evolve in the manner specified by the interconnection of primitives in the model. Variables are declared and used according to rules that govern the data types[2] supported by the language.

All variables in Verilog have a predefined type, and there are only two families of data types: nets and registers. *Net variables* act like wires in a physical circuit and establish connectivity between design objects that represent hardware units. *Register variables* act like variables in ordinary procedural languages—they store information while the program executes. Not all data types are useful in a synthesis methodology, and we will use mainly the net type ***wire*** and the register types ***reg*** and ***integer***. A ***wire*** and a ***reg*** have a default size of 1 bit. The size of an ***integer*** is automatically fixed at the word length supported by the host computer, at least 32 bits.

5.3 Boolean Equation-Based Behavioral Models of Combinational Logic

A Boolean equation describes combinational logic by an expression of operations on variables. Its counterpart in Verilog is the continuous assignment statement.

Example 5.1

The five-input and-or-invert (AOI) circuit that was shown in Figure 4-7 can be described by a single continuous assignment statement that forms the output of the circuit from operations on its inputs, as shown in *AOI_5_CA0* below.

```
module AOI_5_CA0 (
  input          x_in1, x_in2, x_in3, x_in4, x_in5,
  output         y_out
);
 assign y_out = !((x_in1 && x_in2) || (x_in3 && x_in4 && x_in5));
endmodule
```

End of Example 5.1

[2]See Appendix C for a discussion of Verilog data types.

The keyword **assign** declares a *continuous assignment* and associates the Boolean expression on the right-hand side (RHS) of the statement with the variable on the left-hand side (LHS) to declare how the value of the LHS variable depends on the value of the expression on the RHS. Verilog has several built-in operators for arithmetic, logical, and machine-oriented operations (e.g., concatenation, reduction, and shifting operators) that can be used in expressions.

The expression in *AOI_5_CA0* in Example 5.1 uses the logical inversion (!) operator,[3,4] the logical AND operator (&&), and the logical OR operator (||). The assignment is said to be sensitive to the variables in the RHS expression because any time a variable in the RHS changes during simulation (i.e., there is an event), the RHS expression is reevaluated and the result is used to update the LHS. A continuous assignment is said to describe *implicit combinational logic* because the RHS expression of the continuous assignment is equivalent to logic gates that implement the same Boolean function, but note that a continuous assignment can be more compact and understandable than a schematic or a netlist of primitives.

Example 5.2

The five-input AOI circuit can be modified to have an additional input, *enable*, and to have a three-state output, as described by *AOI_5_CA1* below.

```
module AOI_5_CA1 (
  input         x_in1, x_in2, x_in3, x_in4, x_in5, enable,
  output        y_out
);
  assign y_out = enable ? !((x_in1 && x_in2) || (x_in3 && x_in4 && x_in5)) : 1'bz;
endmodule
```

The *conditional operator* (? :) acts like a software if-then-else switch that selects between two expressions. In this example, if the value of *enable* is true in *AOI_5_CA1*, the expression to the right of the *?* is evaluated and used to assign value to *y_out*; otherwise, the expression to the right of the *:* is used.[5] This example also illustrates how to write models that include three-state outputs. The value of *y_out* is formed by combinational logic while *enable* is asserted, but has the value *z* otherwise. This functionality corresponds to the logic circuit shown in Figure 5-1. Note that the continuous assignment statement is an implicit, abstract, and compact representation of the structure described by an equivalent gate-level schematic or netlist of primitives.

End of Example 5.2

A continuous assignment statement establishes an event-scheduling rule between a target net variable and a Boolean expression. A module may contain multiple continuous

[3]||, && and ! denotes the logical operations OR, AND, NOT, respectively. When the operands are scalar the corresponding bitwise operators may be used, but it is advisable to use logical operators where the intent is to express logic.

[4]See Appendix D for a discussion and more examples of Verilog operators.

[5]The syntax of the conditional operator requires that both alternative expressions be specified.

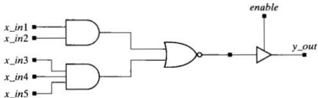


FIGURE 5-1 Equivalent circuit modeled by *AOI_5_CA1*.

assignments; the assignments are active concurrently with all other continuous assignments, primitives, behavioral statements, and instantiated modules, just as all of the electrical signals in a circuit are active concurrently. Continuous assignments can also be written implicitly and efficiently (without the keyword ***assign***) as part of the declaration of a wire.

Example 5.3

The model described by *AOI_5_CA2* implicitly declares *y_out* to be of the type ***wire,*** and associates with it a Boolean expression that defines the value of *y_out*. An output port that is not declared to be of the type ***reg*** is implicitly of the type ***wire.***

```
module AOI_5_CA2 (
  input        x_in1, x_in2, x_in3, x_in4, x_in5, enable,
  output       y_out
);
  assign y_out = enable ? !((x_in1 && x_in2) || (x_in3 && x_in4 && x_in5)) : 1'bz;
endmodule
```

End of Example 5.3

A continuous assignment statement uses built-in Verilog operators to express how a signal's value is formed abstractly. Each operator has a gate-level counterpart, so such expressions are easily synthesized into physical circuits.

Example 5.4

A continuous assignment statement with a conditional operator provides a convenient way to model the multiplexer circuit shown in Figure 5-2. In *Mux_2_32_CA* an event for signal *select* or for a selected datapath will cause *mux_out* to be updated during simulation.

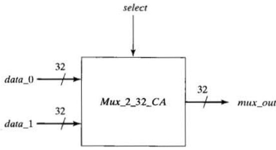


FIGURE 5-2 A two-channel mux with 32-bit datapaths.

```
module Mux_2_ 32_CA #(parameter word_size = 32) (
   output   [word_size -1: 0]        mux_out,
   input    [word_size -1: 0]        data_1, data_0,
   input                             select
);
  assign  mux_out = select ? data_1 : data_0;
endmodule
```

End of Example 5.4

Note that the size of the 32-bit datapath in *Mux_2_32_CA* is specified by a parameter to make the model scalable, portable, and more useful. Parameters are constant values, and remain fixed during simulation.

5.4 Propagation Delay and Continuous Assignments

Propagation (inertial) delay can be associated with a continuous assignment so that its implicit logic has the same functionality and timing characteristics as its gate-level counterpart.

Example 5.5

The functionality of an AOI gate structure with unit propagation delays on its implicit gates is described below by *AOI_5_CA3*. Each declaration of a wire includes a logic expression assigning value to the wire, and includes a unit time delay. The three continuous assignments are active concurrently during simulation, each having a monitoring mechanism, implemented by the simulator, which detects a change in its RHS expression

and schedules a change to update the LHS variable (subject to the propagation delay), just as the equivalent combinational logic would operate under the influence of its inputs.

```
module AOI_5 _CA3 (output y_out, input x_in1, x_in2, x_in3, x_in4);
  wire #1 y1 = x_in1 && x_in2;
  wire #1 y2 = x_in3 && x_in_4;
  wire #1 y_out = !(y1 || y2);
endmodule
```

End of Example 5.5

Example 5.6

As an alternative to the structural model of a 2-bit comparator that was presented in Example 4.4, the model *Comp_2_CA0* given below is described by three (concurrent) continuous assignment statements (implicit combinational logic).[6] The model is equivalent to *Comp_2_str* but has no explicit binding to hardware or to primitive gates.

```
module Comp_2_CA0 (
  input A1, A0, B1, B0,
  output A_lt_B, A_gt_B, A_eq_B
);
  assign A_lt_B = (!A1) && B1 || (!A1) && (!A0) && B0 || (!A0) && B1 && B0;
  assign A_gt_B = A1 && (!B1) || A0 && (!B1) && (!B0) || A1 && A0 && (!B0);
  assign A_eq_B = (!A1) && (!A0) && (!B1) && (!B0) || (!A1) && A0 && (!B1) && B0
        || A1 && A0 && B1 && B0 || A1 && (!A0) && B1 && (!B0);
endmodule
```

End of Example 5.6

Note that continuous assignment statements suppress detail about the internal structure of the module, and deal only with the Boolean equations that describe the input–output relationships of the comparator. A synthesis tool will create the actual hardware realization of the assignment.

The three Verilog language constructs corresponding to schematic, truth table and Boolean equation descriptions of combinational logic are show in Figure 5-3. All

[6]The order in which multiple continuous assignments are listed in the source code is arbitrary; that is, the order of the statements does not establish a precedence for their evaluation and has no effect on the results of simulation.

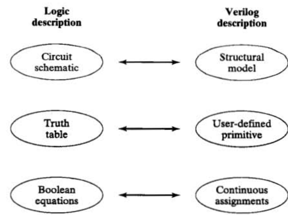


FIGURE 5-3 Verilog counterparts of three common descriptions of combinational logic.

three descriptions describe level-sensitive behavior; that is, variables are updated immediately when an input changes during simulation. None of the examples modeled feedback structures, but we will see that a continuous assignment with feedback is a synthesizable model for a hardware latch.

5.5 Latches and Level-Sensitive Circuits in Verilog

The level-sensitive storage mechanism of a latch (see Chapter 3) can be modeled in a variety of ways. First, note that a set of continuous assignment statements has implicit feedback if a variable in one of the RHS expressions is also the target of an assignment. For example, a pair of cross-coupled NAND gates could be modeled as

```
assign q = set ~& qbar;
assign qbar = rst ~& q;
```

The implied behavior will still be level sensitive, but it will correspond to the feedback structure of a hardware latch. Synthesis tools do not accommodate this form of feedback, but they do support the feedback that is implied by a continuous assignment in which the RHS uses a conditional operator, as shown in Example 5.7.

Example 5.7

The output of a transparent latch follows the data input while the latch is enabled, but otherwise will hold the value it had when the *enable* input was de-asserted. Example 4.14 presented a truth table model of a transparent latch. Here, *Latch_CA* uses a continuous assignment statement with feedback to model this functionality.

```
module Latch_CA (output q_out, input data_in, enable);
  assign q_out = enable ? data_in: q_out;
endmodule
```

Figure 5-4 shows the waveforms produced by simulation of *Latch_CA*. Note how *q_out* follows *data_in* while *enable* is asserted, and latches *q_out* to the value of *data_in* when *enable* is de-asserted. The appearance of *q_out* in the RHS expression and as the LHS target variable implies a feedback structure in hardware, and will be synthesized as a latch.

End of Example 5.7

When feedback is used in a continuous assignment statement with a conditional operator, a synthesis tool will infer the functionality of a latch and its hardware implementation. Chapter 6 will discuss descriptive styles that lead to intentional and accidental synthesis of latches.

Example 5.8

The latch model *Latch_Rbar_CA* below uses a nested conditional operator to add the functionality of an active-low reset to a transparent latch. Simulation of *Latch_Rbar_CA* produces the waveforms shown in Figure 5-5, in which the actions of *enable* and *rst_b* are apparent.

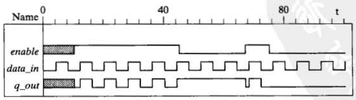


FIGURE 5-4 Simulation results for a transparent latch modeled by a continuous assignment statement with feedback.

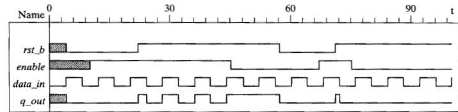


FIGURE 5-5 Simulation results for a transparent latch with active-low *reset* and active-high *enable*.

```
module Latch_Rbar_CA (
  output        q_out,
  input         data_in, enable, rst_b
);
 assign q_out = !(rst_b == 1'b0) ? 0 : enable ? data_in : q_out;
endmodule
```

End of Example 5.8

Verilog supports multiple descriptive styles that can define the same functionality. Continuous assignments are convenient for modeling small Boolean expressions, three-state behavior, and transparent latches. But designers writing large Boolean equation models (continuous assignments) are prone to making mistakes when there are several variables and large expressions. Also, the Boolean expressions might obscure the functionality of the design, even if they are written correctly. So, it is worthwhile to consider other language constructs that offer simpler, but more readable alternatives that describe edge-sensitive as well as level-sensitive behavior.

5.6 Cyclic Behavioral Models of Flip-Flops and Latches

Continuous assignment statements are limited to modeling level-sensitive behavior—combinational logic and transparent latches. They cannot model an element that has edge-sensitive behavior, such as a flip-flop. Many digital systems operate synchronously, with activity triggered by an edge of a synchronizing signal (commonly called a clock). Verilog uses a *cyclic behavior* to model edge-sensitive functionality. Like the single-pass behaviors that are used to model signal generators in testbenches (see Example 4.7), cyclic behaviors are abstract—they do not use hardware to specify signal values. Instead, they execute procedural statements to generate the values of variables,

just as the statements of an ordinary procedural language (e.g., C) execute to extract, manipulate, and store variables in memory. They are called cyclic behaviors because they do not expire after their last procedural statement has executed; instead, they re-execute. The execution of these statements can be unconditional or can be governed by an optional event-control expression or delay-control expression. Cyclic behaviors are used to model (and synthesize) both level-sensitive and edge-sensitive (synchronous) behavior (e.g., flip-flops).

Example 5.9

The keyword ***always*** in *df_behav* declares a cyclic behavior corresponding to an edge-triggered flip-flop. At every rising edge of *clk* the behavior's procedural statements execute, computing the value of *q* and storing it in memory. A continuous assignment statement forms *q_bar* from *q* immediately after *q* has changed.[7] The nonblocking, or concurrent, assignment operator (<=) will be explained later.

```
module df_behav (output reg q, output q_bar, input data, set_b, reset_b, clk);
  assign q_bar = !q;
  always @ (posedge clk)     // Synchronous set/reset
   if (reset_b == 1'b0) q <= 0;
     else if (set_b == 1'b0) q <= 1;
     else q <= data;
   end
endmodule
```

End of Example 5.9

In *df_behav* the action of *rst_b* is synchronous because it has no influence until the procedural statements are evaluated at the active edge of *clk*. The variable *q* retains its residual value until the next active edge of *clk*, as specified by ***posedge*** *clk*, because *q* was declared as a register variable of type ***reg***.

The operator (<=) in a procedural statement is called a *nonblocking assignment operator*. A variable that is assigned value by a nonblocking assignment operator in a single-pass or cyclic behavior must be a declared register-type variable (i.e., not a net). All register variables store information during simulation, but they do not necessarily synthesize a hardware register.[8]

[7]The bitwise operator ~ is commonly used to express an inverter. Since *q_bar* is a scalar variable the operator has the same effect as the operator !.

[8]Chapter 6 will discuss whether a register variable in a model synthesizes to a hardware storage element.

5.7 Cyclic Behavior and Edge Detection

A cyclic behavior is activated at the beginning of simulation, and it will execute its associated procedural statements subject to timing control imposed by delay-control (# operator) and event-control expressions (@ operator).[9] The Verilog keyword ***posedge*** qualifies an event-control expression to execute its procedural statements only when a rising edge of the argument signal (e.g., *clk* in Example 5.9) has occurred. Edge semantics for rising (***posedge***) and falling (***negedge***) edges are built into Verilog.

A simulator automatically monitors the variables in an event-control expression, and when the expression changes value, the associated procedural statements execute if the enabling change took place. When all the statements of a cyclic behavior complete execution, the computational activity flow returns to the ***always*** keyword and commences execution again, subject to its event-control expression. If a delay-control operator or an event-control expression is encountered in one of the statements being executed, the activity flow of the behavior is suspended to wait until the indicated time has elapsed or until the event-control expression detects the qualifying activity. The conditionals in the procedural statements in the cyclic behavior in *df_behav* (***if*** and ***else if***) test whether the associated expression evaluates to *true*. If it does, the associated statement (or ***begin*** ... ***end*** block statement) is executed.

Example 5.10

The reset action of a flip-flop can be asynchronous. The functionality modeled below by *asynch_df_behav* is sensitive to the rising edge of the clock, but also to the falling edge of *rst_b* and *set_b*, with priority given to *rst_b*. The last clause in the conditional statement executes at a rising edge of *clk* only if the asynchronous inputs are not asserted.

```
module asynch_df_behav (
  input          data, set_b, rst_b, clk,
  output reg     q,
  output         q_bar
);
  always @ (posedge clk, negedge set_b, negedge rst_b)
    if (rst_b == 1'b0) q <= 0;
```

[9]A ***wait*** statement will also suspend execution, but it is not synthesized by the leading synthesis tools and will not be discussed here or used in our models.

```
      else if (set_b == 1'b0) q <= 1;
        else q <= data;                    // synchronized activity
endmodule
```

End of Example 5.10

Note that *clk* and *clock* are not keywords in the Verilog language, *so it is important to place in the last conditional clause of the **if** statement the computational activity associated with the synchronizing signal of a synchronous behavior*. This coding discipline allows a synthesis tool to correctly (1) identify the synchronizing signal (its name and its location in the event-control expression are not predetermined), and (2) infer the need for a flip-flop to hold the value of *q* between the active edges of the synchronizing signal.

The cyclic behavior in *asynch_df_behav* is activated at the beginning of simulation and immediately suspends until its event-control expression changes.[10] The expression is formed as an "event or" of *set, reset*, and *clk*.[11] The Verilog language allows a mixture of level-sensitive and edge-qualified variables in an event-control expression, but synthesis tools do not support such models of behavior. Be certain that your description is entirely edge sensitive or entirely level sensitive.

Example 5.11

A transparent latch is modeled in *tr_latch* by a cyclic behavior whose level-sensitive event-control expression is sensitive to a change in *enable* or a change in *data*.[12]

```
module tr_latch (output reg q_out, input data, enable);
  always @ (enable, data)
    if (enable == 1'b1) q_out <= data;
endmodule
```

When *enable* is asserted in *tr_latch* the behavior is activated, and *q_out* immediately gets the value of *data*. Then the activity flow returns to the ***always*** construct and

[10] An event for *clk* occurs at the beginning of simulation if it is assigned a value of 1.

[11] Verilog 2001 (see Appendix I) introduced the option to form the event-control expression more conveniently as a comma-separated sensitivity list. Prior to Verilog 2001 the event-control expression would be written as: enable **or** data.

[12] This is the preferred model of a transparent latch (1364.1 IEEE Standard for Verilog Register Transfer Level Synthesis)

suspends to await the next change of the event-control expression. If *data* changes while *enable* is asserted, the cycle of *q_out* getting *data* repeats. The control flow of the *if* statement has no branch, so while *enable* is de-asserted *q_out* retains the value it had when *enable* was de-asserted. While *enable* is de-asserted the events of *data* reactivate the process, but no assignment is made to *q_out*.

End of Example 5.11

5.8 A Comparison of Styles for Behavioral Modeling

We have already seen how the 2-bit comparator can be described by a gate-level structure (Example 4.4) and by a Boolean equation-based behavioral model (Example 5.6). Next, we compare simpler and more readable alternatives that also use continuous assignments, and then we contrast modeling styles based on (1) continuous assignments, (2) register transfer level (RTL) logic, and (3) behavioral algorithms.

5.8.1 Continuous Assignment Models

A modeling style based on continuous assignments describes level-sensitive behavior. Continuous assignments execute concurrently with each other, with gate-level primitives, and with all of the behaviors in a description.

Example 5.12

The functionality in *Comp_2_CA1* is evident from the expressions in the continuous assignment statements. Here, the Verilog concatenation operator ({ }) concatenates the bits of the datapaths to form 2-bit vectors. The Boolean value of the RHS expression determines the assignment of 0 or 1 to the LHS variable.[13] Note that the gate-level implementation is not apparent.

```
module Comp_2_CA1 (output A_lt_B, A_gt_B, A_eq_B, input A1, A0, B1, B0);
  assign          A_lt_B = ({A1, A0} < {B1, B0});
  assign          A_gt_B = ({A1, A0} > {B1, B0});
  assign          A_eq_B = ({A1, A0} == {B1, B0});
endmodule
```

End of Example 5.12

[13]In general, a Verilog expression is true if it evaluates to the binary equivalent of a positive integer and false otherwise.

Another simple, and more elegant, implementation of the 2-bit comparator uses continuous assignment statements and the relational operators, with declared 2-bit vectors A and B, as shown next.

Example 5.13

The RHS expression of the continuous assignment statements in *Comp_2_CA2* are sensitive to A and B and evaluate to 1 (true) or 0 (false).

```
module Comp_2_CA2 (output A_lt_B, A_gt_B, A_eq_B, input [1: 0] A, B);
  assign                          A_lt_B = (A < B);
  assign                          A_gt_B = (A > B);
  assign                          A_eq_B = (A == B);
endmodule
```

End of Example 5.13

Suppose now that we want to extend this model to compare two 32-bit words, as shown in Figure 5-6. It is not feasible to write the Boolean equations that compare 32-bit words.

Example 5.14

A 32-bit comparator has the same functionality as a 2-bit comparator. So we modify the model from the previous example by declaring a parameter to size the word length

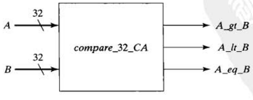


FIGURE 5-6 Block diagram symbol for a 32-bit comparator.

of the datapath. The description also declares a comma-separated list of continuous assignments with one statement. The model is readable, understandable, compact, and extendable to datapaths of arbitrary sizes.

```
module compare_32_CA #(parameter word_size = 32)(
  output A_gt_B, A_lt_B, A_eq_B,
  input [word_size -1: 0] A, B
);
  assign A_gt_B = (A > B);
  assign A_lt_B = (A < B);
  assign A_eq_B = (A == B);

endmodule
```

End of Example 5.14

5.8.2 Dataflow/RTL Models

Dataflow models of combinational logic describe concurrent operations on signals, usually in a synchronous machine, where computations are initiated at the active edges of a clock and are completed in time to be stored in a register at the next active edge. At each active edge, the hardware registers read and store the data inputs that were formed as a result of activity that began at the previous clock edge, and then propagate new values to be stored in registers at the next edge. Dataflow models for synchronous machines are also referred to as RTL (register transfer level) models because they describe register activity in a synchronous machine [1, 2]. RTL models are written for a specific architecture—that is, the registers, datapaths, and machine operations and their schedule are known a priori.

A behavioral model of combinational logic can be described by a set of concurrent continuous assignments (see Examples 5.6, and 5.13) or by an equivalent asynchronous (i.e., level-sensitive) cyclic behavior. Remember that cyclic behaviors are declared by the keyword ***always***, execute statements in sequential order, and re-execute indefinitely.

Example 5.15

The level-sensitive cyclic behavior in *Comp_2_RTL* executes and updates the outputs whenever a bit of either datapath changes.

```
module Comp_2_RTL (output reg A_lt_B, A_gt_B, A_eq_B, input A1, A0, B1, B0);

  always @ (A0, A1, B0, B1) begin
    A_lt_B =   ({A1, A0} < {B1, B0});
    A_gt_B =   ({A1, A0} > {B1, B0});
    A_eq_B =   ({A1, A0} == {B1, B0});
  end

endmodule
```

End of Example 5.15

The assignment operator in Example 5.15 is the ordinary procedural assignment operator (=). Consequently, the statements are executed in the listed order, with the storage of value occurring immediately after any statement executes and before the next statement can execute. Because there are no data dependencies between the variables on the LHS of the three procedural assignments in *Comp_2_RTL*, the order in which the assignments are listed does not affect the outcome. That is not always the case.

Example 5.16

The shift register shown in Figure 5-7 is described below by a synchronous cyclic behavior with a list of procedural assignments using the blocking assignment operator (=).

```
module shiftreg_PA (output reg A, input E, clk, rst);
 reg B, C, D;
 always @ (posedge clk, posedge rst) begin
  if (rst == 1'b1) begin A = 0; B = 0; C = 0; D = 0; end
  else begin
   A = B;
   B = C;
   C = D;
   D = E;
  end
 end

endmodule
```

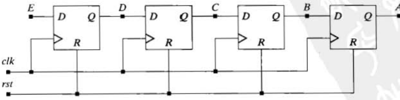


FIGURE 5-7 A 4-bit serial shift register.

Now consider what happens if the order of the procedural assignments in the model is reversed, as in *shiftreg_PA_rev* below. The list of statements executes in sequential order, from top to bottom. The effect of the assignment made by the first procedural statement is immediate. So *D* changes, and the updated value is used in the second statement, and so forth. The statements execute sequentially, but at the same time step of the simulator. The list of four statements is equivalent to a single statement that assigns *E* to *A*. Synthesis tools recognize this form of expression substitution, and will synthesize a circuit consisting of a single flip-flop, shown in Figure 5-8.

```
module shiftreg_PA_rev (output reg A, input E, clk, rst);
 reg B, C, D;

 always @ (posedge clk, posedge rst) begin
  if (rst == 1'b1) begin A = 0; B = 0; C = 0; D = 0; end
  else begin
   D = E;
   C = D;
   B = C;
   A = B;
  end
 end

endmodule
```

End of Example 5.16

Procedural assignments using the = operator are called *blocked* assignments (or blocked procedural assignments). The statements that follow a blocked procedural assignment cannot execute until the statement with the blocking assignment operator (=) completes execution and updates memory. This sets the stage for expression substitution. *Failure to appreciate the effects of expression substitution can lead to incorrect models.*

An alternative Verilog dataflow model uses concurrent procedural assignments, also called nonblocking assignments (or nonblocking procedural assignments).

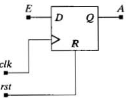


FIGURE 5-8 Circuit synthesized as a result of expression substitution in an incorrect model of a 4-bit serial shift register.

PDG

Nonblocking assignments are made with the nonblocking assignment operator (<=), instead of the = assignment operator. Nonblocking assignment statements effectively execute concurrently (in parallel) rather than sequentially, so the order in which they are listed has no effect. Moreover, a simulator must implement a sampling mechanism by which all of the variables referenced by the RHS of the statements with nonblocking assignments are sampled, held in memory, and used to update the LHS variables concurrently.[14] Consequently, changes to the listed order of the nonblocking assignments do not affect the outcome of the assignments to the LHS variable because the assignments are based on the values that were held by the RHS variables immediately before the statements executed.

Example 5.17

An equivalent model of the 4-bit serial shift register shown in Figure 5-7 is described below with nonblocking assignment operators (<=).

```
module shiftreg_nb_V05 (
 output reg     A,
 input          E, clk, rst
);
 reg            B, C, D;

 always @ (posedge clk, posedge rst) begin
  if (rst == 1'b1) begin A <= 0; B <= 0; C <= 0; D <= 0; end
   else begin
    A <= B;       //     D <= E;
    B <= C;       //     C <= D;
    C <= D;       //     B <= C;
    D <= E;       //     A <= B;
   end
  end

endmodule
```

End of Example 5.17

The commented (//) nonblocking assignments in *shiftreg_nb* have reversed order and would lead to the same results in simulation and will synthesize to the same structure. The statements in a list of nonblocking assignments execute concurrently, without dependence on their relative order. This style describes the concurrency that is found in actual hardware and the register transfers that occur within synchronous machines.

[14]It is advisable to avoid having multiple behaviors assign value to the same variable because software race conditions make the outcome indeterminate. See Appendix G.

When a cyclic behavior executes nonblocking assignments, the simulator evaluates each of the RHS expressions before assigning values to their LHS targets. This, in general, prevents any interaction between the assignments and eliminates dependencies on their relative order. This is not the case for blocked procedural assignments because such statements execute in sequence, and only after the immediately preceding statement has completed execution and updated memory. If the functionality being modeled by a cyclic behavior does not depend on the sequence in which the statements are written, either blocking or nonblocking assignments can be used (see Example 5.15). However, if we are modeling logic that includes edge-driven register transfers, it is strongly recommended that the edge-sensitive (synchronous) operations be described by nonblocking assignments and that combinational logic be described with blocked assignments. This practice will, in general, prevent race conditions between combinational logic and register operations.

5.8.3 Algorithm-Based Models

A behavioral model described by a circuit's input–output algorithm is more abstract than an RTL description. The algorithm prescribes a sequence of procedural assignments within a cyclic behavior. The outcome of executing the statements determines the values of storage variables and, ultimately, the output of the machine. The algorithm described by the model does not have explicit binding to hardware, and it does not have an implied architecture of registers, datapaths, and computational resources. This style is most challenging for a synthesis tool because it must perform what is referred to as *architectural synthesis*, which extracts the resources (e.g., determines actual requirements for processors, datapaths, and hardware memory) and scheduling requirements that support the algorithm and then maps the description into an RTL model whose logic can be synthesized.

Not all algorithms can be implemented in hardware. Nonetheless, this descriptive style is useful and attractive, because it is abstract, and eliminates the need for an a priori architecture. Also, the description can be very readable and understandable. The key distinction to remember is that the assignment statements in a dataflow (RTL) model execute concurrently (in parallel) and operate on explicitly declared registers in the context of a specified architecture; the statements in an algorithmic model execute sequentially, without an explicit architecture.

Example 5.18

By initializing all variables to 0, the algorithm in *Comp_2_ algo* need assign only by exception to the original value, resulting in simplified code. Then the algorithm traverses a decision tree to determine which of the three outputs to assert. The non-asserted outputs will retain the value that was assigned to them at the beginning of the sequence.

```
module Comp_2_algo (output reg A_lt_B, A_gt_B, A_eq_B, input [1: 0] A, B);
 always @ (A, B)        // Level-sensitive behavior
  begin
```

```
        A_lt_B = 0;
        A_gt_B = 0;
        A_eq_B = 0;
        if (A == B)         A_eq_B = 1;   // Note: parentheses are required
        else if (A > B)     A_gt_B = 1;
        else                A_lt_B = 1;
    end
endmodule
```

End of Example 5.18

Figure 5-9 shows the gate-level schematic obtained by synthesizing[15] *Comp_2_algo* and targeting the implementation to generic gates. Note that the Verilog model of the algorithm has register variables to support its execution, but does not need hardware memory because it synthesizes to combinational logic.

5.8.4 Naming Conventions: A Matter of Style

Design teams in industry follow elaborate enterprise-specific rules that govern the style of their Verilog models. This is done to ensure that only constructs supported by synthesis tools are used. Other rules govern the use of upper- and lower-case text, and naming conventions for signals, modules, functions, tasks, and ports, with the aim of increasing the readability and the re-usability of the code [3]. Signals should be given names that describe their use (e.g., *clock*), and modules, functions and tasks should be given names that describe the encapsulated functionality (e.g., *comparator*). The examples in the rest of this book will generally follow a particular port-naming convention.

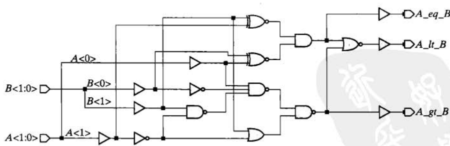


FIGURE 5-9 Synthesis results derived from Comp*Comp_2_algo*.

[15]With Synopsys' Design Compiler. Note that the tool represents vector ranges by the braces < > rather than [ ].

The ports will be ordered in the following sequence: (datapath bidirectional signals, bidirectional control signals, datapath outputs, control outputs, datapath inputs, control inputs, synchronizing and set/reset signals). Likewise, the instance names of modules can be named to clearly identify their functionality and/or location in a design hierarchy. However, for simplicity, we have almost exclusively used brief, non-descriptive instance names.

5.8.5 Simulation with Behavioral Models

An event at the input of a primitive causes the simulator to schedule an updating event for its output. Likewise, an event in the RHS expression of a continuous assignment statement causes the scheduling of an event for the assignment's target variable. In both cases, the scheduling is governed by any propagation delay associated with the primitive or continuous assignment, which has the effect of scheduling the output/target event to occur at a future time step of the simulator, rather than in the current time step. Simulators behave differently, though, when a cyclic behavior is activated. Their associated statements execute sequentially, in the same time step, until the simulator encounters either (1) a delay-control operator (**#**), (2) an event-control operator **@**), (3) a ***wait*** construct, or (4) the last statement of the behavior. The first three have the effect of suspending the execution of the behavioral statement until a condition is satisfied; the last possibility causes the activity to restart from the first statement of the behavior. Models for primitives and continuous assignments cannot suspend themselves. They execute immediately. Cyclic behaviors can suspend themselves. When they do, their activity can cause other behaviors, primitives, and continuous assignments to be activated. But until an active behavior is suspended, the rest of the world waits for it to suspend. A consequence of this is that a cyclic or single-pass behavior that has a loop that does not include a mechanism for suspension of its activity will execute endlessly, and consume the attention of the simulator. Good modeling will prevent this from happening; otherwise, reach for the *off* button.

If multiple behaviors are activated at the same time step, the order in which the simulator executes them is indeterminate. Care must be taken to avoid having such behaviors assign value to the same register, because the outcome of the assignments will be indeterminate. Synthesis tools will warn you of such features in your model.

5.9 Behavioral Models of Multiplexers, Encoders, and Decoders

In Chapter 3, we examined some of the basic building blocks of combinational logic: multiplexers, encoders, and decoders. Here we present their Verilog models to illustrate alternative level-sensitive behavioral descriptions and the results of synthesizing them into an ASIC library.[16]

[16]Styles for writing synthesis-friendly models of combinational and sequential logic will be presented in Chapter 6.

Example 5.19

Mux_4_32_case is a behavioral model of the four-channel, 32-bit, multiplexer shown in Figure 5-10 with a three-state output. The ***default*** case item covers cases that might occur in simulation in four-value logic system, and it is a recommended practice to avoid unintentional synthesis of hardware latches if a case statement is not fully decoded for all possibilities that use 0 and 1. If the case items are not completely decoded, then the default assignment would be treated as a don't-care condition in synthesis and could lead to a smaller circuit.

```
module Mux_4_32_case (
  output      [31: 0]      mux_out,
  input       [31: 0]      data_3, data_2, data_1, data_0,
              [1: 0]       select,
  input                    enable
);
  reg         [31: 0]      mux_int;
  assign mux_out = enable ? mux_int : 32'bz;
  always @ (data_3, data_2, data_1, data_0, select)
    case (select)
      0:                   mux_int = data_0;
      1:                   mux_int = data_1;
      2:                   mux_int = data_2;
      3:                   mux_int = data_3;
      default:             mux_int = 32'bx;    // For simulation
    endcase
endmodule
```

End of Example 5.19

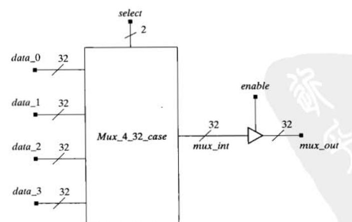


FIGURE 5-10 A four-channel, 32-bit multiplexer.

The Verilog ***case*** statement is similar to its counterpart in other languages (e.g., the *switch* statement in C). It searches from top to bottom to find a match between the case expression and a case item, expressed as a value in Verilog's four-value logic system. The ***case*** statement executes the first match found, and does not consider any remaining possibilities.

The keyword ***always*** in *Mux_4_32_case* declares a behavior, or process (computational activity flow), that begins execution when the event-control expression *(data_3, data_2, data_1, data_0, select)* changes during simulation. The behavior has a simple interpretation: whenever a datapath input or the select bus changes value, decode and update the value of an internal storage variable, *mux_int*. A continuous assignment statement is included in *Mux_4_ 32_case* to describe a three-state output under the active-high control of *enable*.

The **@** operator in *Mux_4_32_case* denotes event control, meaning that the procedural statement(s) that follow the event-control expression do not execute until an activating event occurs. When such an event occurs, the statements execute in sequence, top to bottom. When the last statement completes execution, the computational activity returns to the location of the keyword ***always***, where the event-control operator **@** suspends the behavior until the next sensitizing event occurs. Then the cycle repeats. A cyclic behavior becomes active in simulation at time 0, when the simulation begins, but in this example the activity immediately suspends until the event-control expression changes. Then the ***case*** statement executes, assigns value to *mux_int*, and immediately returns control to the event-control operator.

Example 5.20

An alternative model uses nested conditional statements (***if***) to model a multiplexer. The model *Mux_4_32_if* also includes a continuous assignment that forms a three-state output.

```
module Mux_4_32_if (
   output  [31: 0]   mux_out,
   input   [31: 0]   data_3, data_2, data_1, data_0,
   input   [1: 0]    select,
   input             enable
);
   reg     [31: 0]   mux_int;
   assign mux_out = enable ? mux_int : 32'bz;
   always @ (data_3, data_2, data_1, data_0, select)
      if (select == 0) mux_int = data_0; else
       if (select == 1) mux_int = data_1; else
```

```
        if (select == 2) mux_int = data_2; else
        if (select == 3) mux_int = data_3; else mux_int = 32'bx;
endmodule
```

End of Example 5.20

Example 5.21

Nested conditional assignments, using the *? :* operator, are used in *Mux_4_32_CA* to model the same functionality as *Mux_4_32_if*.

```
module Mux_4_32_CA (
 output      [31: 0]   mux_out,
 input       [31: 0]   data_3, data_2, data_1, data_0,
 input       [1: 0]    select,
 input                 enable
);
 wire        [31: 0]   mux_int;
 assign mux_out = enable ? mux_int : 32'bz;
 assign mux_int = (select == 0) ? data_0 :
                              (select == 1) ? data_1:
                                    (select == 2) ? data_2:
                                          select == 3) ? data_3: 32'bx;
endmodule
```

End of Example 5.21

The combinational encoders and decoders discussed in Chapter 3 can be modeled conveniently with cyclic behaviors.

Example 5.22

Two implementations of an 8:3 encoder are shown below. Neither decodes fully all possible patterns of *Data*, but both cover the remaining outcomes with a ***default*** assignment. The result of synthesis, shown in Figure 5-11, is combinational. This model is intended for applications in which only the indicated words of *Data* occur in operation. The default assignments will be interpreted as don't-cares by a synthesis tool, and are needed to prevent synthesis of a circuit having latched outputs.

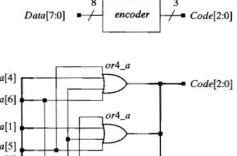


FIGURE 5-11 Result of synthesizing an encoder described by ***if*** statements or a ***case*** statement, as shown in Example 5.22.

```
module encoder (output reg [2: 0] Code, input [7: 0] Data);
  always @ (Data)
  begin
    if (Data == 8'b00000001) Code = 0; else
    if (Data == 8'b00000010) Code = 1; else
    if (Data == 8'b00000100) Code = 2; else
    if (Data == 8'b00001000) Code = 3; else
    if (Data == 8'b00010000) Code = 4; else
    if (Data == 8'b00100000) Code = 5; else
    if (Data == 8'b01000000) Code = 6; else
    if (Data == 8'b10000000) Code = 7; else Code = 3'bx;
  end
/* Alternative description is given below
always @ (Data)
  case (Data)
    8'b00000001   :   Code = 0;
    8'b00000010   :   Code = 1;
    8'b00000100   :   Code = 2;
    8'b00001000   :   Code = 3;
    8'b00010000   :   Code = 4;
    8'b00100000   :   Code = 5;
    8'b01000000   :   Code = 6;
    8'b10000000   :   Code = 7;
    default       :   Code = 3'bx;
  endcase
*/
endmodule
```

End of Example 5.22

Example 5.23

Alternative behaviors describing an 8:3 priority encoder are shown below.[17] The result of synthesizing the circuit is shown in Figure 5-12. Note that the conditional (***if***) statement has an implied priority of execution, and that the ***casex*** statement combined with ***x*** in the case items implies priority also. The ***casex*** statement ignores ***x*** and ***z*** in bits of the *case item* (e.g., *Data[6]*) and the *case expression (Data)*—they are treated as don't-cares. The default assignments in both styles provide flexibility to the logic optimizer of a synthesis tool.

```
module priority (output reg [2: 0] Code, output valid_data, input [7: 0] Data);
  assign valid_data = |Data;                              // "reduction or" operator
  always @ (Data)
    begin
      if (Data[7]) Code = 7; else
      if (Data[6]) Code = 6; else
      if (Data[5]) Code = 5; else
      if (Data[4]) Code = 4; else
      if (Data[3]) Code = 3; else
      if (Data[2]) Code = 2; else
      if (Data[1]) Code = 1; else
                   Code = 3'bx;
    end

/*// Alternative description is given below
always @ (Data)
  casex (Data)
     8'b1xxxxxxx      :   Code = 7;
     8'b01xxxxxx      :   Code = 6;
     8'b001xxxxx      :   Code = 5;
     8'b0001xxxx      :   Code = 4;
     8'b00001xxx      :   Code = 3;
     8'b000001xx      :   Code = 2;
     8'b0000001x      :   Code = 1;
     8'b00000001      :   Code = 0;
     default          :   Code = 3'bx;
  endcase
*/
endmodule
```

End of Example 5.23

[17]The *reduction or* operator is used to form the logic for *valid_data*. The operator forms the ***or*** of the bits in a word.

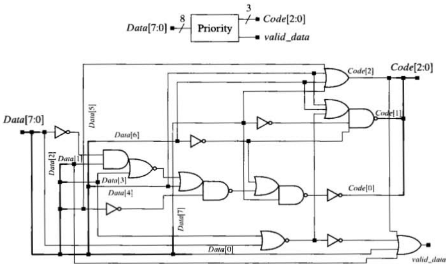


FIGURE 5-12 Block diagram and circuit synthesized for the 8:3 priority encoder described in Example 5.23.

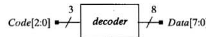
Example 5.24

A 3:8 decoder is described by the alternative behaviors shown below. The decoders synthesize to the circuit in Figure 5-13.

```
module decoder (output reg [7: 0] Data, input [2: 0] Code);
  always @ (Code)
    begin
      if (Code == 0) Data = 8'b00000001; else
      if (Code == 1) Data = 8'b00000010; else
      if (Code == 2) Data = 8'b00000100; else
      if (Code == 3) Data = 8'b00001000; else
      if (Code == 4) Data = 8'b00010000; else
      if (Code == 5) Data = 8'b00100000; else
      if (Code == 6) Data = 8'b01000000; else
      if (Code == 7) Data = 8'b10000000; else
                     Data = 8'bx;
    end
/* Alternative description is given below
always @ (Code)
```

```
case (Code)
   e          :   Data = 8'b00000001;
   1          :   Data = 8'b00000010;
   2          :   Data = 8'b00000100;
   e          :   Data = 8'b00001000;
   4          :   Data = 8'b00010000;
   5          :   Data = 8'b00100000;
   6          :   Data = 8'b01000000;
   7          :   Data = 8'b10000000;
   default    :   Data = 8'bx;
 endcase
*/
endmodule
```

End of Example 5.24



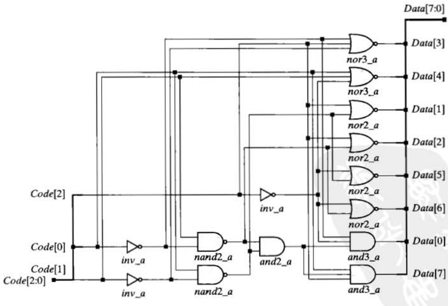


FIGURE 5-13 Block diagram and circuit synthesized from a behavioral model of a 3:8 decoder.

Example 5.25

The seven-segment light-emitting diode (LED) display depicted in Figure 5-14 is a useful circuit in many applications using prototyping boards. Module *Seven_Seg_Display* accepts 4-bit words representing binary-coded decimal (BCD) digits and displays their decimal value. The display has active-low illumination outputs,[18] and can be implemented with combinational logic.[19] The description synthesizes into a combinational circuit. Several of the input codes are unused and should not occur under ordinary operation. One possibility is to assign don't-cares to those codes. However, this would display an output if such an input code occurred. Instead, the default assignment blanks the display for all unused codes. This prevents a bogus display condition, and, as we will see in Chapter 6, prevents the synthesis tool from synthesizing a latched output. Why? If the default assignment is omitted, an event of an input that is not decoded will be detected by the event-control expression of the cyclic behavior, but will not cause *Display* to be an assigned value. The implication is that the output should remain at whatever value it had before the input event occurred; that is, it should behave like a latch! Consequently, the displayed value of *Display* would not correspond to the BCD.

```
module Seven_Seg_Display (output reg [6: 0] Display, input [3: 0] BCD);
  //                                    abc_defg
  parameter          BLANK          = 7'b111_1111;
  parameter          ZERO           = 7'b000_0001;          // h01
  parameter          ONE            = 7'b100_1111;          // h4f
  parameter          TWO            = 7'b001_0010;          // h12
  parameter          THREE          = 7'b000_0110;          // h06
  parameter          FOUR           = 7'b100_1100;          // h4c
  parameter          FIVE           = 7'b010_0100;          // h24
  parameter          SIX            = 7'b010_0000;          // h20
  parameter          SEVEN          = 7'b000_1111;          // h0f
  parameter          EIGHT          = 7'b000_0000;          // h00
  parameter          NINE           = 7'b000_0100;          // h04
  always @ (BCD)
    case (BCD)
     0:              Display = ZERO;
     1:              Display = ONE;
     2:              Display = TWO;
     3:              Display = THREE;
     4:              Display = FOUR;
     5:              Display = FIVE;
     6:              Display = SIX;
```

[18]An active-low signal is asserted if its value is 0.

[19]The underscore character is used in the parameters of *Seven_seg_display* to make the representation of a number more readable.

```
            7               Display = SEVEN;
            8:              Display = EIGHT;
            9:              Display = NINE;
            default:        Display = BLANK;
        endcase
endmodule
```

End of Example 5.25

5.10 Dataflow Models of a Linear-Feedback Shift Register

RTL models are popular in industry because they are easily synthesized by modern tools for electronic design automation (EDA). The next example illustrates an RTL model of a synchronous circuit, an autonomous linear-feedback shift register that executes concurrent transformations on a datapath under the synchronizing control of its only input, a clock signal.

Example 5.26

Linear-feedback shift registers (LFSRs) are commonly used in data-compression circuits implementing a signature analysis technique called *cyclic-redundancy check* (CRC) [4]. Autonomous LFSRs are used in applications requiring pseudo-random binary numbers.[20] For example, an autonomous LFSR can be a random pattern generator providing stimulus patterns to a circuit. The response to these patterns can be

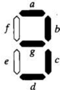


FIGURE 5-14 A seven-segment LED display.

[20]LFSRs are also used as fast counters when only the terminal count is needed.

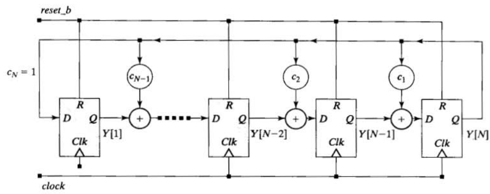


FIGURE 5-15 LFSR with modulo-2 (exclusive-or) addition.

compared to the circuit's expected response and thereby reveal the presence of an internal fault (See built-in self testing in Chapter 11). The autonomous LFSR shown in Figure 5-15 has binary tap coefficients $C_{1,\ldots,} C_N$ that determine whether $Y(N)$ is fed back to a given stage of the register. The structure shown has $C_N = 1$ because $Y[N]$ is connected directly to the input of the leftmost stage. In general, if $C_{N-j+1} = 1$, then the input to stage j is formed as the *exclusive-or* of $Y[j - 1]$ and $Y[N]$, for $j = 2, \ldots N$. Otherwise, the input to stage j is the output of stage $j - 1 - Y[j] <= Y[j - 1]$. The vector of tap coefficients determines the coefficients of the characteristic polynomial of the LFSR, which characterize its cyclic nature [2]. The characteristic polynomial determines the period of the register (the number of cycles before a pattern repeats).

The Verilog code below describes an eight-cell autonomous LFSR with a synchronous (edge-sensitive) cyclic behavior using an RTL style of design. Each bit of the register is assigned a value concurrently with the other bits; the order of the listed non-blocking assignments is of no consequence. The movement of data through the register under simulation is shown in binary and hexadecimal format in Figure 5-16 for the initial state and three cycles of the clock. Note that this model is not fully parameterized, because the register transfers are correct only if $Length = 8$.

```
module Auto_LFSR_RTL #(
  parameter                      Length = 8,
                                 initial_state = 8'b1001_0001,    // 91h
  parameter [ Length: 1]         Tap_Coefficient = 8'b1100_1111
)( input                         clock, reset_b,
  output reg [1: Length]         Y
);

  always @ (posedge clock)
    if (reset_b == 1'b0) Y <= initial_state;        // Active-low reset to initial state
```

```
      else begin
        Y[1] <= Y[8];
        Y[2] <= Tap_Coefficient[7] ? Y[1] ^ Y[8] : Y[1];
        Y[3] <= Tap_Coefficient[6] ? Y[2] ^ Y[8] : Y[2];
        Y[4] <= Tap_Coefficient[5] ? Y[3] ^ Y[8] : Y[3];
        Y[5] <= Tap_Coefficient[4] ? Y[4] ^ Y[8] : Y[4];
        Y[6] <= Tap_Coefficient[3] ? Y[5] ^ Y[8] : Y[5];
        Y[7] <= Tap_Coefficient[2] ? Y[6] ^ Y[8] : Y[6];
        Y[8] <= Tap_Coefficient[1] ? Y[7] ^ Y[8] : Y[7];
      end
endmodule
```

End of Example 5.26

5.11 Modeling Digital Machines with Repetitive Algorithms

An algorithm for modeling the behavior of a digital machine may execute some or all of its steps repeatedly in a given machine cycle, depending on whether the steps execute unconditionally or not. For example, an algorithm that sequentially shifts the bits of an LFSR can be described by a ***for*** loop in Verilog.

Example 5.27

The LFSR in Example 5.26 is modeled again here by *Auto_LFSR_ALGO*, an algorithm-based behavioral model that uses a ***for*** loop to sequence through the concurrent

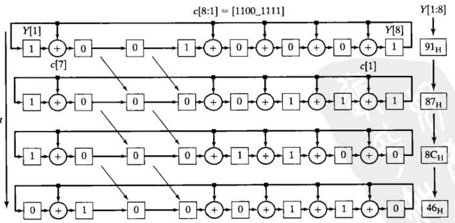


FIGURE 5-16 Data movement in an LFSR with modulo-2 (exclusive-or) addition.

(nonblocking) register assignments one at a time, beginning with the cell to the right of the most significant bit (MSB). The machine's activity in one clock cycle is determined by seven iterations of the loop, followed by a final assignment statement that updates the cell of the MSB. The functionality of this machine is identical to that of the machine in Example 5.26.

```
module Auto_LFSR_ALGO #(parameter
  Length = 8,
  initial_state = 8'b1001_0001,
  parameter [1: Length] Tap_Coefficient = 8'b1111_0011
  ) (
  input              Clock, Reset,
  output reg         [1: Length] Y
);
  integer            Cell_ptr;

  always @ (posedge Clock)
   if (Rst_b == 1'b0) Y <= initial_state;                    // Arbitrary initial state, 91h
    else begin  for (Cell_ptr = 2; Cell_ptr <= Length; Cell_ptr = Cell_ptr +1)
     if (Tap_Coefficient [Length - Cell_ptr + 1] == 1) Y[Cell_ptr] <= Y[Cell_ptr - 1] ^ Y [Length];
     else Y[Cell_ptr] <= Y[Cell_ptr - 1];
     Y[1] <= Y[Length];
  end
endmodule
```

End of Example 5.27

A **for** loop has the form:

```
for (initial_statement; control_expression; index_statement)
statement_for_execution;
```

At the beginning of execution of a ***for*** loop, *initial_statement* executes once, usually to initialize a register variable (i.e., an ***integer*** or a ***reg***) that controls the loop. If *control_expression* is true, the *statement_for_execution* will execute.[21] After the *statement_for_execution* has executed, *the index_statement* will execute (usually to increment a counter). Then the activity flow will return to the beginning of the ***for*** statement and check the value of the *control_expression* again. If *control_expression* is false, the loop terminates and the activity flow proceeds to whatever statement immediately follows the *statement_for_execution*. (*Note*: The value of the register variable governed by *control_expression* in the ***for*** loop may be changed in the body of the loop during execution.)

[21] *Statement_for_execution* can be a single statement or a block statement (i.e., **begin** ... **end**).

Verilog has three additional loop constructs for describing repetitive algorithms: ***repeat, while***, and ***forever***. The ***repeat*** loop (see Example 5.28) executes an associated statement or block of statements a specified number of times. When the activity flow within a behavior reaches the ***repeat*** keyword, an expression is evaluated once to determine the number of times that the statement is to be executed. If the expression evaluates to ***x*** or ***z***, the result will be treated as 0 and the statement will not be executed; that is, the execution skips to the next statement in the behavior. Otherwise, the execution repeats for the specified number of times, unless it is prematurely terminated by a ***disable*** statement within the activity flow (see Example 5.33).

Example 5.28

A ***repeat*** loop is used in the fragment of code below to initialize a memory array.

```
...
word_address = 0;
repeat (memory_size)
  begin
   memory [ word_address] = 0;
   word_address = word_address + 1;
  end
...
```

End of Example 5.28

Example 5.29

In this example, the ***for*** loop is used to assign values to bits within a register after it has been initialized to ***x***. The results of executing the loop are shown in Figure 5-17.

```
reg [15: 0] demo_register;
integer K;

...
for (K = 4; K; K = K - 1)
  begin
   demo_register [K + 10] = 0;
   demo_register [K + 2] = 1;
  end
...
```

At the beginning of execution the statement $K = 4$ executes and assigns the value 4 to K. Thus, the *control_expression*, K, is a "TRUE" value. The assignments to *demo_register* are made and then K is decremented. This process continues until a

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	0	0	0	0	x	x	x	x	1	1	1	1	x	x	x

FIGURE 5-17 Register contents after execution of the ***for*** loop.

decrementation assigns the value $K = 0$. This condition produces a false value for the expression that controls the loop (i.e., K) and control passes to whatever statement follows the ***for*** loop's *statement_for_execution*.

End of Example 5.29

Example 5.30

A majority circuit asserts its output if a majority of the bits of an input word are asserted. The description in *Majority_4b* is suitable for a 4-bit datapath, and uses a ***case*** statement to decode the bit patterns. However, this model is hardwired and becomes cumbersome for long word lengths. A parameterized alternative, *Majority*, uses a ***for*** loop to count the asserted bits in *Data*. A final procedural assignment asserts *Y* after the loop has competed execution, provided that *count* exceeds the value defined by the parameter *majority*. The parameters in *Majority* provide flexibility in sizing *Data* and *count*, and in setting an assertion threshold, *majority*. Figure 5-18 shows a segment of simulation results for *Majority*.

```
module Majority_4b (output reg Y, input A, B, C, D);
  always @ (A, B, C, D) begin
    case ({A, B, C, D})
      7, 11, 13, 14, 15:        Y = 1;
      default                   Y = 0;
    endcase
  end
endmodule

module Majority #(parameter size = 8, max = 3, majority = 5)(
  input          [size-1: 0]    Data
  output reg                    Y
```

```
);
   reg                [max-1: 0]      count;
   integer                            k;
   always @ (Data) begin
     count = 0;
   for (k = 0; k < size; k = k + 1) begin
      if (Data[k] == 1) count = count + 1;
     end
     Y = (count >= majority);
   end
endmodule
```

A Verilog ***while*** loop has the form:

```
while (expression) statement;
```

End of Example 5.30

When the ***while*** statement is encountered during the activity flow of a cyclic or single-pass behavior, *statement*[22] executes repeatedly while a Boolean *expression* is

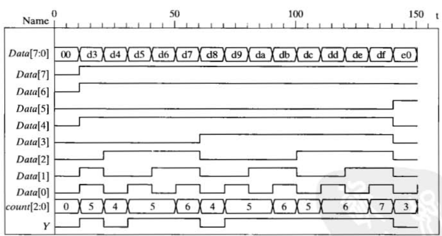


FIGURE 5-18 Simulation results for a parameterized *majority* circuit.

[22]Statement can be a single statement or a block statement (i.e., **begin** . . . **end**).

true. When the *expression* is false the activity flow skips to whatever statement follows *statement*. For example, the statement below increments a synchronous counter while *enable* is asserted.

```
while (enable) begin @ (posedge clock) count <= count + 1; end
```

5.11.1 Intellectual Property Reuse and Parameterized Models

Models have increased value if they are extendable to more than one application. Use parameters to specify bus widths, word length, and other details that customize a model to an application.

Example 5.31

The model *Auto_LFSR_Param* describes the same functionality as *Auto_LFSR_RTL* and *Auto_LFSR_ALGO* (see Example 5.26 and Example 5.27), but uses a parameterized ***for*** loop, conditional operators, and concurrent assignments to the register cells. Unlike *Auto_LFSR_RTL*, which was hard-wired to eight cells, *Auto_LFSR_param* is easily extended to an arbitrary length by changing only its parameters.[23]

```
module Auto_LFSR #(
  parameter                 Length = 8,
  parameter                 initial_state = 8'b1001_0001,    // Arbitrary initial state
  parameter [1: Length]     Tap_Coefficient = 8'b1100_1111
)( input                    Clock, Rst_b,
  output reg [1: Length]    Y
);
  integer                   k;
  always @ (posedge Clock)
     if (Rst_b == 1'b0) Y <= initial_state;
      else begin
       for (k = 2; k <= Length; k = k + 1)
         Y[k] <= Tap_Coefficient[Length-k+1] ? Y[k-1] ^ Y[Length] : Y[k-1];
         Y[1] <= Y[Length];
  end
endmodule
```

End of Example 5.31

[23]For example, a testbench could assign a different initial state.

Example 5.32

The algorithm in the named block *count_of_1s* uses the Verilog right-shift operator (≫) in counting the number of 1s that are within a register, by counting the number of times that a 1 is observed in the least significant bit (LSB) as the word is shifted repeatedly to the right.[24] The right-shift operator fills a 0 in the MSB position that is emptied by the shift operation.

```
begin: count_of_1s          // count_of_1s declares a named block of statements
 reg [7: 0] temp_reg;
 count = 0;

 temp_reg = reg_a;          // load a data word
 while (temp_reg)
  begin
   if (temp_reg[0]) count = count + 1;
    temp_reg = temp_reg >> 1;
  end
end
```

An alternative description simplifies the logic by eliminating the **if** statement, as shown below:

```
begin: count_of_1s
 reg [7: 0] temp_reg;
 count = 0;
 temp_reg = reg_a;          // load a data word
 while (temp_reg)
  begin
   count = count + temp_reg[0];
   temp_reg = temp_reg >> 1;
 end
end
```

The right-shift operation will eventually cause *temp_reg* to have a value of 0, thereby causing the loop to terminate.

End of Example 5.32

[24]In general, the operator can be accompanied by an integer value to shift a word by a specified number of positions, e.g., the statement, *word* <= *word* ≫ 3; will shift ***word*** by three bits to the right, and fill in with 0s from the left. The left-shift operator (≪) has similar effect, but in the reverse direction as the right-shift operator.

5.11.2 Clock Generators

Clock generators are used in testbenches to provide a clock signal for testing the model of a synchronous circuit. A flexible clock generator will be parameterized for a variety of applications. The ***forever*** loop causes unconditional repetitive execution of statements, subject to the ***disable*** statement, and is a convenient construct for describing clocks.

Example 5.33

The code below produces the symmetric waveforms in Figure 5-19 under simulation. The loop mechanism ***forever*** executes until the simulation terminates. This example also illustrates how the activity of an ***initial*** behavior may continue for the duration of a simulation, without expiring. The ***disable*** statement terminates execution after 3500 time steps by disabling the named block *clock_loop*.[25]

```
parameter half_cycle = 50;
parameter stop_time = 350;
initial
  begin: clock_loop         // Note: clock_loop is a named block of statements
    clock = 0;
    forever
      begin
        #half_cycle clock = 1;
        #half_cycle clock = 0;
      end
    end
initial
 #stop_time disable clock_loop;
```

End of Example 5.33

In many situations, loops can be constructed using any of the four basic looping mechanisms of Verilog, but be aware that some EDA synthesis tools will synthesize only the ***for*** loop. Also, note that ***always*** and ***forever*** are not the same construct, though both are associated with cyclic execution. First, the ***always*** construct declares a concurrent behavior. The ***forever*** loop is a computational activity flow and is used only within

[25]In general, a named block may contain local register variables.

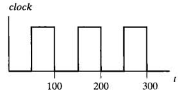


FIGURE 5-19 Clock waveform implemented with a ***forever*** loop.

a behavior. Its execution is not necessarily concurrent with any other activity flow. The second significant distinction is that ***forever*** loops can be nested; on the other hand, cyclic and single-pass behaviors may not be nested. Finally, a ***forever*** loop executes only when it is reached within a sequential activity flow. An ***always*** behavior becomes active and can execute at the beginning of simulation.

The ***disable*** statement is used to prematurely terminate a named block of procedural statements. The effect of executing the ***disable*** is to transfer the activity flow to the statement that immediately follows the named block or task in which ***disable*** was encountered during simulation.

Example 5.34

The *find_first_one* module below finds the location of the first 1 in a 16-bit word. When ***disable*** executes, the activity flow exits the ***for*** loop, proceeds to ***end***, and then returns to the ***always*** to await the next event on *trigger*. At that time *index_value* holds the value at which *A_word* is one. The word is assumed to contain at least one 1.

```
module find_first_one (output reg [3: 0] index_value, input [15: 0] A_word, input trigger);
  always @ (posedge trigger) begin: search_for_1
    for (index_value = 0; index_value < 15; index_value = index_value + 1)
      if (A_word[index_value] == 1) disable search_for_1;
  end
endmodule
```

End of Example 5.34

5.12 Machines with Multicycle Operations

Some digital machines have repetitive operations distributed over multiple clock cycles. This activity can be modeled in Verilog by a synchronous cyclic behavior that has as many nested edge-sensitive event-control expressions as are needed to complete the operations.

Example 5.35

A machine that is to form the sum of four successive samples of a datapath could store the samples in registers and then use multiple adders to form the sum, or it could use one adder to accumulate the sum sequentially. The implicit state machine *add_4cycle* adds four successive samples on a data bus.

```
module add_4cycle (output reg [5: 0] sum, input [3: 0] data, input clk, reset);
  always @ (posedge clk) begin: add_loop
    if (reset == 1'b1) disable add_loop;                        else sum <= data;
     @ (posedge clk) if (reset == 1'b1) disable add_loop;       else sum <= sum + data;
      @ (posedge clk) if (reset == 1'b1) disable add_loop;      else sum <= sum + data;
       @ (posedge clk) if (reset == 1'b1) disable add_loop;     else sum <= sum + data;
  end
endmodule
```

The behavior in *add_4cycle* contains four event-control expressions. The sum is initialized to the first sample of data in the first clock cycle. Four samples of data are accumulated after four clock cycles, before the activity flow returns to the first event-control expression to await a new sequence of samples of data. Note that the ***disable*** statement is included within the reset statement in each clock cycle to ensure that the machine reinitializes properly, regardless of when *reset* is asserted [5]. A hardware realization of *add_4cycle* is shown in Figure 5-20. It synthesizes a state machine[26] to control the four-cycle operation and uses only one adder.

End of Example 5.35

[26]See Appendix H for a description of ASIC flip-flop standard cells used in the examples.

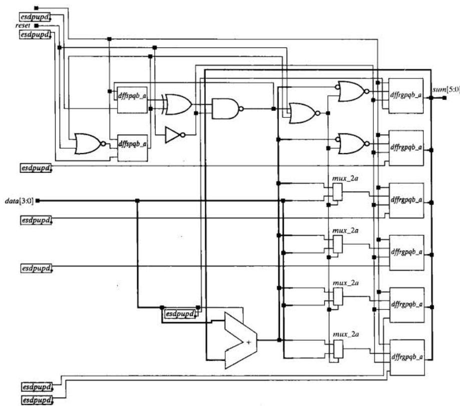


FIGURE 5-20 Circuit synthesized for a four-sample adder.

5.13 Design Documentation with Functions and Tasks: Legacy or Lunacy?

Verilog models are a legacy of their author. Whether a model is useful to anyone else depends on the correctness and clarity of the description. Even a correct model has limited utility if its credibility is compromised by poor documentation and style. Verilog has two types of subprograms that can improve the clarity of a description by encapsulating and organizing code into tasks and functions. Tasks create a hierarchical organization of the procedural statements within a Verilog behavior; functions

substitute for an expression. Tasks and functions let designers manage a smaller segment of code. Both constructs facilitate a readable style of code, with a single identifier conveying the meaning of many lines of code. Encapsulation of Verilog code into tasks or functions hides the details of an implementation from the outside world. Overall, tasks and functions improve the readability, portability, and maintainability of a model.

5.13.1 Tasks

Tasks are declared within a module, and they may be referenced only from within a cyclic or single-pass behavior. A task can have parameters passed to it, and the results of executing the task can be passed back to the environment. When a task is called, copies of the parameters in the environment are associated with the inputs, outputs, and inouts within the task according to the order in which the inputs, outputs, and inouts are declared. The variables in the environment are visible to the task. Additional, local variables may be declared within a task. A word of caution: a task can call itself, but the memory supporting the variables of a task is shared by all calls. The original standard (1995) language does not support recursion, so anticipate side effects.[27]

A task must be named, and may include declarations of any number or combination of the following: ***parameter, input, output, inout, reg, integer, real, time, realtime,*** and ***event***. The variable types ***real, time*** and ***realtime*** are additional members of the register family of types (see Appendix D). The keyword *event* declares an abstract event. Abstract events are used in high-level modeling, but we will not use them in our examples because they are not supported by synthesis tools. All of the declarations of variables are local to the task. The arguments of the task retain the type they hold in the environment that invokes the task. For example, if a wire bus is passed to the task, it may not have its value altered by an assignment statement within the task. All the arguments to the task are passed by a value—not by a pointer to the value. When a task is invoked, its formal and actual arguments are associated in the order in which the task's ports have been declared.

Example 5.36

The module *adder_task* contains a user-defined task that adds two 4-bit words and a carry bit. The circuit produced by the synthesis tool is in Figure 5-21.

```
module adder_task (
  output reg c_out, output reg [3: 0] sum, input [3: 0] data_a, data_b, input c_in, clk, reset
);
```

[27] Verilog-2001 adds *automatic* tasks and functions, which allocate unique storage to each call of a task or function, thereby supporting recursion.

```
always @ (posedge clk, posedge reset)
  if (reset == 1'b1) {c_out, sum} <= 0; else add_values (c_out, sum, data_a, data_b, c_in);

task add_values (
  output c_out, output [3: 0] sum, input [3: 0] data_a, data_b, input c_in
);
  begin
    {c_out, sum} <= data_a + (data_b + c_in);
  end
endtask
endmodule
```

End of Example 5.36

5.13.2 Functions

Verilog functions are declared within a parent module and can be referenced in any valid expression—for example, in the RHS of a continuous assignment statement. A function is implemented by an expression and returns a value at the location of the

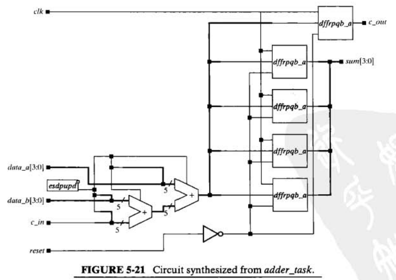


FIGURE 5-21 Circuit synthesized from *adder_task*.

function's identifier. Functions may implement only combinational behavior, that is, they compute a value on the basis of the present value of the parameters that are passed to the function.[28] Consequently, they may not contain timing controls (no delay control (**#**), event control (**@**), or ***wait*** statements), and may not invoke a task. However, they may call other functions, but not recursively.

A function may contain a declaration of inputs and local variables. The value of a function is returned by its name when the expression calling the function is executed. Consequently, a function may not have any declared output or inout port (argument). It must have at least one input argument. The execution, or evaluation, of a function takes place in zero time, that is, in the same time step that the calling expression is evaluated by the host simulator. The definition of a function implicitly defines an internal register variable with the same name, range, and type as the function itself; this variable must be assigned value within the function body.

Example 5.37

The function *aligned_word* in *word_aligner* shifts (<< is the left-shift operator) a word to the left until the most significant bit is a 1. The input to *word_aligner* is an 8-bit word, and the output is also an 8-bit word.

```
module word_aligner #(parameter word_size = 8)(
 output [word_size -1: 0] word_out, input [word_size -1: 0] word_in
);
 assign word_out = aligned_word(word_in);

 function [word_size -1: 0] aligned_word;
  input [word_size -1: 0] word;
  begin
   aligned_word = word;
   if (aligned_word != 0)
    while (aligned_word[word_size -1] == 0) aligned_word = aligned_word << 1;
  end
 endfunction
endmodule
```

End of Example 5.37

[28] A function may not contain a nonblocking assignment.

Example 5.38

The Verilog model *arithmetic_unit* uses functions with descriptive names to make the source code more readable. The combinational circuit synthesized from *arithmetic_unit* is shown in Figure 5-22.

```
module arithmetic_unit (
  output          [4: 0] result_1,
  output          [3: 0] result_2,
  input           [3: 0] operand_1, operand_2
);
  assign result_1 = sum_of_operands (operand_1, operand_2);
  assign result_2 = largest_operand (operand_1, operand_2);
  function [4: 0] sum_of_operands (input [3: 0] operand_1, operand_2);
    sum_of_operands = operand_1 + operand_2;
  endfunction
  function [3: 0] largest_operand (input [3: 0] operand_1, operand_2);
    largest_operand = (operand_1 >= operand_2) ? operand_1 : operand_2;
  endfunction
endmodule
```

End of Example 5.38

Functions and tasks are used both to improve the readability of a Verilog model and to exploit re-usable code. Functions are equivalent to combinational logic, and cannot be used to replace code that contains event-control (@) or delay-control (#) operators. Tasks are more general than functions, and may contain timing controls. Tasks that are to be synthesized may contain event-control operators, but not delay-control operators.

5.14 Algorithmic State Machine Charts for Behavioral Modeling

Many sequential machines implement algorithms (i.e., multistep sequential computations) in hardware. A machine's activity consists of a synchronous sequence of operations on the registers of its datapaths, usually under the direction of a controlling state machine. State-transition graphs (STGs) indicate the transitions that result from inputs that are applied when a state machine is in a particular state, but STGs do not directly display the evolution of states under the application of input data. Fortunately, there is an alternative format for describing a sequential machine.

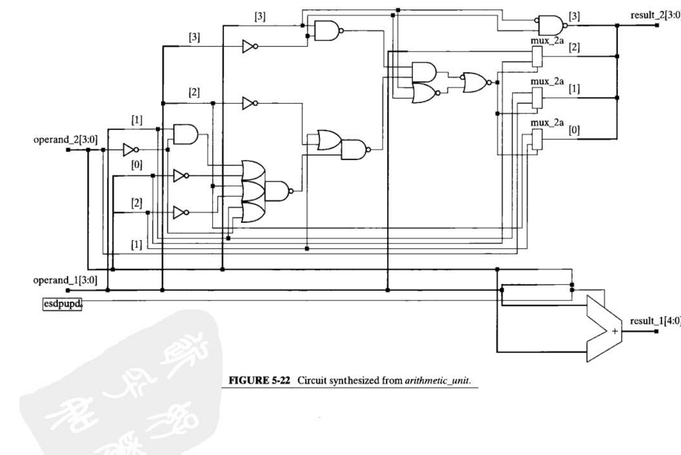


FIGURE 5-22 Circuit synthesized from *arithmetic_unit*.

Algorithmic state machine (ASM) charts are an abstraction of the functionality of a sequential machine, and are a key tool for modeling their behavior [1, 2, 6, 7, 8]. They are similar to software flowcharts, but display the time sequence of computational activity (e.g., register operations) as well as the sequential steps that occur under the influence of the machine's inputs. An ASM chart focuses on the activity of the machine, rather than on the contents of all the storage elements. Sometimes it is more convenient, and even essential, to describe the state of a machine by the activity that unfolds during its operation, rather than by the data that are produced by the machine. For example, instead of describing a 16-bit counter by its contents we can view it as a datapath unit and describe its activity (e.g., counting, waiting, etc.).

ASM charts can be very helpful in describing the behavior of sequential machines and in designing a state machine to control a datapath. We will introduce ASM charts in this chapter and make extensive use of them in designing sequential machines and datapath controllers in Chapter 6 and Chapter 7.

An ASM chart is organized into blocks having an internal structure formed from the three fundamental elements shown in Figure 5-23(a): a state box, a decision box, and a conditional box [2]. State boxes are rectangles, conditional boxes are rectangles with round corners, and decision boxes are diamond-shaped. The basic unit of an ASM chart is an ASM block, shown in Figure 5-23(b). A block contains one state box and an optional configuration of decision diamonds and conditional boxes placed on directed paths leaving the block. An ASM chart is composed of ASM blocks; the state box represents the state of the machine between synchronizing clock events. The blocks of an ASM chart are equivalent to the states of a sequential machine. Given an ASM chart, equivalent information can be expressed by a state-transition graph, but with less clarity about the activity of the machine. An STG uses two symbols, nodes and edges, but an ASM chart uses state boxes, conditional boxes, edges, and decision diamonds. With more symbols, an ASM chart is a higher level of abstraction than an STG, and therefore is more easily understood and used. Another noteworthy comparison is that the order of multiple decision diamonds on a path in an ASM chart implies a precedence of the associated signals and/or conditions. The structure of the chart reveals this detail with clarity.

Both types of state machines (Mealy and Moore) can be represented by ASM charts. The outputs of a Moore-type machine are usually listed inside a state box. The values of the variables in the decision boxes determine the possible paths through the block under the action of the inputs. The ASM chart for a vehicle speed controller (see Figure 5-23(c)) has a Mealy-type output indicating that the tail lights of the vehicle are illuminated while the brake is applied.

Conditional outputs (Mealy outputs) are placed in a conditional box on an ASM chart. These boxes are sometimes annotated with the register operations that occur with the state transition in more general machines that have datapath registers as well as a state register, but we will avoid that practice in favor of the ASMD charts that will be discussed below. The decision boxes along a path in an ASM chart imply a priority decoding of the decision variables. For example, in Figure 5-23(c) the brake has priority over the accelerator. Only paths leading to a change in state are shown, and if a variable does not appear in a decision box on a path leaving a state, then it is understood that the path is independent of the value of the variable. The accelerator is not decoded in state *S_high* in Figure 5-23(c).

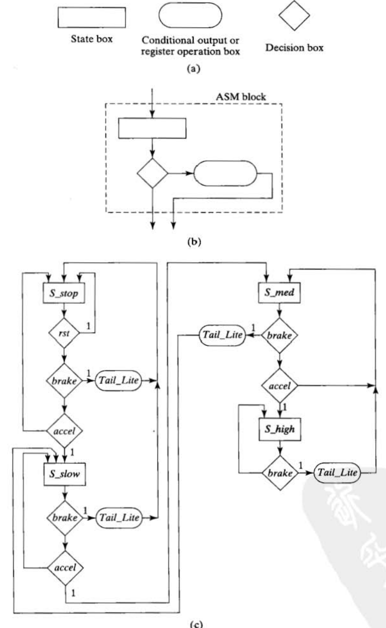


FIGURE 5-23 Algorithmic state machine charts: (a) symbols, (b) an ASM block, and (c) an ASM chart for a vehicle speed controller.

Note: ASM charts can become cluttered, so we sometimes place only the asserted value of a decision variable on the corresponding path and do not label paths with de-asserted decision variables unless the omission would lead to confusion. We also may omit showing default transitions and assertions that return to the same state and paths that return to the reset state when a reset signal is asserted.

5.15 ASMD Charts

One important use of a finite-state machine is to control register operations in a more general sequential machine. For convenience, such machines are partitioned into a controller and a datapath. The controller is described by an ASM chart; the controller's outputs govern the operations of the datapath and/or communicate with its environment. Assertions of the machine's Moore outputs are indicated on the controller's ASM chart in state boxes; Mealy outputs are indicated in conditional boxes. Primary inputs (from the environment) and status inputs (from the datapath) are indicated in decision diamonds. To form an algorithmic state machine and datapath (ASMD) chart, annotate the ASM chart of the controller to indicate the concurrent register operations that occur in the associated datapath unit when the state of the controller makes a transition along the path and the indicated input signal is asserted. A complete ASMD chart contains the annotation describing the datapath operations and the identity of the signals produced by the controller to cause the operations. ASM charts that have been linked to a datapath in this manner are called ASMD charts. The annotation that transforms an ASM chart into an ASMD chart establishes a clear relationship between the controller and the datapath, without confusing the two units.

ASMD charts are motivated by the finite-state machine–datapath paradigm (FSMD) that was introduced as a universal model that represents all hardware design [9]. ASMD charts help clarify the design of a sequential machine by separating the design of its datapath from the design of the controller, while maintaining a clear relationship between the two units. Register operations that occur concurrently with state transitions are annotated on a path of the chart, rather than in conditional boxes on the path, or in state boxes, because these registers are not part of the controller. The outputs generated by the controller are the signals that control the registers of the datapath and cause the corresponding register operations that annotate the ASMD chart.

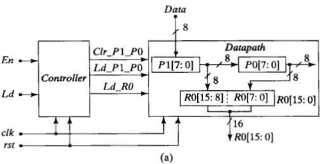
The distinction between primary inputs and status signal inputs from the datapath unit is apparent in the Verilog model of the machine. The primary inputs are inputs at the top level of the machine (i.e., from the environment). The status signals are outputs of the embedded datapath unit and are also inputs to the control unit. This suggests that the model should list ports in the following order: outputs, inputs, clock, reset. Within the list of outputs, vectors are listed before scalars, and primary signals are listed before status signals. This ordering in the Verilog description can be matched by the listings in the block diagram to facilitate communication about the design.

In the examples that follow, we will adhere to a practice of indicating an asynchronous reset signal by a labeled path entering a reset state, but not emanating from another state. An asserted asynchronous reset signal holds the state of the machine in its reset state until the reset condition is de-asserted. A synchronous reset signal will be

denoted by a decision diamond placed on the path leaving the reset state. The diamond will have an exit path that returns to the reset state if the reset signal is asserted. Resets are checked at every state, but their return paths to a reset state will not be shown.

Example 5.39

The internal architecture of the datapath and a block diagram of *pipe_2stage*, a two-stage pipeline that acts as a 2:1 decimator with a parallel input and output, are shown in Fig. 5-24 (a), identifying the interface signals between the controller and the datapath.



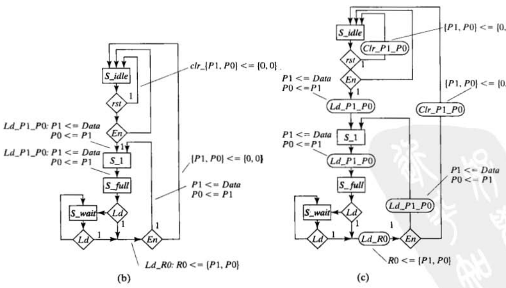


FIGURE 5-24 Two-stage pipeline register: (a) block diagram of pipeline architecture, (b) simplified ASMD chart, and (c) complete ASMD chart.

Decimators are used in digital signal processors to move data from a high-clock-rate datapath to a lower-clock-rate datapath. They are also used to convert data from a parallel format to a serial format. In the example shown here, entire words of data can be transferred into the pipeline at twice the rate at which the content of the pipeline must be dumped into a holding register or consumed by some processor. The content of the holding register, *R*0, can be shifted out serially, to accomplish an overall parallel-to-serial conversion of the data stream.

The simplified ASMD chart in Figure 5-24(b) identifies the output signals of the controller and the datapath operations that they cause to execute. The completed chart in Figure 5-24(c) places the output signals of the controller in conditional output boxes in the ASMD chart.[29] The machine has synchronous reset to *S_idle*, where it waits until *rst* is de-asserted and *En* is asserted. Note that transitions that would occur from the other states to *S_idle* under the action of *rst* are not shown. With *En* asserted, the machine transitions from *S_idle* to *S_1*, accompanied by concurrent register operations that load the MSByte of the pipe with *Data* and move the content of register *P*1 to the LSByte register (*P*0). At the next clock the state goes to *S_full*, and now the pipe is full. If *Ld* is asserted at the next clock, the machine moves to *S_1* while dumping the pipe into a holding register *R*0. If *Ld* is not asserted, the machine enters *S_wait* and remains there until *Ld* is asserted, at which time it dumps the pipe and returns to *S_1* or to *S_idle*, depending on whether *En* is asserted too. The data rate at *R*0 is one-half the rate at which data bytes are supplied to the unit from an external datapath. The signals generated by the control unit to control the indicated register operations are named and added to the chart to obtain the complete ASMD chart shown in Fig. 5-24(c). The chart completely describes the machine by providing a description of the datapath operations and the finite-state machine that controls those operations. The Verilog model of the controller given below is described by two cyclic behaviors. The first, an edge-sensitive behavior, merely synchronizes the state transitions and responds to a reset. The second, a level-sensitive cyclic behavior, describes the combinational logic forming the next state and the outputs of the machine. Note that the reset action is described in the edge-sensitive behavior and is not part of the combinational logic. There are various ways to describe finite-state machines in Verilog, but this style is preferred because it is clear and synthesizes readily.

```
module Controller (output reg Clr_P1_P0, Ld_P1_P0, Ld_R0, input En, Ld, clk, rst);
 parameter S_idle = 2'b00, S_1 = 2'b01, S_full = 2'b10, S_wait = 2'b11;
 reg [1: 0] state, next_state;

 always @ (posedge clk)
   if (rst) state <= S_idle;
   else state <= next_state;

 always @ (state, En, Ld) begin
   Clr_P1_P0 = 0;
   Ld_P1_P0 = 0;
   Ld_R0 = 0;
```

[29]Unconditional outputs (i.e., Moore outputs) are annotated within a state box.

```
case (state)
 S_idle:      if (En) begin next_state = S_1; Ld_P1_P0 = 1; end
              else next_state = S_idle;

 S_1:         begin next_state = S_full; Ld_P1_P0 = 1; end

 S_full:      if (!Ld) begin next_state = S_wait; Ld_R0 = 1; end
              else if (En) begin next_state = S_1; Ld_P1_P0 = 1; end
              else begin next_state = S_idle; Clr_P1_P0 = 1; end

 S_wait:      if (!Ld) next_state = S_wait;
              else begin
               Ld_R0 = 1;
               if (En) begin Ld_P1_P0 = 1; next_state = S_1; end
               else begin next_state = S_idle; Clr_P1_P0 = 1; end
              end
 endcase
end
endmodule
```

End of Example 5.39

The design of a datapath controller (1) begins with an understanding of the sequential register operations that must execute on a given datapath architecture, (2) defines an ASM chart describing a state machine that is controlled by primary input signals and/or status signals from the datapath, (3) forms an ASMD chart by annotating the arcs of the ASM chart with the datapath operations associated with the state transitions of the controller, (4) annotates the state of the controller with unconditional output signals, and (5) includes conditional boxes for the signals that are generated by the controller to control the datapath. If signals report the status of the datapath to the controller, these are placed in decision diamonds too, to indicate that there is feedback linkage between the machines. This decomposition of effort leads to separately verifiable models for the controller and the datapath. The final step in the design process is to integrate the verified models within a parent module and to verify the functionality of the overall machine. We will consider this methodology and the role of status signals fed back to the control unit from the datapath in more detail in Chapter 7.

The register operations of ASMD charts are usually written in register transfer notation (RTN), a set of symbols and semantics that compactly specifies the instruction set of a computer [2, 6, 7]. We will describe those operations by Verilog's operators, which correspond to common hardware operations, and will annotate the ASMD chart accordingly. Note the concatenation and nonblocking assignment operators in Figure 5-24. Datapath register operations made with a nonblocking assignment operator are concurrent, so the register transfers denoted by $R0 <= \{P1, P0\}$ and $\{P1, P0\} <= 0$ are concurrent and do not race.

5.16 Behavioral Models of Counters, Shift Registers, and Register Files

Counters, shift registers, and register files are important datapath units and are used in many digital machines. The storage elements of counters and registers usually have the same synchronizing and control signals.[30] A counter generates a sequence of related binary words; a register stores data that can be retrieved and/or overwritten under the control of a host processor. The cells of a shift register exchange contents in a systematic and synchronous manner. Register files are a collection of registers that share the same synchronizing and control signals. Behavioral descriptions of a wide variety of counters, shift registers, and register files are routinely synthesized by modern synthesis tools. The descriptions specify synchronous register operations under the control of external input signals (presumably generated as the output signals of a control unit). The datapath and control units can be designed and verified separately, before verifying their integrated operation.

5.16.1 Counters

Example 5.40

Consider a 4-bit counter which can count up, count down, or hold the count. The counter could be modeled as a state machine by choosing a state consisting of the content of the register holding the count, but instead we choose to associate the state with the activity of the machine, which consists of idling, incrementing, or decrementing, rather than associating the state with the data that result from the machine's activity. This approach treats the counter as a datapath unit, rather than as a finite-state machine. Associating the state of the datapath unit with its activity allows us to model the counter independently of its word length and its data. It also reduces the machine to one having a single state, which is simply a single-cycle datapath unit. Implicitly, such a machine is controlled by an external agent. For example, the counter described here can be controlled by a finite-state machine having a 2-bit input word, *up_dwn*, which controls the activity of the datapath unit. Depending on *up_dwn*, the machine has options to count up, count down, or hold the count. The finite-state machine could also include an active-low asynchronous reset of the counter. Figure 5-25(a) shows a block diagram and a partial ASMD chart for the counter. The concurrent register operations that are linked to the state transitions of the control unit can be indicated by annotating the ASM chart of Figure 5-25(a) to produce the completed ASMD chart in Figure 5-25(b).

The activity of the counter has three states: idling (*S_idle*), incrementing (*S_incr*), and decrementing (*S_decr*). The asynchronous active-low reset signal *reset_* drives the state to *S_idle* and its action is not confined to the active edges of the clock. The signal *reset_* is shown only at *S_idle*, to indicate that *S_idle* is reached from any state when

[30] An exception is a ripple counter, which connects the output of a stage to the clock input of an adjacent stage [5].

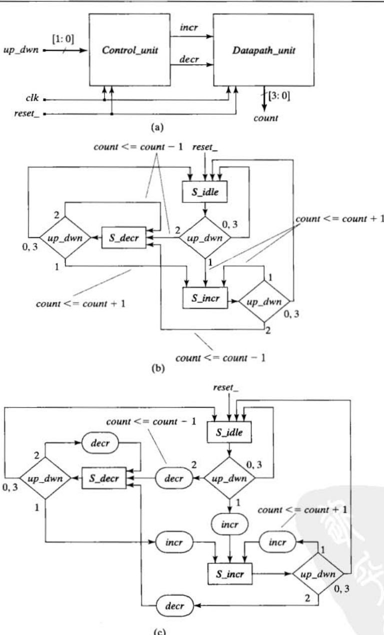


FIGURE 5-25 ASM and ASMD charts for a behavioral model of an up–down counter having synchronous reset: (a) without conditional output boxes, and (b) with conditional output boxes for register operations outputs generated by the state machine.

*reset_* is asserted. The machine enters *S_idle* asynchronously from any state under the action of *reset_* and enters synchronously from *S_decr* and *S_incr* if *up_dwn* is 0 or 3. Otherwise, the count is either incremented or decremented.

Note that the charts in Figure 5-25 are independent of the word length of the counter and that they capture the functionality of the machine. They can be adapted to a variety of applications.

A controller and datapath implementation of a sequential machine based on Figure 5-25 would require a 4-bit register to hold *count* and a separate 2-bit register to hold the state. A closer look at the machine suggests even further simplification. The counter can be viewed as having a single (equivalent) state, *S_running*, and there is no need for a state register, only a datapath register for *count*. Two ASMD charts are shown in Figure 5-26, one (a) for a machine with asynchronous reset, and the other (b) for a machine with synchronous reset. The action of *reset_* is to drive the state to *S_running* and flush the register holding *count*. *reset_* is shown as a synchronous entry into *S_running* in Figure 5-26(a). A decision diamond for *reset_* is shown in Figure 5-26(b) on the path leaving *S_running* to remind us that the machine ignores *up_dwn* if *reset_* is asserted. The relative location of the decision diamonds indicates that *reset_* has priority.

The Verilog model of the counter can be derived from the ASMD chart by noting that at every clock edge the machine either increments the count, decrements the count, or leaves the count unchanged. The cyclic behavior shown in *Up_Down_Implicit1* describes the decision tree for the state changes and the operations on the datapath register. It suppresses the details of the control signals that will control the hardware datapath. In contrast to an explicit enumeration of states in the controller in Example 5.39, the state machine in

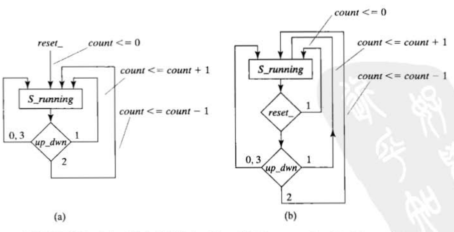


FIGURE 5-26 A simplified ASMD chart for a 4-bit binary counter: (a) with asynchronous active-low reset, and (b) synchronous active-low reset.

this example is implicit. This style of description is not recommended for large machines having more complex interaction between the control unit and the datapath unit.[31]

```
module Up_Down_Implicit1 (output reg [2: 0] count, input up_dwn, clock, reset_);
 always @ (negedge clock, negedge reset_)
  if (reset_ == 1'b0) count <= 3'b0; else
   if (up_dwn == 2'b00 || up_dwn == 2'b11) count <= count; else
    if (up_dwn == 2'b01) count <= count + 1; else
     if (up_dwn == 2'b10) count <= count -1;
endmodule
```

End of Example 5.40

Example 5.41

A ring counter asserts a single bit that circulates through the counter in a synchronous manner. The movement of data in an 8-bit ring counter is illustrated in Figure 5-27. Given an external synchronizing signal, *clock*, the behavior described by *ring_counter* ensures the synchronous movement of the asserted bit through the register and the automatic restarting of the count at *count[0]* after *count[7]* is asserted at the end of a cycle. Note that the activity of the machine is the same in every clock cycle and that *ring_counter* is an implicit state machine. The synthesized circuit is shown in Figure 5-28.

count [7:0]

0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0
0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1

t

FIGURE 5-27 Data movement in an 8-bit ring counter.

[31]The limitations and utility of implicit state machines will be considered in Chapter 6.

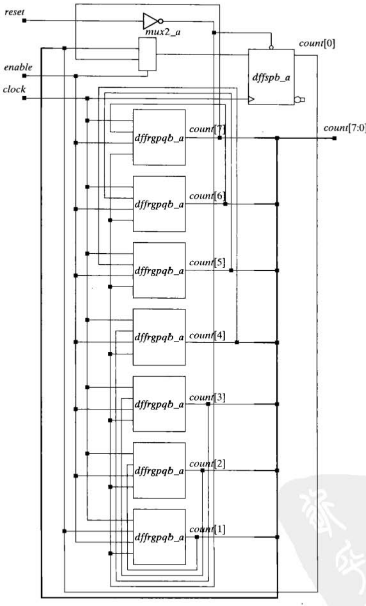


FIGURE 5-28 Ring counter synthesized from a Verilog behavioral description (asynchronous reset).

The D-type flip-flops in the implementation are active on the rising edge of the clock, have gated data (i.e., the datapath of the flip-flop is driven by the output of a multiplexer whose control signal selects between the output of the flip-flop and the external datapath), and have an asynchronous active-high reset.

```
module ring_counter #(parameter word_size = 8)(
  output reg [word_size -1: 0] count, input enable, clock, reset
);
  always @ (posedge clock, posedge reset)
    if (reset) count <= {{(word_size -1){1'b0}}, 1'b1};
    else if (enable == 1'b1) count <= {count[word_size -2: 0], count[word_size -1]};
endmodule
```

End of Example 5.41

Example 5.42

Our last example of a counter is a 3-bit up–down counter, but modified to include two additional features: a signal, *counter_on*, which enables the counter, and a signal, *load*, which loads an initial count from an external datapath. The description of the counter exploits Verilog's built-in arithmetic and implements the counter with an ***if*** statement. The synthesized circuit and block diagram are shown in Figure 5-29. In this implementation, the library cell *dffrgpqb_a* is a D-type flip-flop active on the rising edge, having internally gated data[32] and asynchronous active-low reset.

```
module up_down_counter (
  output reg [2: 0]      Count,
  input                  load, count_up, counter_on, clk, reset,
  input    [2: 0]        Data_in
);
  always @ (posedge clk, posedge reset)
    if (reset == 1'b1)              Count <= 3'b0; else
      if (load == 1'b1)             Count <= Data_in; else
        if (counter_on == 1'b1) begin
          if (count_up == 1'b1)     Count <= Count + 1;
            else                    Count <= Count - 1;
        end
endmodule
```

End of Example 5.42

[32]Cell libraries include such flip-flops because the physical layout of the mask for the integrated unit requires less area than connected, but distinct units that accomplish the same functionality. The integrated unit will also have superior performance (smaller input–output propagation delays).

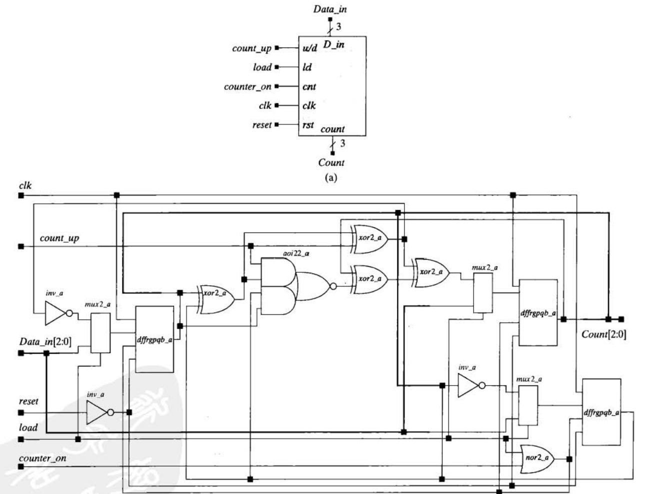


FIGURE 5-29 3-bit up–down counter with additional features that load an initial count and enable the counting activity: (a) block diagram symbol, and (b) the circuit synthesized for the counter.

5.16.2 Shift Registers

Example 5.43

Shift_reg4 below declares an internal 4-bit register, *Data_reg*, which creates *Data_out* by a continuous assignment to the LSB of the register and forms the register contents synchronously from a concatenation of the scalar *Data_in* with the three leftmost bits of the register. Notice that the register variable, *Data_reg*, is referenced by concatenation ({ }) in a nonblocking assignment before it is assigned value in a synchronous behavior. This implies the need for memory, and synthesizes to the flip-flop structure shown in Figure 5-30. Also, recall that the values on the RHS of the nonblocking assignments are the values of the variables immediately before the active edge of the clock, and the values on the LHS are the values formed after the edge.

```
module Shift_reg4 #( parameter word_size = 4)(
  output Data_out,
  input  Data_in, clock, reset
);
  reg [word_size -1: 0] Data_reg;
  assign Data_out = Data_reg[0];
  always @ (posedge clock, negedge reset)
    begin
      if (reset == 1'b0) Data_reg <= {word_size {1'b0}};
      else               Data_reg <= {Data_in, Data_reg[word_size -1: 1]};
    end
endmodule
```

End of Example 5.43

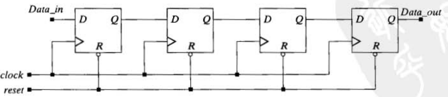


FIGURE 5-30 4-bit shift register synthesized from a Verilog behavior.

Example 5.44

In this example, a register with reset and parallel load is synthesized from the Verilog description of *Par_load_reg4*. The structure of the synthesized result is shown in Figure 5-31. The muxes and flip-flops are implemented as library cells.

```
module Par_load_reg4 #(parameter word_size = 4)(
  output reg     [word_size -1: 0]  Data_out,
  input          [word_size -1: 0]  Data_in,
  input                             load, clock, reset
);
  always @ (posedge clock, posedge reset)
    begin
      if (reset == 1'b1)       Data_out <= {word_size {1'b0}};
      else if (load == 1'b1)   Data_out <= Data_in;
    end
endmodule
```

End of Example 5.44

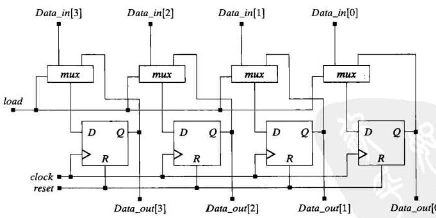


FIGURE 5-31 4-bit register with parallel load, synthesized from a Verilog behavior.

Example 5.45

Barrel shifters are used in digital signal processors to avoid overflow problems by scaling the input and output of a datapath operation. Scaling is accomplished by shifting the bits of a word to the left or to the right. Shifting a word to the right effectively divides the word by a power of 2, and shifting the word to the left multiplies the word by a power of 2. Words are shifted to the right to prevent overflow that might result from an arithmetic operation, and then the final result is shifted to the left. The shifting action of a barrel shifter can be implemented with combinational logic, but the model presented below uses registered logic and circulates the word through a storage register, by exploiting concatenation, as depicted in Figure 5-32. The top register shows the pattern before the shift, and the bottom register shows the pattern that results from the shift. The circuit synthesized from *barrel_shifter* is shown in Figure 5-32. A more general barrel shifter can shift by a specified number of bits.

```
module barrel_shifter #( parameter word_size = 8)(
 output reg            [word_size -1: 0]  Data_out,
 input                 [word_size -1: 0]  Data_in,
 input                                    load, clock, reset
);

 always @ (posedge clock, posedge reset)
  begin
   if (reset == 1'b1)      Data_out <= {word_size {1'b0}};
   else if (load == 1'b1)  Data_out <= Data_in;
   else                    Data_out <= {Data_out[word_size -2: 0],
                                      Data_out[word_size -1]};

  end
endmodule
```

End of Example 5.45

Example 5.46

A 4-bit universal shift register is an important unit of digital machines that employ a bit-slice architecture, with multiple identical slices of a 4-bit shift register chained together with additional logic to form a wider and more versatile datapath [8]. Its features include synchronous reset, parallel inputs, parallel outputs, bidirectional serial input from either the LSB or the MSB, and bidirectional serial output to either the LSB or the MSB. In the serial-in, serial-out mode the machine can delay an input signal for 4 clock ticks, and act as a unidirectional shift register. In parallel-in, serial-out mode it operates as a parallel-to-serial converter, and in the serial-in, parallel-out mode it operates as a serial-to-parallel converter. Its parallel-in, parallel-out mode, combined

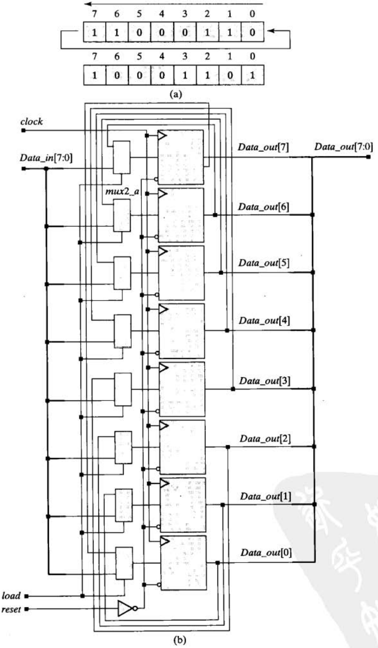


FIGURE 5-32 8-bit barrel shifter with registered output: (a) data movement, and (b) circuit synthesized from a Verilog behavioral model, *barrel shifter*.

with shift operations, allows it to perform any of the operations of less versatile unidirectional shift registers.

```
module Universal_Shift_Reg #(parameter word_size = 4)(
  output reg[word_size -1: 0]    Data_Out,
  output                         MSB_Out, LSB_Out,
  input     [word_size -1: 0]    Data_In,
  input                          MSB_In, LSB_In,
  input                          s1, s0, clk, rst
);

  assign MSB_Out = Data_Out[word_size -1];
  assign LSB_Out = Data_Out[0];

  always @ (posedge clk) begin
   if (rst == 1'b1) Data_Out <= 0;
   else case ({s1, s0})
    0:   Data_Out <= Data_Out;                                 // Hold
    1:   Data_Out <= {MSB_In, Data_Out[word_size -1: 1]};  // Serial shift from MSB
    2:   Data_Out <= {Data_Out[word_size -2: 0], LSB_In};  // Serial shift from LSB
    3:   Data_Out <= Data_In;                                  // Parallel Load
   endcase
  end
endmodule
```

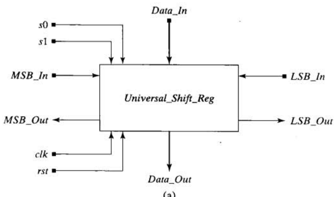
We can anticipate that the gate-level machine will consist of four D-type flip-flops with steering logic to manage the datapaths supporting the specified features. The block diagram symbol and simulation results verifying the functionality of the model are shown in Figure 5-33. The waveforms for *Data_Out* illustrate the right-shift, left-shift, and load operations. For example, when (s1, s0) = (1, 0) the machines shifts bits from the LSB towards the MSB.

End of Example 5.46

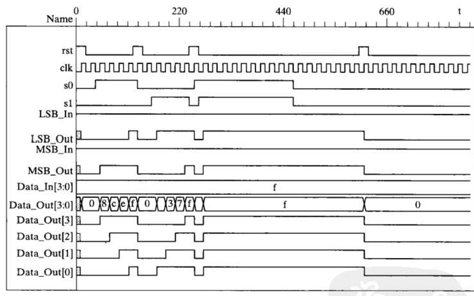
5.16.3 Register Files and Arrays of Registers (Memories)

Usually implemented by D-type flip-flops, register files are not used for mass storage because they occupy significantly more silicon area than compiled memory. A common application combines a register file in tandem with an ALU, as shown in Figure 5-34. The dual-channel outputs of the register file form the datapaths to the ALU, and the output of the ALU is stored in the register file at a designated location. A host processor provides the addresses for the operations and controls the sequence of reading and writing to prevent a simultaneous read and write affecting the same location.[33]

[33]More complex register files have logic that allows a read operation to return the value currently written.



(a)



(b)

FIGURE 5-33 4-bit universal shift register: (a) block diagram symbol and (b) simulation results verifying the 4-bit universal shift register.

Example 5.47

The single-input, dual-output register file modeled on the next page as an implicit state machine by *Register_File* introduces the concept of a Verilog memory, a provision for declaring an array of words.[34] By appending an additional array range ([31:0]) to the

[34]A word in a Verilog memory (i.e., an array of words) can be addressed directly. A cell (bit) in a word can be addressed indirectly by first loading the word into a buffer register and then addressing the bit of the word.

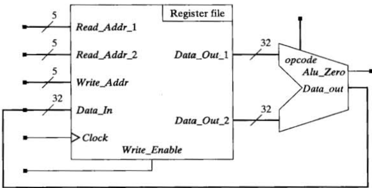


FIGURE 5-34 A 32-word register file in tandem with an ALU with a 32-bit datapath.

declaration of *Reg_File* in the module *Register_File*, we declare 32 words of memory, with each word having 32 bits. The dual-output datapaths are formed by continuous assignment statements using the 5-bit address provided by the host processor. The writing operation occurs synchronously under the control of *Write_Enable*, and the processor must ensure that data is not read while *Write_Enable* is asserted and *Clock* has a rising edge. Decoders are synthesized automatically by a synthesis tool, and are inside the register file, where they decode the addresses to locate a specific register.

```
module Register_File #(parameter word_size = 32, addr_size = 5)(
 output   [word_size -1: 0]   Data_Out_1, Data_Out_2,
 input    [word_size -1: 0]   Data_in,
 input    [addr_size -1: 0]   Read_Addr_1, Read_Addr_2, Write_Addr,
 input                        Write_Enable, Clock
);
 reg      [word_size -1: 0]   Reg_File [31: 0];   // 32bit x32 word memory declaration

 assign Data_Out_1 = Reg_File[Read_Addr_1];
 assign Data_Out_2 = Reg_File[Read_Addr_2];

 always @ (posedge Clock) begin
  if (Write_Enable == 1'b1) Reg_File [Write_Addr] <= Data_in;
 end
endmodule
```

End of Example 5.47

5.17 Switch Debounce, Metastability, and Synchronizers for Asynchronous Signals

Sequential circuits use flip-flops and latches as storage elements, but both devices are subject to a condition called *metastability*. A hardware latch can enter the metastable state if a pulse at one of its inputs is too short, or if both inputs are asserted either

simultaneously or within a sufficiently small interval of each other. A transparent latch can go into a metastable state too, if the data are unstable around the edge of the *enable* input. D-type flip-flops (edge-triggered) are formed from two cascaded transparent latches with complementary clocks. A D-type flip-flop can enter the metastable state if the data are unstable in the setup interval preceding the clock edge or if the clock pulse is too narrow. Given the vulnerability of storage devices to metastability, it is important that systems be designed to minimize the impact of signals that could cause the system to be upset by this condition.

Many physical systems that are intended to operate synchronously have asynchronous input signals. A signal is asynchronous if it is not controlled by a clock or if it is synchronized by a clock in a different domain. In both cases, a signal transition can occur in a random manner with respect to the active edge of the clock that is controlling sequential devices. Traffic lights, computer keyboards, and elevator buttons have inputs that arrive randomly. If they happen to arrive during the setup interval of a flip-flop they could cause the flip-flop to enter a metastable state and remain there for an indefinite time, upsetting the operation of the system.

If a mechanical switch generates an input that drives a flip-flop of a circuit, the input signal could oscillate during the setup interval of the flip-flop and cause it to enter a metastable state [7–10]. Figure 5-35 illustrates a simple push-button switch configuration in which the data line to the flip-flop is normally pulled down. When the spring-loaded button is pushed down, a connection is made to pull the line up to Vdd. The mechanical contact will vibrate momentarily, for a few milliseconds, creating an unstable signal on the line. There are various ways to deal with switch bounce, depending on the application. For example, the push-button switches on a typical student prototyping board for an FPGA have a resistor-capacitor (RC) lowpass filter and a buffer placed between the switch and the chip [11].

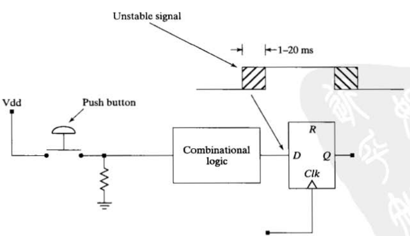


FIGURE 5-35 A push-button input device with closure bounce.

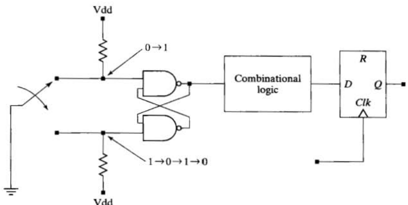


FIGURE 5-36 A NAND latch configuration for eliminating the effects of switch closure bounce.

As an alternative remedy, the circuit shown in Figure 5-36 uses a single pole-double throw switch to eliminate the effect of the bounce. With the switch initially in the upper position, the line driving the upper NAND gate is pulled down to ground. The connection between the arm of the switch and the circuit is broken momentarily when the arm moves from the top position to the bottom contact. The arm of the switch still bounces when it arrives at the bottom contact, but as long as it does not bounce back to the top contact, the signal at the bottom input to the NAND latch can oscillate and not affect the circuit because the top output of the latch has already made a transition from 1 to 0, thereby blocking the activity at the bottom switch contact from affecting the circuit. A similar action occurs when the switch is thrown to the top position. Keyboards commonly have debounce circuitry built into the keys.

A flip-flop may enter a metastable state if the data input changes within a finite interval before or after the clock transition. The output of the device has an output between a 0 and 1, and cannot be decoded with certainty. The physical situation is illustrated in Figure 5-37, where a ball must roll over a pinnacle before making a state transition. If it should be pushed with enough energy to only reach, but not pass, the pinnacle, it would reside there indefinitely. A circuit in the metastable state will remain

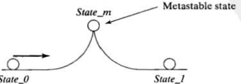


FIGURE 5-37 An illustration of how metastability can happen in a physical system.

there for an unpredictable time before returning to the state it had before the clock, or making a transition to the opposite state. Asynchronous inputs are problematic, because their transitions are unpredictable. Moreover, a flip-flop that enters a metastable state may still be in that state when its data is to be sampled, thereby propagating erroneous data to receivers.

If inputs are allowed to be asynchronous, metastability cannot be prevented, but its effect can be reduced. Experimental results have shown that the mean time between failures of a circuit with an asynchronous input is exponentially related to the length of time available for recovery from the metastable condition. Thus, high-speed digital circuits rely on synchronizers to create a time buffer for recovering from a metastable event, thereby reducing the possibility that metastability will cause a circuit to malfunction.[35]

The first rule of synchronization is that an asynchronous signal should never be synchronized by more than one synchronizer. To do so would risk having the outputs of multiple synchronizers produce different synchronized signals in the event that one or more of them is driven into the metastable condition.

There are two basic types of synchronizer circuits, depending on whether the asynchronous input pulse has a width that is larger or smaller than the period of the clock. In the former case, a synchronizer consists of a multistage shift register placed between the asynchronous input and the circuit. Multiple stages are used because the clock periods are ever-shrinking as technology advances, making it more likely that the metastability of a single flip-flop might not be resolved in a single period. The circuit in Figure 5-38(a) treats the situation in which the width of the asynchronous input pulse is larger than the clock period. Two flip-flops are placed between *Asynch_in* and the circuit; the second flip-flop operates synchronously, and the flip-flop driven by *Asynch_in* guards against metastability. Consider how the circuit operates: if the asynchronous input signal reaches a stable condition outside the setup interval, it will be clocked through with a latency of two cycles. On the other hand, if *Asynch_in* is unstable during the setup interval (due to bounce or to a late-arriving input) there are two possibilities. Assuming that the circuit was in a reset condition, if the unstable input is sampled as a 0, but ultimately settles to a 1, the 1 will appear at the output with a latency of three cycles. If the signal settles to 0, it will have arrived with a latency of two cycles. Thus, the maximum latency is $n + 1$, where n is the number of stages in the synchronizer chain. The second flip-flop does not see the instability; the only effect is at the entry stage, which could lead to one additional cycle of latency. Latency is tolerable because the asynchronous input itself does not have a predictable arrival time; but ambiguous output transition times resulting from metastability are problematic because they can cause upset of the system. Additional stages of flip-flops may be used if the condition of metastability does not resolve within a clock cycle.

If the width of the asynchronous input pulse may be less than the period of the clock, the circuit in Figure 5-38(b) is used, with an additional cost in hardware. Note that the first flip-flop has *Vcc* connected to its data input and has *Asynch_in* connected

---

[35]For an extensive treatment of synchronizers, see Reference 8.

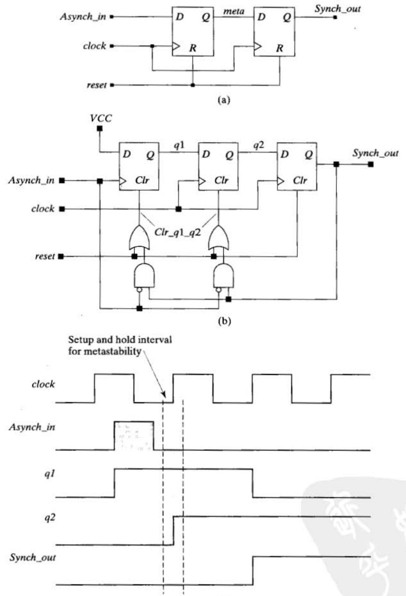


FIGURE 5-38 Synchronizer circuits for asynchronous input signals: (a) circuit for use when the width of the asynchronous input pulse is greater than the period of the clock, (b) circuit for use when the width of the asynchronous input pulse is less than the period of the clock, (c) waveforms in the circuit of (b) when the asynchronous pulse does not cause a metastable condition, and (d) waveforms of the circuit in (b) when the asynchronous input signal causes a metastable condition.

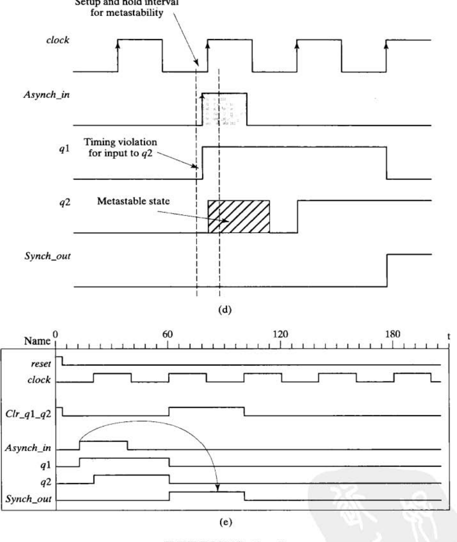


FIGURE 5-38 (Continued)

to its clock input, and that the remaining two flip-flops are triggered by the clock of the system, i.e., the clock to which *Asynch_in* will be synchronized. A short pulse at *Asynch_in* will drive $q1$ to 1; this value will propagate to *Synch_out* after the next two clock edges. When *Synch_out* becomes 1, the signal at *Clr_q1_q2* is asserted, assuming that *Asynch_in* returns to 0 before *Synch_out* becomes 1. Clearing $q1$ and $q2$ constrains the length of the synchronized pulse to be one cycle of *clk*; otherwise, if only $q1$ is

cleared the length of the pulse would be two cycles of *clk*. The flip-flop driving *q2* may experience metastability, but it is assumed to end and have no effect on the flip-flop driving *Synch_out*. In Figure 5-38(c), the pulse of *Asynch_in* does not cause metastability in the circuit of Figure 5-38(b), but the pulse in Figure 5-38(d) causes metastability. Figure 5-38(e) shows waveforms from a simulation of the circuit in Figure 5-38 (b). Note that *q1* is asserted when *Asynch_in* is asserted, and that *q2* is asserted at the first edge after assertion of *Asynch_in*; and that *Synch_out* is asserted at the second edge after *Asynch_in* is asserted. Also, *Synch_out* persists for exactly one cycle of clock.

Synchronizers are also used when a signal must cross a boundary between two clock domains (For a detailed discussion of synchronization across clock domains, see Example 9.11). If *clock_1* in Figure 5-39 is slower than *clock_2*, the synchronizer in Figure 5-38(a) should be used to synchronize the interface signals controlling data transfer between the domains, otherwise the synchronizer in Figure 5-38(b) should be used. Care must be taken to anticipate that more than one active edge of *clock_2* will occur while *asynch_in* is asserted if *asynch_in* is synchronized to *clock_1*, that is, it has duration $T_{clock_1} > T_{clock_2}$.[36]

5.18 Design Example: Keypad Scanner and Encoder

Keypad scanners are used to enter data manually in digital telephones, computer keyboards, and other digital systems. Cell phones, computers, automatic tellers, fuel dispensers, personal media devices, and other digital-based devices have a keypad. A keypad scanner responds to a pressed key and forms a code that uniquely identifies the key that is pressed. It must take into account the asynchronous nature of the inputs and deal with switch debounce. Also, it must not interpret a key to be pressed repeatedly if it is pressed once and held down.

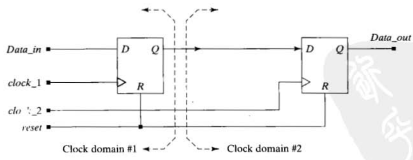


FIGURE 5-39 A situation requiring synchronization across clock domains.

[36] See Chapter 9.

Let us consider a scheme for designing a scanner/decoder for the hexadecimal keypad circuit shown in Figure 5-40. Each row of the keypad is connected to ground by a pull-down resistor. When a button is pressed, a connection is established between a row and a column at the location of the button; this connection will pull the row line up to the value of the column line corresponding to the location of the pressed key. If that column line is connected to the supply voltage, the row that is connected to that column by the pressed button will be pulled to the supply too; otherwise, the row line is pulled down to 0. The keypad code generator unit has control over the column lines and will exercise that control to systematically assert voltage on the column lines to detect the location of a pressed button.

A keypad code generator must implement a decoding scheme that (1) detects whether a button is pressed, (2) identifies the button that is pressed, and (3) generates an output consisting of the unique code of the button. The scanner/encoder will be implemented as a synchronous sequential machine with the button codes shown in Table 5-1. The outputs of the machine are the column lines, the code lines, and a signal, *Valid*, that indicates whether the value of *code* is valid. We will use the synchronizer in Figure 5-38(a) for the asynchronous input – the duration of a key press is long compared to the period of the system clock.

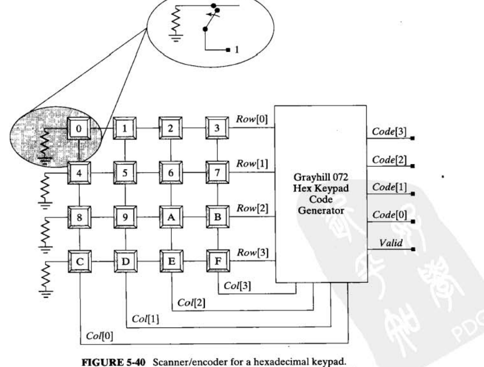


FIGURE 5-40 Scanner/encoder for a hexadecimal keypad.

TABLE 5-1 Keypad codes for a hexadecimal scanner.

Key	Row[3:0]	Col[3:0]	Code
0	0001	0001	0000
1	0001	0010	0001
2	0001	0100	0010
3	0001	1000	0011
4	0010	0001	0100
5	0010	0010	0101
6	0010	0100	0110
7	0010	1000	0111
8	0100	0001	1000
9	0100	0010	1001
A	0100	0100	1010
B	0100	1000	1011
C	1000	0001	1100
D	1000	0010	1101
E	1000	0100	1110
F	1000	1000	1111

To detect that a button is pressed, the machine can assert a 1 simultaneously on all of the column lines until detecting that a row line has been pulled up (by sensing that the OR of the row lines is a 1). The identity of the row line that is asserted is still not known. Then the scanner applies a 1 sequentially to each of the column lines, one at a time, until a row line is detected to be asserted. The location of the asserted column and row corresponds to the key that is pressed. This information is encoded uniquely, for each key. We will assume that only one switch is closed at a time (only one key is pressed).

The behavior of the keypad scanner/encoder is represented by the ASM chart[37] shown in Figure 5-41. The machine resides in state *S_0*, with all column lines asserted, until at least one row line is asserted. A signal formed from OR-ing the rows launches the activity of the machine. In *S_1* only column 0 is asserted. If a row is also asserted, the output *Valid* is asserted for one clock and the machine moves to *S_5*, where it remains with all of the column lines asserted until the row is de-asserted.[38] Then the machine returns to *S_0* for one cycle. Note that all of the columns are asserted in *S_5*, although only the column corresponding to the key needs be asserted. This eliminates the need to add two more states to the machine as it moves from *S_2*, *S_3*, and *S_4* and awaits de-assertion of the row.

The ASM chart implies that the decoding of the columns occurs in a priority manner, beginning with *Column_0*. If more than one column is asserted, the first one

[37]The asynchronous reset is shown at only *S_0*, but it is understood that asynchronous reset drives the state to *S_0* from any state in the chart.

[38]*Valid* can be used to control the writing of data to a storage unit, such as a first in, first out (FIFO) memory. See Chapters 8 and 9.

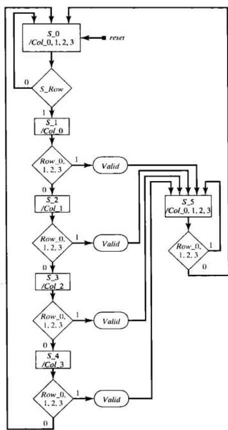


FIGURE 5-41 ASM chart of keypad scanner circuit.

decoded determines the code. The decision diamond leaving *S_0* tests the synchronized row signal. The decision diamonds leaving the other states can test the unsynchronized signal because in those states the signal from the key has already settled.

The Verilog models for the keypad scanner and its testbench are presented below, along with supporting modules.[39] The scanner is to be tested within the Verilog environment, not on a physical prototyping board with a physical keypad. Therefore, the testbench shown in Figure 5-42 must include (1) a signal generator that will simulate the assertion of a key, (2) a module, *Row_Signal*, which will assert a row line corresponding to the asserted key, and (3) *Hex_Keypad_Grayhill_072*,[40] the unit under test

[39]States are encoded as sized numbers of type **reg**; an **integer** would lead to synthesis of a needlessly large hardware register.
[40]See www.grayhill.com.

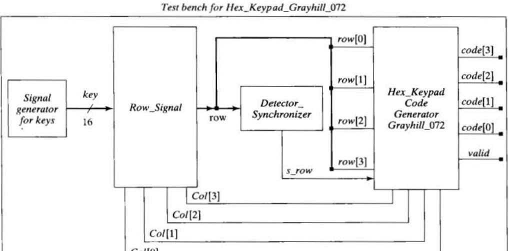


FIGURE 5-42 Testbench organization for the Grayhill 072 hexadecimal keypad scanner/encoder.

(UUT). After the model of the keypad scanner has been verified, it can serve as a user interface in simulating other systems, and can also be used in a physical environment with confidence that it should function correctly, which greatly reduces the scope of a search for the source of an error in the operation of a prototype.

The signal generator for key assertions is embedded in the testbench, where a single-pass behavior assigns values to *key* within a ***for*** loop, and a level-sensitive behavior reacts to changes in *key* by generating an ASCII string identifying the pressed key.[41] This enhances the display of waveforms produced by the simulator. The module *Row_Signal* detects assertion of a key and determines the row in which it is located. Its role is to replace the physical keypad driving the encoding unit. *Synchronizer* is a two-stage shift register whose output is a synchronized version of the OR of the asynchronous row signals. It asserts when any key is pushed. When the output of the synchronizer changes, the code generator unit determines which of the possible keys was pressed. It contains a state machine that sequentially asserts columns, and a level-sensitive behavior that parses row and column data to assert a hexadecimal code. The testbench instantiates *Hex_Keypad_Grayhill_072, Row_Signal*, and *Synchronizer*. Note that the code generator is synchronized by the positive edge of the clock, so *Synchronizer* is sensitive to the falling edge. This eliminates a potential race condition in the hardware. The particular synchronizer type used here is effective, provided that the duration of a keystroke is long compared to the period of the clock.

[41]Verilog does not have a data type for strings. Strings must be stored in a declared register, at 8 bits per character.

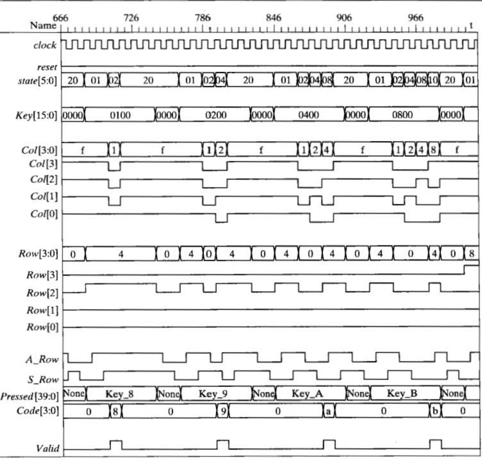


FIGURE 5-43 Simulation results for the Grayhill 072 hexadecimal keypad scanner/encoder.

The simulation results are shown in Figure 5-43, with pressed key values displayed in text format; the signals *A_Row* and *S_Row* are the input and output, respectively, of the synchronizer. For example, Key_9 corresponds to column 2 and row 4. The transitions of *S_Row* exhibit a latency of 1 cycle compared to those of *A_Row*. Notice that *valid* signals completion of the operations by asserting for one cycle after the key has been scanned and encoded.

```
// Decode the asserted Row and Col
//                  Hex Keypad Grayhill 072
//                          Col[0]    Col[1]    Col[2]    Col[3]
//                  Row[0]  0         1         2         3
//                  Row[1]  4         5         6         7
```

```
//                Row[2]  8      9      A      B
//                Row[3]  C      D      E      F

module Hex_Keypad_Grayhill_072 (
  output reg [3: 0]  Code, Col,
  output             Valid,
  input [3: 0]       Row,
  input              S_Row, clock, reset
);
  reg [5: 0] state, next_state;

  // One-hot state assignment
  parameter S_0 = 6'b000001, S_1 = 6'b000010, S_2 = 6'b000100;
  parameter S_3 = 6'b001000, S_4 = 6'b010000, S_5 = 6'b100000;

  assign Valid = ((state == S_1) || (state == S_2) || (state == S_3) || (state == S_4))
  && Row;

  // Does not matter if the row signal is not the debounced version.
  // Assumed to settle before it is used at the clock edge

  always @ (Row, Col)
   case ({Row, Col})
     8'b0001_0001:   Code = 0;
     8'b0001_0010:   Code = 1;
     8'b0001_0100:   Code = 2;
     8'b0001_1000:   Code = 3;
     8'b0010_0001:   Code = 4;
     8'b0010_0010:   Code = 5;
     8'b0010_0100:   Code = 6;
     8'b0010_1000:   Code = 7;
     8'b0100_0001:   Code = 8;
     8'b0100_0010:   Code = 9;
     8'b0100_0100:   Code = 10;      // A
     8'b0100_1000:   Code = 11;      // B
     8'b1000_0001:   Code = 12;      // C
     8'b1000_0010:   Code = 13;      // D
     8'b1000_0100:   Code = 14;      // E
     8'b1000_1000:   Code = 15;      // F
     default:        Code = 0;       // Arbitrary choice
   endcase

  always @ (posedge clock, posedge reset)
   if (reset == 1'b1) state <= S_0; else state <= next_state;

  always @ (state, S_Row, Row) // Next-state logic
   begin next_state = S_0; Col = 0;    // Default values; assign by exception
    case (state)
     // Assert all columns
     S_0: begin Col = 15; if (S_Row) next_state = S_1; end
     // Assert col 0
     S_1: begin Col = 1; if (Row) next_state = S_5; else next_state = S_2; end
```

```
      // Assert col 1
      S_2: begin Col = 2; if (Row) next_state = S_5; else next_state = S_3; end
      // Assert col2
      S_3: begin Col = 4; if (Row) next_state = S_5; else next_state = S_4; end
      // Assert col 3
      S_4: begin Col = 8; if (Row) next_state = S_5; else next_state = S_0; end
      // Assert all rows
      S_5: begin Col = 15; if (S_Row == 0) next_state = S_0; else next_state = S_5; end
      default: next_state = S_0;
    endcase
  end
endmodule

module Synchronizer (output reg S_Row, input [3: 0] Row, input clock, reset);
reg A_Row;

// Two-stage pipeline synchronizer

  always @ (negedge clock, posedge reset) begin
   if (reset == 1'b1) begin A_Row <= 0; S_Row <= 0; end
    else begin
    A_Row <= (Row[0] || Row[1] || Row[2] || Row[3]);
    S_Row <= A_Row;
   end
  end
endmodule

module Row_Signal (output reg [3:0] Row, input [15: 0] Key, input [3: 0] Col);
// Scan rows for pressed key
  always @ (Key, Col) begin

   // Combinational logic for key assertion
   Row[0] = Key[0] && Col[0] || Key[1] && Col[1] || Key[2] && Col[2] || Key[3] && Col[3];
   Row[1] = Key[4] && Col[0] || Key[5] && Col[1] || Key[6] && Col[2] || Key[7] && Col[3];
   Row[2] = Key[8] && Col[0] || Key[9] && Col[1] || Key[10] && Col[2] || Key[11] && Col[3];
   Row[3] = Key[12] && Col[0] || Key[13] && Col[1] || Key[14] && Col[2] || Key[15] &&
   Col[3];
  end
endmodule

module test_Hex_Keypad_Grayhill_072 ();
  wire [3: 0]      Code;
  wire             Valid;
  wire [3: 0]      Col;
  wire [3: 0]      Row;
  reg              clock, reset;
  reg [15: 0]      Key;
  integer          j, k;
  reg[39: 0]       Pressed;
  parameter        [39: 0]   Key_0 = "Key_0",
                             Key_1 = "Key_1",
                             Key_2 = "Key_2",
```

```
                              Key_3 = "Key_3",
                              Key_4 = "Key_4",
                              Key_5 = "Key_5",
                              Key_6 = "Key_6",
                              Key_7 = "Key_7",
                              Key_8 = "Key_8",
                              Key_9 = "Key_9",
                              Key_A = "Key_A",
                              Key_B = "Key_B",
                              Key_C = "Key_C",
                              Key_D = "Key_D",
                              Key_E = "Key_E",
                              Key_F = "Key_F",
                              None = "None";
always @ (Key) begin              // "one-hot" code for pressed key
 case (Key)
  16'h0000:    Pressed = None;
  16'h0001:    Pressed = Key_0;
  16'h0002:    Pressed = Key_1;
  16'h0004:    Pressed = Key_2;
  16'h0008:    Pressed = Key_3;
  16'h0010:    Pressed = Key_4;
  16'h0020:    Pressed = Key_5;
  16'h0040:    Pressed = Key_6;
  16'h0080:    Pressed = Key_7;
  16'h0100:    Pressed = Key_8;
  16'h0200:    Pressed = Key_9;
  16'h0400:    Pressed = Key_A;
  16'h0800:    Pressed = Key_B;
  16'h1000:    Pressed = Key_C;
  16'h2000:    Pressed = Key_D;
  16'h4000:    Pressed = Key_E;
  16'h8000:    Pressed = Key_F;
  default:     Pressed = None;
 endcase
end
Hex_Keypad_Grayhill_072    M1 (Code, Col, Valid, Row, S_Row, clock, reset);
Row_Signal                 M2 (Row, Key, Col);
 Synchronizer              M3 (S_Row, Row, clock, reset);
initial #2000 $finish;
initial begin clock = 0; forever #5 clock = ~clock; end
initial begin reset = 1; #10 reset = 0; end
initial begin for (k = 0; k <= 1; k = k+1)
 begin Key = 0; #20 for (j = 0; j <= 16; j = j+1)
  begin #20 Key[j] = 1; #60 Key = 0; end
 end
end
endmodule
```

Note that the first action of the level-sensitive cyclic behavior describing the next-state logic and the value of *Col* in Hex_Keypad_Grayhill_072 is to immediately assign value to *next_state* and to *Col*. Then the **case** statement decodes the state to make specific assignments to *next_state* and *Col*. The former assignments are referred to as default assignments. They guard against inadvertent omission of assignments in the remaining code, which could lead to accidental synthesis of latches (See Section 6.3). They also enable a more efficient coding style by allowing assignments to be made only by exception to the default assignments.

REFERENCES

1. Lee S. *Design of Computers and Other Complex Digital Devices*. Upper Saddle River, NJ: Prentice-Hall, 2000.
2. Mano MM, Kime CR. *Logic and Computer Design Fundamentals,* 3rd ed. Upper Saddle River, NJ: Prentice-Hall, 2003.
3. *Verilog HDL Coding—Semiconductor Reuse Standard*. Chandler, AZ: Motorola, 1999.
4. Abramovici M, et al. *Digital Systems Testing and Testable Design*. Rockville, MD: Computer Science Press, 1990.
5. Ciletti MD. *Modeling, Synthesis and Rapid Prototyping with the Verilog HDL*. Upper Saddle River, NJ: Prentice-Hall, 1999.
6. Clare CR. *Designing Logic Systems Using State Machines*. New York: McGraw-Hill, 1971.
7. Heuring VP, Jordan, HF. *Computer Systems Design and Architecture*. Upper Saddle River, NJ: Prentice-Hall, 2004.
8. Wakerly JF. *Digital Design Principles and Practices*, 4th ed. Upper Saddle River, NJ: Prentice-Hall, 2006.
9. Gajski D, et al. "Essential Issues in Codesign." In: Staunstrup J, Wolf W, eds. *Hardware/Software Co-Design: Principles and Practices*. Boston: Kluwer, 1997.
10. Katz RH., Boriello, G. *Contemporary Logic Design,* 2nd ed. Upper Saddle River, NJ: Prentice-Hall, 2004.
11. Digilent Inc. documentation (www.digilentinc.com).

PROBLEMS

Note: For all of the problems requiring development and verification of a design, a carefully written test plan, a documented testbench, and simulation results are to be provided with the solution. As a minimum, the test plan should describe (1) the functional features that will be tested and (2) how they will be tested.

1. Using a single continuous assignment, develop and verify a behavioral model implementing a Boolean equation describing the logic of the circuit below. Use the following names for the testbench, the model, and its ports: *t_Combo_CA()* and *Combo_CA (Y, A, B, C, D)*, respectively. *Note*: The testbench will have no ports. Exhaustively simulate the circuit and provide graphical and text output demonstrating that the model is correct.

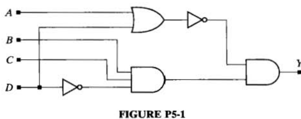


FIGURE P5-1

2. Using continuous assignments, develop Combo_CA, a model of the circuit in Figure P4-1. Then develop a testbench *t_Combo_all*, a testbench in which *Combo_str*, *Combo_UDP*, and *Combo_CA*, (see Problems 4.1 and 4.2) are all instantiated. In the testbench environment, name their outputs as *Y_str*, *Y_UDP*, and *Y_CA*, respectively. Simulate the models and produce graphical output showing their waveforms. Briefly discuss the results.
3. Repeat Problem 2 after assigning unit delay to all primitives and continuous assignments. Discuss the results.
4. Write a structural model that has the same functionality as *AOI_5_CA1*, (see Example 5.2) and then write a testbench that confirms that the two models have the same behavior.
5. Write a testbench and verify that *tr_latch* (Example 5.11) correctly models a transparent latch.
6. Develop a testbench and verify a gate-level model of an SR-latch.
7. Develop a testbench to verify the functionality of the user-defined primitive *AOI_UDP* (See Example 4.11). The testbench is to simulate the primitive and compare its output to that of a continuous assignment statement that describes the same functionality. The result of the comparison is to be reported by an error signal.
8. Write and verify a behavioral model of J-K flip-flop with active-low asynchronous reset.
9. Write and verify a Verilog model that will assert its output if a 4-bit input word is not a valid binary-coded decimal code.
10. Explain why the code fragment shown below will execute endlessly, and recommend an alternative description.

```
reg [3: 0] K
for (K=0; K<=15; K = K+1) begin
...
end
```

11. Using continuous assignment statements, develop and verify a model for *Comp_4_32_CA*, a circuit that compares four 32-bit unsigned binary words and asserts output(s) indicating which words have the largest value and which words have the smallest value.
12. Using a level-sensitive cyclic behavior and a suitable algorithm, develop and verify a model for *Comp_4_32_ALGO*, a circuit that compares four 32-bit words and asserts output(s) indicating which words have the largest value and which words have the smallest value.
13. Verify the functionality of *Universal_Shift_Reg* in Example 5.45.
14. Write a Verilog description of the circuit shown in Figure P5-15 and verify that the circuit's output, *P_odd*, is asserted if successive samples of *D_in* have an odd number of 1s.
15. Develop and verify a Verilog model of a 4-bit binary synchronous counter with the following specifications: negative edge-triggered synchronization, synchronous load and reset, parallel load of data, active-low enabled counting.

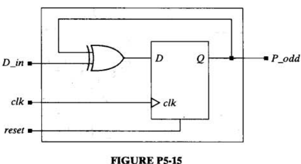


FIGURE P5-15

16. Modify the counter of the previous problem to have an additional output (ripple-carry output [RCO]) that asserts while the counter is at 1111_2. Cascade two such counters and demonstrate that the unit now works as an 8-bit counter.
17. Develop and verify a Verilog model of a 4-bit Johnson counter.
18. Develop and verify a Verilog model of a 4-bit BCD counter.
19. Develop and verify a Verilog model of a modulo-6 counter.
20. Write and verify a Verilog model of the datapath unit described by Figure 5-24.
21. Write a testbench and verify the functionality of *ring _counter* in Example 5.41.
22. Write and verify a Verilog model of *ring _counter_par_load*, by modifying the counter in Example 5.40 to have a parallel load capability.
23. Write a parameterized and portable Verilog model of an 8-bit ring counter whose movement is from its MSB to its LSB.

24. Write and verify a Verilog mode for a "jerky" ring counter having the register sequence shown in Figure P5-24(a); repeat for the counter in Figure P5-24(b).

count [7:0] (t ↓)

0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	1
0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	1
0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0

(a)

count [7:0] (t ↓)

1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1

(b)

FIGURE P5-24

25. Write and verify a Verilog model for a counter with the 18-step e18-step pattern cyclically until the counter is interrupted by a reset.

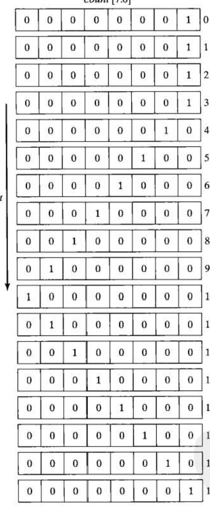


FIGURE P5-25

26. The partially completed ASMD chart in Figure P5-26 describes a machine having inputs *rst*, *G0*, *F*1, and *F*2, and scalar outputs *B* and *C* and the 8-bit datapath, *A*. Develop (a) a block diagram showing the datapath unit, the control unit and the interface signals, (b) a complete ASMD chart, and (c) a Verilog model for the partitioned machine. *Note*: The machine has synchronous reset, and the action of *rst* drives the state to *S_idle* from every state and clears registers *A* and *C*. The register operations and state transitions are to be synchronized to the rising edge of a common clock. Verify the behavior of the machine.

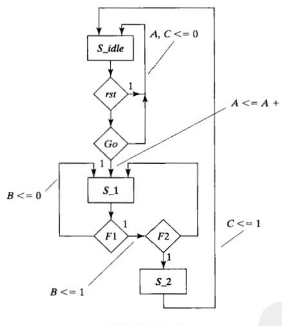


FIGURE P5-26

27. Develop and verify an 8-bit ALU having input datapaths *a* and *b*, output data path {*c_out*, *sum*} and an operand *Oper*, and having the functionality indicated in Figure P5-27.

Operand	Function
Add	$a + b + c_in$
Subtract	$a + \sim b + c_in$
Subtract_a	$b + \sim a + \sim c_in$
Or_ab	{1'b0, *a* / *b*}
And_ab	{1'b0, *a* & *b*}
Not_ab	{1'b0, (~*a*) & *b*}
Exor	{1'b0, *a* ^ *b*}
Exnor	{1'b0, *a* ~^ *b*}

FIGURE P5-27

28. Combine the ALU from Problem 27 with an 8-bit version of a register file to form an architecture like that shown in Figure 5-34. Develop a testbench to verify each functional unit and the overall structure.
29. For the four-stage pipeline machine shown below, (a) develop the block diagram for a partitioned machine showing the control unit, a datapath unit, the input and output signals, and the interface signals between the control and datapath units, (b) develop a complete ASMD chart describing the operation of the machine, and (c) develop and verify a Verilog model of the machine. The machine's output is the content of the 32-bit register *R*0.

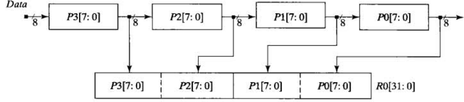


FIGURE P5-29

30. Modify the machine described by the ASMD chart in Figure 5-24 by replacing the registers *P0* and *P1* by an appropriately sized shift register, whose contents are shifted under the actions of *En* and *Ld*. Develop and verify a Verilog model of the machine.
31. A designer contends that the reset signal can be removed from the hexadecimal keypad machine presented in Section 5.18 if logic is added to direct unused states to *S_5*. Discuss the validity of this claim. If it is true, what are the tradeoffs between the two circuits?
32. Modify the ASM chart for the keypad scanner circuit (see Section 5.18) to require that a key be held for 25 clock cycles before it is interpreted to be a valid keystroke. Write and verify a Verilog model of the revised design.

33. Write and verify a Verilog model of a programmable clock generator, *Clock_Prog*, having input ports *En* and *rst*, an output port (*clk*), and the parameters *Latency*, *Offset*, and *Pulse_Width* as shown in Figure P5-33. The default (declared) values of the parameters are 100, 75, and 25, respectively. *Clock_Prog* is not intended to be synthesized; it can be used to generate a clock signal for simulating other modules that are instantiated in a testbench, say *t_Some_Unit_Under_Test*. To provide a mechanism for overriding the default parameters of *Clock_Prog* in a particular application, include the following "annotation module" in your project. The annotation module uses hierarchical dereferencing, where M1 is the instance name of the unit under test in *t_Some_Unit_Under_Test*. The testbench must demonstrate that this works. The example below replaces the default values of *Latency*, *Offset*, and *Pulsewidth* by 10, 5, and 5, respectively. The parameters represent time steps in the units used for the simulation. The input signal *En* is to be generated within the testbench to (optionally) simulate a condition in which the clock signal is not immediately provided to the unit under test. *Note*: The latency determines the delay between the assertion/de-assertion of reset at any time after the start of simulation and the availability of the clock. After the period of latency expires the waveform defined by the cycle time is to be generated repeatedly for the duration of simulation.

```
module annotate_Clock_Prog ();
 defparam t_Some_Unit_Under_Test.M1.Latency = 40;
 defparam t_Some_Unit_Under_Test.M1.Offset = 15;
 defparam t_Some_Unit_Under_Test.M1.Pulse_Width = 5;
endmodule
```

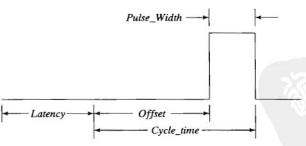


FIGURE P5-33 Parameterized waveform of a programmable clock generator.

34. Write and verify a Verilog module of a programmable 8-bit testbench pattern generator, *Pattern_Gen_8B*, having an output word, *Stim_Pattern*, and an input, *enable*, which activates the generator. Once the generator is activated, the bit patterns are to be generated exhaustively for a prescribed number of cycles, provided that *enable* is asserted. The module is to have the following parameters:

    Offset: An initial offset between the time that enable is asserted and the time that the first pattern of *Stim_Pattern* is asserted.
    Stim_size: The width of *Stim_Pattern*.
    Cycles: The number of times the entire pattern set is to be repeated.
    Period: The interval between successive patterns.

35. Hamming codes are used in computer memory systems to detect and correct errors in data [8]. Each information word is augmented by additional bits that distinguish valid encoded words from corrupted words. A Hamming code with a minimum distance of 4 bits can correct a single-bit error in the encoded distance-2 word and detect, but not correct, any 2-bit error. Figure P5-35 lists

H_word	
D_word	Parity Bits
0000	000
0001	011
0010	101
0011	110
0100	110
0101	101
0110	011
0111	000
1000	111
1001	100
1010	010
1011	001
1100	001
1101	010
1110	100
1111	111

FIGURE P5-35

information bytes (*D_word*) and the parity bits that are appended to the data bits to form an encoded word (*H_word*). (a) Develop a Verilog module, *Hamming_Encoder_MD3*, that will produce a 7-bit value *H_word* from a 4-bit value, *D_word*. (b) Develop a testbench to verify the functionality of *Hamming_Encoder_MD3*. (c) Browse the literature on parity circuits to find a decoder that will correct a single-bit error in *H_word*. (d) Develop and verify a Verilog module, *Hamming_Decoder_MD3*, that will detect any 2-bit error in *H_word* and correct any single-bit error. (e) Develop a testbench that will simulate the encoder, introduce random 1- and 2-bit errors in *H_word*, and take action to correct a single-bit error and signal the presence of a 2-bit (uncorrectable) error. (f) Demonstrate the decoder's behavior if a 3-bit (burst) error corrupts *H_word*.

36. Using library cells with known setup and hold timing parameters, and develop and verify a model of the synchronizer circuit in Figure 5-38(b). Explore its operation with *Asynch_in* being (a) a long pulse and (b) a short pulse relative to the clock. How does it behave if an asynchronous pulse arrives within two or more successive clock cycles?
37. Using continuous assignments, develop and verify a model of a transparent latch having active-high enable, active-low reset, and active-high set. Use the text below.

```
module Latch_RbarS_CA (q_out, data_in, enable, reset_bar, set);
...
endmodule
```

38. Using *d_prim1* (See Example 4.15) develop and verify a structural model of a 4-bit shift register whose datapath consists of a parallel load of data and a serial output of the least significant bit.
39. The barrel shifter described in Example 5.45 forms *Data_out* by rotating *Data_in* by one bit toward its MSB. Develop a more general barrel shifter that can rotate in either direction by a specified number of bits.
40. Using state transitions graphs, develop and verify Mealy and Moore behavioral models (Verilog) of code converters that produce an output waveform having an NRZI format (See Chapter 3). For the waveform of *B_in* shown in Figure 3-24, identify and compare latency and valid bit times of the output waveforms of the machines. Caution: Avoid the trap of attempting to read the data bits on the clock edge that synchronizes their transitions.
41. Design and verify a structural (gate-level) Verilog module of a decade counter—that is, a counter whose count sequence is 0, 1, 2, . . . , 9, 0, 1, 2 ,,, repeatedly. Hint: consider a user-defined primitive of a D-type flip-flop.
42. Complete the partially drawn ASMD charts given in Figure 5-41.
43. Draw an ASMD chart for the up-down counter described in Example 5.42.
44. Draw an ASMD chart for the barrel shifter described in Example 5.45.
45. Draw an ASMD chart for the universal shift register described in Example 5.46.
46. Using the conditional operator, write a continuous assignment statement that is equivalent to the cyclic behavior: **always @** (data, enable) **if** (enable) y <= data;
47. Develop an ASM chart and write a Verilog model of a Moore machine sequence detector that, if enabled, will detect the first occurrence of the pattern 101, then suspend activity until it is enabled again. Clearly state any assumptions needed for your design.

48. Draw an ASM chart of a Mealy machine that samples a serial bit stream on the rising edge of the clock and asserts an output if the last four samples are such that two ones are followed by two zeros.
49. A pipelined datapath for a sequential machine is described by the datapath unit and the *partially* completed ASMD chart shown below. Under external control, the machine is to load successive 8-bit words into a 16-bit register, and then produce a serial output, with the LSB leading the bit sequence. A counter, *N*, is to be used to control the serial bit stream. The machine is to assert a signal, *Ready*, while the machine is in the state *S_idle*, a signal, *Full*, for one cycle after the 16-bit register is full, a signal, *Busy*, while the serial output is being transmitted, and a signal, *Done*, for one cycle after the serial output has been sent. (a) Provide additional details to complete the ASMD chart, (b) name and list the signals that will control the data path unit, (c) write and verify a Verilog model for the datapath unit, (d) write and verify a Verilog model for the control unit, and (e) write and verify a Verilog model of *Top_Unit* containing the datapath and control units and interfacing to the environment.

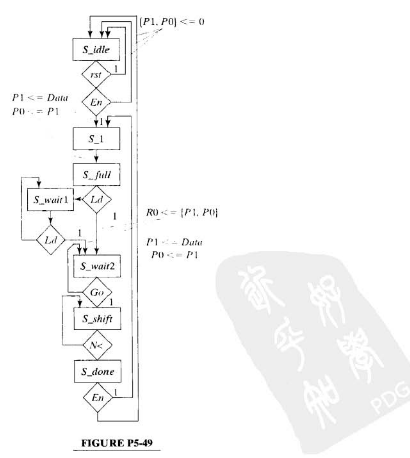


FIGURE P5-49

50. Draw the ASM chart of a Moore machine that (a) remains in a reset state until an external signal, *En*, is asserted, (b) samples a bit stream at the rising edge of a clock, (c) asserts an output signal, *Found*, for one cycle if the last four samples are such that two 1s are followed by two 0s, and (d) pauses for two cycles before returning to the reset state after asserting *Found*.

CHAPTER 6

Synthesis of Combinational and Sequential Logic

Chapters 2 and 3 reviewed basic concepts and Karnaugh maps for designing combinational and sequential circuits by manual methods. This chapter presents an automated approach to optimizing the Boolean equations describing a multi-input, multi-output logic circuit.

The design flow for an application-specific integrated circuit (ASIC) depends on software tools to manage and manipulate the databases that describe large, complex circuits. Among these tools, the synthesis engine plays a strategic role by automating the task of minimizing a set of Boolean functions and mapping the result into a hardware implementation that meets design objectives. Manual methods relying on Karnaugh maps cannot accommodate large circuits, and their use is error-prone, tedious, and time consuming. Automated software tools can optimize logic quickly, and without error. To use them effectively, a designer must understand a hardware description language (HDL) and be skilled at writing descriptions that conform to the constraints imposed by the tool. Just as understanding Karnaugh maps is the key to manual design methods, *understanding how to write synthesis-friendly Verilog models is the key to automated design methods.*

Synthesis tools perform many tasks, but the following steps are critical: (1) detect and eliminate redundant logic, (2) detect combinational feedback loops, (3) exploit don't-care conditions, (4) detect unused states, (5) detect and collapse equivalent states, (6) make state assignments, and (7) synthesize optimal, multilevel realizations of logic subject to constraints on area and/or speed in a physical technology. This last step

involves both optimization and technology mapping. The steps that were performed manually in Chapters 2 and 3 will be executed automatically by a synthesis tool. This will shorten the design cycle, reduce the burden placed on the designer, and increase the likelihood that the design will be correct.

HDLs are the entry point for a modern synthesis-oriented design flow for ASICs and field-programmable gate arrays (FPGAs). A designer must understand how to use language constructs to describe combinational and sequential logic, and must know how to write synthesis-friendly descriptions. In this chapter, we will present several examples that demonstrate how to write synthesis-ready models of combinational and sequential logic (that is, models that can be used with a synthesis tool to produce a gate-level realization of the described functionality). The examples will help the reader anticipate the results of synthesis—that is, know what circuit will be created from the description.

6.1 Introduction to Synthesis

Circuit design begins with specification of the circuit's functionality—what the circuit does—and ends with physical hardware that implements the functionality reliably, with sufficient performance and acceptable cost. Models of circuits can be classified according to levels of abstraction and views [1]. There are three common levels of abstraction: architectural, logical, and physical. A description at the architectural level of abstraction implies the operations that must be executed by the circuit to transform a sequence of inputs into a specified sequence of outputs, but does not associate the operations with specific clock cycles. These operations will ultimately be implemented by distinct, interconnected, and synchronized functional units, hence our use of the term *architecture*. The design challenge here is to extract from the architectural description a structure of computational resources that implements the functionality of the machine.

A model at the logical level of abstraction describes a set of variables and a set of Boolean functions that the circuit must implement. An architecture of register resources and functional units and the sequential activity of a logic-level model are part of its description. The design task here is to translate the Boolean descriptions into an optimized netlist of combinational gates and storage registers that will implement the circuit at a satisfactory level of performance. Synthesis tools are used for this task. A geometrical model describes the shapes that define the doping regions of semiconductor materials used to fabricate transistors. HDLs do not treat geometric models.

Design may begin at a high level of abstraction, but ultimately ends in physical reality. Along the path between the two, an HDL can facilitate the design process by providing different views of the circuit. There are three common views: behavioral, structural, and physical. A behavioral view of an architectural model could be an algorithm that specifies a sequence of data transformations. A structural view of the same model might consist of a structure of datapath elements (registers, memory, adders) and a controller that implement the algorithm. State-transition graphs (STGs), next state/output tables, and algorithmic state machine (ASM) charts are behavioral views of a logic-level model of a

circuit. A structural view of a logic-level model would consist of a schematic of gates that implement the functionality of the behavior described by the ASM chart in the behavioral view of models. Physical descriptions, the actual geometric patterns of the physical devices that implement the circuit, will not be considered here.

Synthesis creates a sequence of transformations between views of a circuit, from a higher level of abstraction to a lower one, with each step leading to a more detailed description of the physical reality. Figure 6-1 shows a modified Y-chart [2] depicting an axis for the behavioral, structural, and physical views of a circuit. We have annotated the behavioral and structural axis to show Verilog constructs that can be used to create a view. The chart also shows a sequence of transformations, in which (1) behavioral synthesis transforms an algorithm (behavioral view) to an architecture of registers and a schedule of operations that occur in specified clock cycles (structural view), (2) a Verilog model of this architecture is formed as a data flow/register transfer level (RTL) description (behavioral view), and (3) logic synthesis translates the data flow/RTL description into a Boolean representation and synthesizes it into a netlist (structural view). The Y-chart in Figure 6-1 has been annotated to represent these transformations, and to indicate which Verilog constructs (e.g., continuous assignments) describe the design at each stage of the activity.

6.1.1 Logic Synthesis

Logic synthesis generates a structural view from a logic level view (description) of a circuit [1]. The resulting structural view is a netlist of structural primitives. The logic-level view is a set of Boolean equations described by a set of continuous-assignment statements in Verilog or an equivalent level-sensitive behavior. Logic synthesis includes

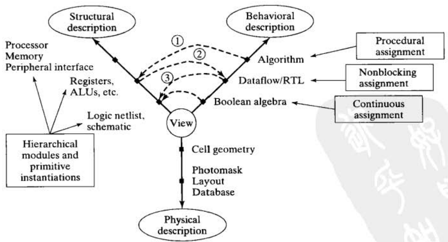


FIGURE 6-1 A Y-chart representation of Verilog constructs supporting synthesis activity in three views of a circuit: behavioral, structural, and physical.

transforming a given netlist of primitives into an optimized netlist of Verilog primitives and mapping a generic optimal netlist into an equivalent circuit composed of physical resources in a target technology, such as the cells in an ASIC cell library. A logic synthesis tool has the general organization shown in Figure 6-2. The tool forms a technology-specific hardware realization from a Verilog RTL model or primitive netlist.

The translation engine of a synthesis tool reads and translates a Verilog-based description of the input–output behavior of a circuit into an intermediate internal representation of Boolean equations describing combinational logic and other representations of storage elements and synchronizing signals. Techniques for simultaneously optimizing the set of equations will remove redundant logic, exploit don't-care conditions, and share internal logic sub-expressions as much as possible to produce an optimized, generic (technology-independent) multilevel logic implementation.

In general, there may be multiple realizations of a multi-input, multi-output combinational logic circuit, but the transformations made in synthesis are guaranteed to maintain the input–output equivalence of the circuit and produce a testable circuit [3].[1] The optimization process is based on an iterative search, not the solution of an analytical model, so the result is not necessarily the global optimum that could be found over the domain of possible circuits that have equivalent input–output behavior. Logic optimization is followed by performance optimization, which seeks a circuit that has optimal performance in the physical technology.

The translation engine creates an internal product-of-sums (POS) form of a Boolean expression by factoring the sum-of-products (SOP) form into expressions whose Boolean product is equivalent to the SOP form. When the POS representation for two or more outputs contains a common sub-expression, the synthesis tool may minimize the internal logic needed to realize the circuit by generating the common sub-expression once and sharing it (through fanout) among output variables.

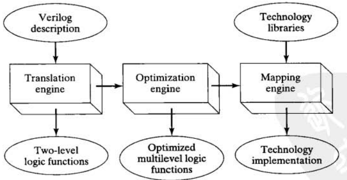


FIGURE 6-2 Synthesis-tool organization.

[1] We will consider testability, fault simulation, and test generation in Chapter 11.

Techniques for simultaneously optimizing the set of equations will remove redundant logic, exploit don't-care conditions, and share internal logic sub-expressions [3] as much as possible to produce an optimized, generic (technology-independent) multilevel logic implementation. For example, a "?" entry in the input–output table of a combinational user-defined primitive (UDP) represents a don't-care condition that can be used in Boolean minimization. However, this may lead to a mismatch between the seed code's behavior and that of the synthesis product.

The functional relationships between the inputs and outputs of a combinational Boolean circuit can be expressed as a set of two-level Boolean equations in either SOP or POS form, which must be optimized by the tool. A set of Boolean equations describing a multi-input, multi-output (MIMO) combinational logic circuit can always be optimized to obtain a set of Boolean equations whose input–output behavior is equivalent while containing the fewest literals [4]. Software tools exist to perform this optimization and to express the resulting Boolean equations with the resources of a technology-specific parts library. Consequently, a Verilog description consisting only of a netlist of combinational primitives without feedback can always be synthesized.

Espresso [5] is a commonly used software system developed at the University of California at Berkeley for minimizing the number of cubes in a single Boolean function. It performs several transformations on a circuit to arrive at its optimal representation. For example, the transformation *Expand* replaces cubes with prime implicants that have fewer literals. *Irredundant* extracts from a cover of the function a minimal subset that also covers the function (i.e., redundant logic has been removed). *Reduce* transforms an irredundant cover into a new cover of the same function.

Espresso minimizes a single Boolean function of several Boolean variables, but it does not provide a solution to the problem of optimizing a multi-input, multi-output combinational logic circuit. In general, the optimization of a set of Boolean equations is not obtained by applying Espresso separately to the individual functions in the set. Instead, a multilevel optimization program, such as misII [3], must be used to simultaneously optimize the set of equations as an aggregate.

Logic synthesis treats a set of individual Boolean input–output equations as a multilevel circuit (see Figure 6-3). By removing redundant logic, sharing internal logic, and exploiting input and output don't-care conditions, a logic synthesis tool optimizes a multilevel set of Boolean equations and achieves a better realization (i.e., area-efficient) than could be obtained by merely optimizing the individual input–output equations.

Like Espresso, the misII multilevel logic optimization program performs several transformations on a logic circuit while searching for an optimal description of a digital circuit. Four transformations play a key role in the misII algorithm for logic synthesis: decomposition, factoring, substitution, and elimination.

The operation of *decomposition* transforms the circuit by expressing a single Boolean function (i.e., the Boolean expression representing the logic value of a node in the circuit) in terms of new nodes.

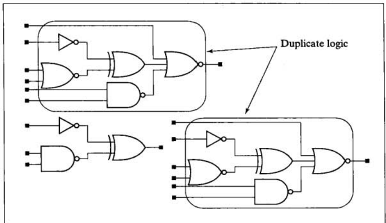


FIGURE 6-3 Multilevel combinational logic.

Example 6.1

Figure 6-4 shows the schematic of a function, F, that is to be decomposed in terms of new nodes X and Y. The original form of F is described by the Boolean equation:

$$F = abc + abd + a'c'd' + b'c'd'$$

The Espresso operation of *decomposition* expresses F in terms of two additional internal nodes, X and Y, to form the circuit shown in Figure 6-5. These internal nodes could then be re-used to form other expressions and thereby achieve a reduction in hardware area.

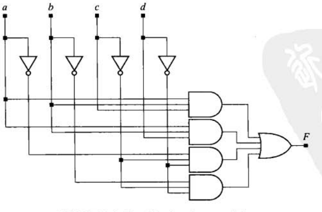


FIGURE 6-4 Circuit before decomposition.

$$F = XY + X'Y'$$
$$X = ab$$
$$Y = c + d$$

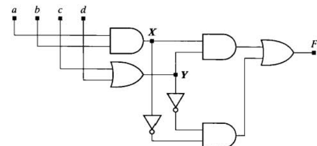


FIGURE 6-5 Circuit after decomposition.

End of Example 6.1

In contrast to *decomposition*, which represents an *individual* function in terms of intermediate nodes,[2] the operation of *extraction* expresses a *set* of functions in terms of intermediate nodes by expressing each function in terms of its factors and then detecting which factors are common to the functions.

Example 6.2

Figure 6-6 shows a directed acyclic graph (DAG) [1] representing the set of functions, F, G, and H, which is to be decomposed in terms of new nodes X and Y, with

$$F = (a + b)cd + e$$
$$G = (a + b)e'$$
$$H = cde$$

and X and Y are given by

$$X = a + b$$
$$Y = cd$$

The nodes of the graph represent Boolean operations on the data that are associated with the edges that enter the node. The *extraction* process finds those members of the set of functions with the factor $(a + b)$ and the factor cd. The factors are extracted from those functions and replaced by the new internal nodes X and Y to produce the new DAG shown in Figure 6-7.

End of Example 6.2

[2]The nodes of the directed acyclic graph represent operations that are performed on data (e.g., addition).

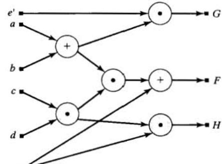


FIGURE 6-6 DAG of a set of functions before extraction.

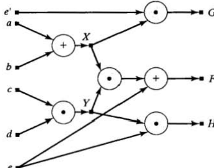


FIGURE 6-7 Directed acyclic graph of a set of functions after extraction.

The optimization process seeks a set of intermediate nodes to optimize the circuit's delay and area. This step may lead to significant reduction in the overall silicon area, because the intermediate nodes correspond to factors that are common to more than one Boolean function, and therefore can be shared to eliminate replicated logic. The task of finding the common factors among a set of functions is called *factoring*. Factoring produces a set of functions in a POS form. It creates a structural transformation of the circuit from a two-level realization to an equivalent multilevel realization that uses less area, but is possibly slower.

Example 6.3

Figure 6-8 shows a DAG representing a function, F, that is factored to identify its Boolean factors in POS form. The function represented by the DAG is described by the Boolean equation:

$$F = ac + ad + bc + bd + e$$

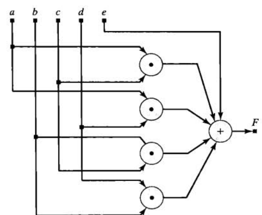


FIGURE 6-8 The DAG of a function before factoring.

The factored form of F is

$$F = (a + b)(c + d) + e$$

Factorization seeks the factored representation of a function with the fewest number of literals. The DAG of the factored form of F is shown in Figure 6-9.

End of Example 6.3

The *substitution* process expresses a Boolean function in terms of its inputs and another function. Since both functions need to be implemented, this step provides a potential reduction of replicated logic.

Example 6.4

The DAG in Figure 6-10(a) represents the function F before the function G is substituted into it, where

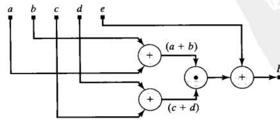


FIGURE 6-9 The DAG of the factored form of a function.

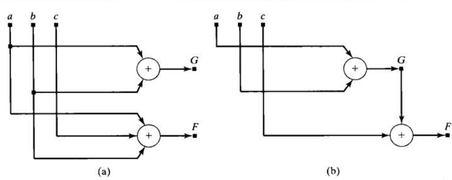


FIGURE 6-10 DAG of a function before (a) and after (b) substitution.

$$G = a + b$$
$$F = a + b + c$$

After substitution, F has the form:

$$F = G + c$$

and the DAG shown in Figure 6-10(b).

End of Example 6.4

Sometimes the process of *decomposition* has to be undone in the search for an optimal realization. *Elimination* removes (collapses) a node in a function and reduces the structure of the circuit. This step is also referred to as *flattening* the circuit. The transformation would ultimately eliminate the area-efficient internal multilevel structure and create a faster two-level structure.

Example 6.5

The DAG in Figure 6-11(a) represents the function F before the function G is eliminated from it, where before elimination

$$F = Ga + G'b$$
$$G = c + d$$

and after elimination:

$$F = ac + ad + bc'd'$$

The new DAG for F is shown in Figure 6-11(b).

End of Example 6.5

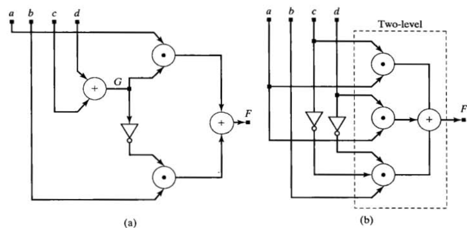


FIGURE 6-11 DAG of a function before (a) and after (b) elimination of an internal node.

In general, there may be multiple realizations of a multi-input, multi-output combinational logic circuit, but the transformations in misII are guaranteed to maintain the input–output equivalence of the circuit and produce a testable circuit. Multilevel networks may be used because two-level networks might require higher fan-in than is practical. Multilevel circuits present an opportunity to share logic, but can be slower than a two-level counterpart.

6.1.2 RTL Synthesis

RTL synthesis begins with an architecture and converts language-based RTL statements into a set of Boolean equations that can be optimized by a logic synthesis tool. This level of synthesis transforms a logic-level view described in terms of operations on registers in the context of a fixed architecture, and creates equivalent Boolean equations and synthesizes an optimal realization of the given architecture.

RTL synthesis begins with the assumption that a set of hardware resources is available and that the scheduling and allocation of resources have been determined,[3] subject to the constraints imposed by the resources of the architecture. The RTL description represents either a finite-state machine (FSM) or a more general sequential machine that makes register transfers within the boundaries of a predefined clock cycle. RTL descriptions in Verilog use language operators and make synchronous concurrent assignments to register variables (i.e., nonblocking assignments). The Verilog language operators represent a variety of register transfer operations, and are easily synthesized. Logic synthesis tools operate in this domain (i.e., they

[3]Resource allocation and scheduling are discussed in Chapter 9.

generally lack the ability to analyze tradeoffs between scheduling and allocation of resources, and use, instead, the implicit solution imposed by the writer of the description). Nonetheless, these tools have a broad and significant scope of use. The synthesis engine must minimize and optimally encode the state of the RTL-described machine, optimize the associated combinational logic, and map the result into the target technology.

6.1.3 High-Level Synthesis

High-level synthesis, also called "behavioral synthesis" or "architectural synthesis," has the goal of creating an architecture whose resources can be scheduled and allocated to implement an algorithm, such as an algorithm for digital signal processing (DSP).[4] The algorithm to be synthesized describes only the functionality of the circuit; it does not explicitly declare a structure of registers and datapaths. Thus, many different architectures may implement the same functional specification.

The starting point for high-level synthesis is an input–output algorithm, with no details about the implementation. A behavioral synthesis tool executes two main steps in order to create an architecture of datapath elements, control units, and memory: resource *allocation* and resource *scheduling*. Dataflow graphs display dependencies between data, and the allocation step identifies the operators used in an algorithm (e.g., +) and infers the need for memory resources to hold data implied by the sequential activity of the algorithm. Allocation binds these operators and memory resources to datapath resources (e.g., multiplication operators can be bound to a multiplier cell).

In scheduling, the operations in the behavioral description are assigned to specific clock cycles (implicit states) to implement the ordered sequential activity flow of the algorithm. For example, Figure 6-12(a) [2] shows three procedural assignments that execute sequentially in a Verilog cyclic behavior within a hypothetical algorithm. A compiler must form parse trees from the statements, and then extract a dataflow graph from the set of parse trees. Figure 6-12(b) shows a DFG and a sequential schedule for its operators. The statements imply a time-ordered sequential activity, and are allocated to the clock cycles.

Scheduling assigns the operations of the behavioral description to clock cycles. The dataflow graph shown in 6-12(b) can be used to infer memory read/writes and schedule activity in clock cycles using the available resources. This ultimately determines the number of computational units (operators) that will be used in a given clock cycle and shared between cycles. Because multiple architectures might have the same functionality, a behavioral synthesis tool must explore architectural alternatives and consider a number of trade-offs and/or constraints (e.g., data rates, input/output channels, clock period, pipelining, datapath width, latency, throughput, speed, area, and

[4]We will consider algorithm synthesis in more detail in Chapter 9.

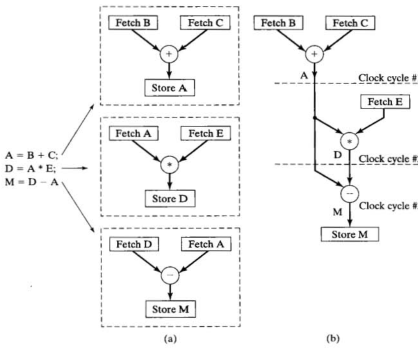


FIGURE 6-12 Representation of a behavioral model: (a) parse trees and (b) data flow graph.

power). High-level synthesis is an area of active research, although electronic design automation (EDA) tool vendors now offer tools for high-level synthesis. Coding styles and other details of these tools will not be presented here.

6.2 Synthesis of Combinational Logic

There are many ways to describe combinational logic with the Verilog HDL, but some are not supported by synthesis tools.[5] Synthesizable combinational logic can be described by (1) a netlist of structural primitives, (2) a set of continuous-assignment statements, and (3) a level-sensitive cyclic behavior. UDPs and ***assign*** ... ***deassign*** procedural continuous assignments can also describe combinational logic, but most EDA vendors have chosen to not support these options.

[5]See "Additional Features of Verilog" at the companion web site for Verilog constructs that are commonly supported by synthesis tools, and constructs that are not supported.

A design that is expressed as a netlist of primitives should be synthesized to remove any redundant logic before mapping the design into a technology. This provides a measure of safety to the design, because most designers have difficulty discovering and removing redundant logic from any but the simplest circuits. Synthesizing a netlist ensures that the logic has been minimized correctly.

Example 6.6

Figure 6-13(a) shows the pre-optimized schematic of the circuit described by the netlist of primitives in *boole_opt*. The synthesized circuit shown in Figure 6-13(b) has a more efficient gate-level implementation (less area) than the generic circuit would have if parts from a cell library are substituted for the primitives.

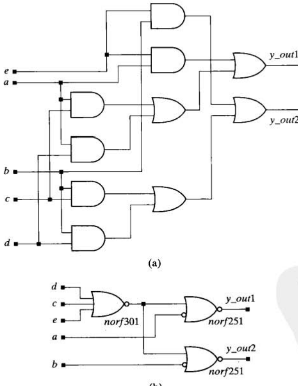


FIGURE 6-13 Logic synthesis: (a) schematic for a netlist of primitives and (b) the circuit synthesized from the netlist.

```
module boole_opt(output y_out1, y_out2, input a, b, c, d, e);
 wire   y1, y2, y3, y4, y5, y6, y7, y8;
 and    (y1, a, c);
 and    (y2, a, d);
 and    (y3, a, e);
 or     (y4, y1, y2);
 or     (y_out1, y3, y4);
 and    (y5, b, c);
 and    (y6, b, d);
 and    (y7, b, e);
 or     (y8, y5, y6);
 or     (y_out2, y7, y8);
endmodule
```

End of Example 6.6

Continuous-assignment statements are synthesizable. The expression that assigns value to a net variable in a continuous assignment statement will be translated by the synthesis tool to an equivalent Boolean equation, which can be optimized and synthesized in physical hardware simultaneously with other continuous assignments in the module.[6] The designer must verify that the continuous assignment correctly describes the logic, and that the synthesized multilevel logic circuit has the correct functionality.

Example 6.7

The continuous assignment in *or_nand* synthesizes to the circuit in Figure 6-14.

```
module or_nand (output y, input enable, x1, x2, x3, x4);
  assign y = ~(enable & (x1 | x2) & (x3 | x4));
endmodule
```

End of Example 6.7

A level-sensitive cyclic behavior will synthesize to combinational logic[7] if it assigns a value to each output for every possible value of its inputs. This implies that the event control expression of the behavior must be sensitive to every input and that every path of the activity flow must assign value to every output.

[6]Remember, simultaneous optimization of Boolean equations discovers and exploits logic that can be shared among them.

[7]Feedback-free and without latches.

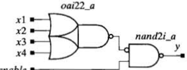


FIGURE 6-14 Combinational logic synthesis: circuit synthesized from a continuous assignment.

Example 6.8

Chapters 4 and 5 presented examples that described a 2-bit comparator by Boolean equations, and a netlist of primitives (see Example 4.4), and a set of continuous assignments (see Examples 5.6, 5.12, and 5.13). The descriptions synthesize equivalent circuits. In the model below, *comparator*, a level-sensitive behavior describes the functionality of a 2-bit comparator. The algorithm in the model exploits the fact that the data words are identical if all of their bits match in each position. Otherwise, the most significant bit at which the words differ determines their relative magnitude. This algorithm is not as simple or as elegant as the ones in Chapters 4 and 5, but it serves to illustrate the power of synthesis tools to correctly synthesize combinational logic from a level-sensitive cyclic behavior containing a loop construct.[8] The synthesized circuit is shown in Figure 6-15.

```
module comparator #(parameter size = 2)(              // Alternative algorithm
 output reg a_gt_b, a_lt_b, a_eq_b, input [size -1: 0] a, b);
 integer k;
 always @(a, b) begin: compare_loop
  for (k = size; k > 0; k = k-1) begin
   if (a[k] != b[k]) begin
    a_gt_b = a[k];
    a_lt_b = ~a[k];
    a_eq_b = 0;
    disable compare_loop;
   end              // if
  end               // for loop
  a_gt_b = 0;
  a_lt_b = 0;
  a_eq_b = 1;
 end                // compare_loop
endmodule
```

[8]A cyclic behavior executes repeatedly, subject to embedded timing controls.

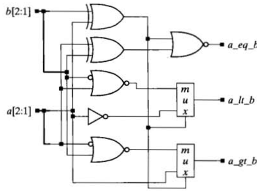


FIGURE 6-15 Combinational logic synthesis: 2-bit comparator circuit synthesized from a level-sensitive cyclic behavior with a ***for*** loop.

End of Example 6.8

A synthesizable Verilog model of combinational logic describes the functionality of the circuit and is written independently of the technology in which the physical circuit will be realized. Timing imposes a constraint on the speed of a circuit, but not on its functionality. The speed at which ASIC parts can operate is proportional to the physical geometry of the part. Faster parts require more area. Technology-dependent timing constructs, such as gate propagation delays, are not to be included in the model of functionality. Synthesis tools may require that the logic be free of feedback loops (e.g., no cross-coupled NAND gates). Functions and tasks will synthesize to combinational logic if they do not contain incomplete case statements or conditionals (***if***), and do not contain embedded timing controls (**#, @**, or ***wait***).

A synthesis tool synthesizes combinational logic to implement the expression of the ***case*** construct, and the expressions associated with a conditional operator. If a multiplexed datapath has control logic other than a single select bus, the synthesis tool will create additional combinational logic on the control line of the mux to govern the activity of the multiplexer.

Example 6.9

The continuous assignment in *mux_logic* has logic determining whether *sig_a* or *sig_b* is selected. The description synthesizes to the circuit shown in Figure 6-16, which has logic at the control line of the synthesized mux.

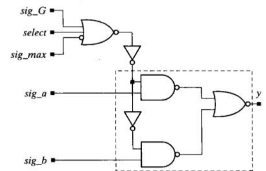


FIGURE 6-16 Circuit synthesized from a mux with selector logic.

```
module mux_logic (output y, input select, sig_G, sig_max, sig_a, sig_b);
 assign y = (select == 1) || (sig_G == 1) || (sig_max == 0) ? sig_a : sig_b;
endmodule
```

End of Example 6.9

6.2.1 Synthesis of Priority Structures

A ***case*** statement implicitly attaches higher priority to the first item that it decodes than to the last one, and an ***if*** statement implies higher priority to the first branch than to the remaining branches. A synthesis tool will determine whether the case items of a ***case*** statement are mutually exclusive. If they are mutually exclusive, the synthesis tool will treat them as though they had equal priority and will synthesize a mux rather than a priority structure. Even when the list of case items is not mutually exclusive, a synthesis tool might allow the user to direct that they be treated without priority (e.g., Synopsys *parallel_case* directive). This would be useful if only one case item could be selected at a time in actual operation. An ***if*** statement will synthesize to a mux structure if the branching is specified by mutually exclusive conditions, as in Example 6.9, but when the branching is not mutually exclusive the synthesis tool will create a priority structure.

Example 6.10

The activity flow within *mux_4pri* is not governed by mutually exclusive conditions. This results in synthesis of an implied priority for datapath *a* because *sel_a* decodes *a* independently of *sel_b* or *sel_c*, shown in Figure 6-17. The event control expression of

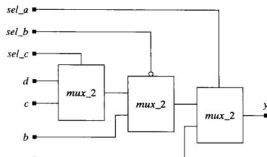


FIGURE 6-17 Circuit synthesized from a mux with priority decode of input conditions.

the cyclic behavior monitors all of the signals that are referenced within the behavior. Verilog 2001 introduced a wild-card token, *, that automatically creates a sensitivity list that is sensitive to all of the variables in the behavior. This feature ensures that latches are not accidentally synthesized.

```
module mux_4pri (output reg y, input a, b, c, d, sel_a, sel_b, sel_c);
 always @ (sel_a, sel_b, sel_c, a, b, c, d)
 // always @ (*)              // Optional wildcard token for complete sensitivity list
  if (sel_a == 1)  y = a;
   else if (sel_b == 0)     y = b;
    else if (sel_c == 1)    y = c;
     else y = d;
endmodule
```

End of Example 6.10

6.2.2 Exploiting Logical Don't-Care Conditions

When ***case***, conditional branch (***if***), or conditional assignment (*? ... :*) statements are used in a Verilog behavioral description of combinational logic, the behavioral model and the synthesized netlist should produce the same simulation results (except for time-dependent behavior) if the seed code has ***default*** assignments that are purely 0 or 1 values (i.e., the ***default*** does not explicitly assign an **x** or a **z** value). Simulation results may differ if the default or branch statement makes an explicit assignment of an **x** or a **z**. A synthesis tool will treat ***casex*** and ***casez*** statements as ***case*** statements. Those ***case*** items that decode to explicit assignment of **x** or **z** will be treated as don't-care conditions for the purpose of logic minimization of the equivalent Boolean expressions, (i.e., it does not matter what value is assigned to the object of the ***case***

assignment under those input conditions).[9] The physical hardware will propagate either a 0 or a 1, while the HDL model will propagate an **x** in simulation. This may lead to a mismatch between the results obtained by simulating the seed code and the synthesis product.

> **Synthesis Tip**
>
> An assignment to **x** in a ***case*** or an ***if*** statement will be treated as a don't-care condition in synthesis.

Example 6.11

The counter and seven-segment display shown in Figure 6-18(a) are modeled below, in Verilog, by *Latched_Seven_Seg_Display*, with active low outputs.[10] The counter increments while *Enable* is asserted. The display is blank while *Blanking* is asserted. If *Blanking* is not asserted the display shows only even-valued numbers, and remains latched if BCD is the code of an odd number or an invalid number. The waveforms for *Display_L* and *Display_R* are shown in Figure 6-18(b), with values corresponding to the codes for active-low seven-segment displays. The latching action is implemented by not having a ***default*** item in the ***case*** statement. When the level-sensitive behavior is activated by a change in *count*, the values of *Display_L* and *Display_R* change only when *count* decodes to an even number; otherwise *Display_L* and *Display_R* remain unchanged (latched).

```
module Latched_Seven_Seg_Display (
  output reg [6: 0] Display_L, Display_R,
  input Blanking, Enable, clock, reset
);
  reg [3: 0] count;
  //                      abc_defg
  parameter BLANK =       7'b111_1111;
  parameter ZERO =        7'b000_0001;    // h01
  parameter ONE =         7'b100_1111;    // h4f
  parameter TWO =         7'b001_0010;    // h12
  parameter THREE =       7'b000_0110;    // h06
```

[9] See Examples 5.22 (8-bit encoder), 5.23 (8:3 priority encoder), and 5.24 (3:8 decoder) in Chapter 5 for circuits synthesized from level-sensitive cyclic behaviors using default assignments in ***case*** and ***if*** statements.
[10] The underscore character is used in the parameters of *Latched_Seven_Seg_Display* to make the representation of a number more readable.

```
parameter FOUR =      7'b100_1100;    // h4c
parameter FIVE =      7'b010_0100;    // h24
parameter SIX =       7'b010_0000;    // h20
parameter SEVEN =     7'b000_1111;    // h0f
parameter EIGHT =     7'b000_0000;    // h00
parameter NINE =      7'b000_0100;    // h04
always @ (posedge clock)
  if (reset) count <= 0;
  else if (Enable) count <= count +1;

always @ (count, Blanking)
  if (Blanking) begin Display_L = BLANK; Display_R = BLANK; end else
    case (count)
      0: begin Display_L = ZERO; Display_R = ZERO; end
      2: begin Display_L = ZERO; Display_R = TWO; end
      4: begin Display_L = ZERO; Display_R = FOUR; end
      6: begin Display_L = ZERO; Display_R = SIX; end
      8: begin Display_L = ZERO; Display_R = EIGHT; end
      10: begin Display_L = ONE; Display_R = ZERO; end
      12: begin Display_L = ONE; Display_R = TWO; end
      14: begin Display_L = ONE; Display_R = FOUR; end
      //default: begin Display_L = BLANK; Display_R = BLANK; end
    endcase
endmodule
```

End of Example 6.11

The absence of ***default*** assignments in the level-sensitive cyclic behavior in *Latched_Seven_Seg_Display* latches the output and will cause latches to be implemented by a synthesis tool. When default assignments are unconstrained, the synthesis tool can exploit them as don't-care conditions and reduce the logic needed to implement the circuit. The next example demonstrates how to exploit the don't-cares and how to form three-state outputs.

Synthesis Tip

If a conditional operator assigns the value z to the right-hand side expression of a continuous assignment in a level-sensitive behavior, the statement will synthesize to a three-state device driven by combinational logic.

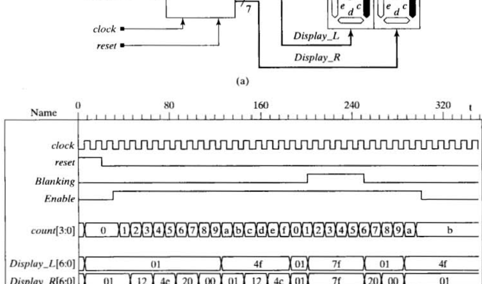


FIGURE 6-18 Seven-segment LED display: (a) counter and display units and (b) simulation results showing the action of *reset*, *Blanking*, and *Enable*. The values of *Display_L* and *Display_R* correspond to the 7-bit codes of the active-low seven-segment display units for the values of *count*.

Example 6.12

The level-sensitive cyclic behavior in *alu_with_z1* describes the combinational logic of a simple arthimetic and logical unit (ALU), and a continuous assignment describes a three-state output. The ***default*** assignments of the ***case*** statement constrain the output of the ALU to be 0. A second version, *alu_with_z2*, which is not shown, has the same description, except the ***default*** assignments of the ***case*** statement are don't-cares (4'*bx*) instead of 0. The synthesized circuits are shown in Figure 6-19. Both have three-state output inverters (which take less area than buffers), but the circuit for *alu_with_z2* has a simpler realization, because the synthesis tool is able to exploit the don't-cares implied by the ***default*** assignment of 4'*bx* in the ***case*** statement.

```
module alu_with_z1 (
  output alu_out,
  input [3: 0] data_a, data_b,
  input [2: 0] opcode,
  input enable
);
 reg      [3: 0]     alu_reg;
 assign  alu_out = (enable == 1) ? alu_reg : 4'bz;
 always @ (opcode,  data_a,  data_b)
  case  (opcode)
   3'b001:         alu_reg = data_a | data_b;
   3'b010:         alu_reg = data_a ^ data_b;
   3'b110:         alu_reg = ~data_b;
   default:        alu_reg = 4'b0;           // alu_with_z2 has default: alu_reg = 4'bx;
  endcase
endmodule
```

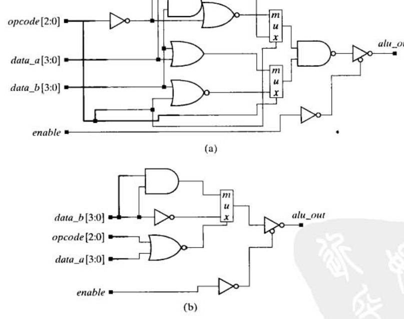


FIGURE 6-19 Circuits synthesized from (a) *alu_with_z1* and (b) *alu_with_z2*.

End of Example 6.12

6.2.3 ASIC Cells and Resource Sharing

An ASIC cell library usually contains cells that are more complex than combinational primitive gates.[11] For example, most libraries will contain a model for a full-adder cell. Whether the synthesis tool exploits the available model or builds another circuit depends on the designer's Verilog description. The tool must share resources as much as possible to minimize needless duplication of circuitry.

Synthesis Tip

Use parentheses to control operator grouping and reduce the size of a circuit.

Example 6.13

A synthesis tool mapped the addition operator in the Verilog description below to a full-adder ASIC library cell to create the circuit shown in Figure 6-20(a). An alternative implementation shown in Figure 6-20(b) builds two different 5-bit adder blocks out of basic library cells in a structure that depends on the speed goal for the design (details not shown). The *esdpupd* device provides 0 (or 1) where needed. The leftmost adder forms $A[3:0] + B[3:0]$ The 5-bit result is one of the inputs to the right-most adder block. That block has a second 5-bit input formed by the *C_in bit* and four 0s. The carry-in of each block is hard-wired to ground.

```
module badd_4 (output [3: 0] Sum, output C_out, input [3: 0] A, B, C_in);
 assign {C_out, Sum} = A + B + C_in;
endmodule
```

End of Example 6.13

A synthesis tool must recognize whether the physical resources required to implement complex (large area) behaviors can be shared. If the data flows within the behavior do not conflict, the resource can be shared between one or more paths. For example, the addition operators in the continuous assignment below are in mutually exclusive datapaths and can be shared in hardware.

[11]The fabrication masks of the complex cells create a more efficient and faster implementation of the functionality than an aggregate of simpler cells with equivalent functionality.

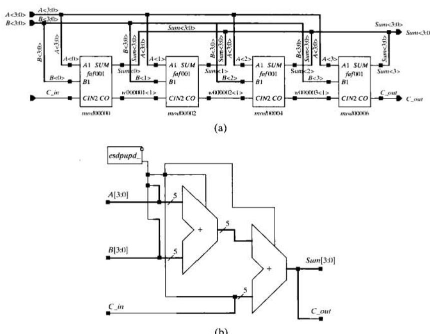


FIGURE 6-20 Result of synthesizing the + operator (a) using library full-adder cells, and (b) using 5-bit adder blocks with hard-wired inputs to accommodate 4-bit datapaths.

```
assign y_out = sel ? data_a + accum : data_a + data_b;
```

Consequently, the operators can be implemented by a shared adder whose input datapaths are multiplexed. This feature is vendor-dependent. If the tool does not automatically implement resource sharing, the description must be written to force the sharing.

Example 6.14

The use of parentheses in the description in *res_share* forces the synthesis tool to multiplex the datapaths and produce the circuit shown in Figure 6-21.

```
module res_share (output [4: 0] y_out, input [3: 0] data_a, data_b, accum, input sel);
  assign y_out = data_a + (sel ? accum : data_b);
endmodule
```

Failure to include the parentheses in the expression for *y_out* in *res_share* will lead to synthesis of a circuit that uses two adders. The most efficient implementation

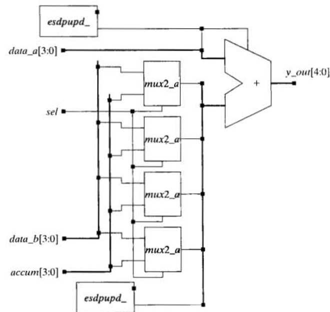


FIGURE 6-21 Implementation of a datapath with shared resources.

multiplexes the datapaths and shares the adder between them, rather than multiplex the outputs of separate adders. The important design trade-off is that the mux will occupy significantly less area than the adder that it replaces.

End of Example 6.14

As a general guideline, if arithmetic functions are to be inferred from language-based operators, the operators should be grouped as much as possible within a single cyclic behavior to allow the synthesis engine to share the hardware resources that will be used to implement the function.

6.3 Synthesis of Sequential Logic with Latches

Latches are synthesized in two ways: intentionally and accidentally. Latches that are synthesized accidentally waste silicon and may compromise the functionality of the circuit. It is important to understand the conditions under which a synthesis tool infers a latch from a Verilog description. A designer who does not understand this relationship will be surprised when what was intended to be combinational logic is synthesized as latched logic. The topic is important enough, in practice, to warrant an elaborate treatment, at the risk of belaboring the point and being verbose.

We have discussed three common ways to describe synthesizable combinational logic: (1) netlist of primitives, (2) Boolean equations described by continuous-assignment statements, and (3) a level-sensitive cyclic behavior. Can any or all of these styles lead to a circuit with latches?

Synthesis Tip

A feedback-free netlist of combinational primitives will synthesize into latch-free combinational logic.

Feedback-free netlists of combinational primitives will synthesize into latch-free combinational logic. Synthesis tools may not allow a netlist of primitives to have feedback (e.g., cross-coupled NAND gates), so such a description will be flagged as an error condition and the code will not synthesize into anything.

A set of continuous assignments describing combinational logic must not have feedback among them. For example, the pair of continuous assignments in Section 5.5 describes a NAND-latch, but this model will not synthesize because it has structural feedback.[12]

Synthesis Tip

A set of feedback-free continuous assignments will synthesize into latch-free combinational logic.

A continuous assignment that uses the conditional operator (**? :**) with feedback (i.e., the target variable of the assignment appears in an expression of the operator) will synthesize to a latch. This is one way to intentionally model a latch, such as static random access memory (SRAM),[13] and it will synthesize.

Synthesis Tip

A continuous assignment using a conditional operator with feedback will synthesize into a latch.

Example 6.15

The output of a cell of an SRAM memory can be modeled by the following continuous assignment statement having feedback:

```
assign data_out = (CS_b == 0) ? (WE_b == 0) ? data_in : data_out : 1'bz;
```

[12]Synthesis tools may not infer latches from structural feedback loops.

[13]We will consider SRAMs in more detail in Chapter 8.

The signals *CS_b* and *WE_b* implement the active-low chip-select and write-enable functions of the cell. If the chip is selected and *WE_b* == 0, *data_out* follows *data_in* (transparent mode), but when *WE_b* switches to 1, *data_out* = *data_out* (latched mode). Synthesis tools infer the behavior of a latch from this statement because the output of the device is not affected by *data_in* while *WE_b*=1, and *data_out* holds the residual value that it had at the moment *WE_b* switched to 1. If *CS_b* == 1, the cell is in the three-state, high-impedance condition.

End of Example 6.15

6.3.1 Accidental Synthesis of Latches

Example 6.16

The model *or4_behav* describes a four-input OR gate. The algorithm within the cyclic behavior initializes the output to 0, then tests the inputs sequentially. If an input is 1, the output is set to 1, and the sequence terminates. The description synthesizes to combinational logic. The loop-index variable, *k*, is eliminated by the synthesis process and has no hardware counterpart. Note that the output variable, *y*, was declared as a register variable but does not synthesize into a storage element.[14]

```
module or4_behav #(parameter word_length = 4)(
  output reg y,
  input [word_length -1: 0] x_in
);
  integer k;

 always @ x_in begin: check_for_1
  y = 0;
  for (k = 0; k <= word_length -1; k = k+1)
   if (x_in[k] == 1) begin
    y = 1;
    disable check_for_1;
   end
 end
endmodule
```

Now consider the description *or4_behav_latch*, whose event control expression is not sensitive to *x_in*[0]. This leads to synthesis of a latched output—the latch implements

[14]Users of Verilog who do not realize that *all* the variables that are assigned value by a procedural assignment are register variables will be mystified by the results of synthesis if they assume that every register variable will synthesize to a flip-flop.

the functionality that is implied by the description's ignoring *x_in*[0] and assigning value to the output only when the cyclic behavior is activated by a change in *x_in*[3: 1].

```
module or4_behav_latch #(parameter word_length = 4)(
  output reg y, input [word_length -1: 0] x_in
);
  integer k;
  always @ (x_in[3: 1])   begin: check_for_1          // incomplete sensitivity list
    y = 0;
    for (k = 0; k <= word_length -1; k = k + 1)
      if (x_in[k] == 1) begin
        y = 1;
        disable check_for_1;
      end
    end
  endmodule
```

The circuits that implement the functionality of *or4_behav* and *or4_behav_latch* are shown in Figure 6-22(a,b). The value of *x_in*[0] is latched in *or4_behav_latch*, and

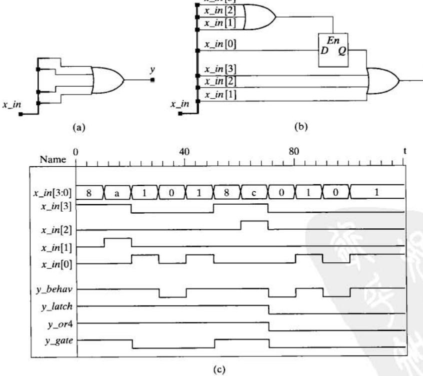


FIGURE 6-22 Four-input OR gate: (a) circuit synthesized from a level-sensitive cyclic behavior, (b) synthesized from a latch-inducing model, and (c) simulation results for both circuits.

the latch is controlled by OR-ing *x_in*[1], *x_in*[2], and *x_in*[3]. If any of these inputs is 1, the value of *x_in*[0] is passed to the OR gate forming *y*. But if these inputs all switch to 0, the value of *x_in*[0] is latched and the value of *y* is independent of changes in *x_in*[0]. The simulation results in Figure 6-22(c) show the waveforms produced by simulation of *or4_behav, or4_behav_latch, y_gate* (the response of the gate-level circuit in Figure 6-22(b), and *y_or3_gate*, a three-input OR gate driven by *x_in*[1], *x_in*[2], and *x_in*[3]. The waveforms demonstrate the latching behavior of *or4_behav_latch*.

Level-sensitive cyclic behaviors will synthesize into combinational logic if the description does not imply the need for storage. If storage is implied by the model, then a latch will be introduced into the implementation. How is the need for storage inferred from the Verilog description? To avoid latches, all of the variables that are assigned value by the behavior must be assigned value under all events that affect the right-hand side expressions of the assignments that implement the logic. Failure to do so will produce a design with unwanted latches. Consequently, all inputs to a level-sensitive behavior that is to implement combinational logic must be included in the event control expression. The right-hand side operands of assignments within the behavior are inputs. Likewise, any control signals whose transitions affect the assignments to the target register variables in the behavior are considered to be inputs to the behavior.

Remember: If the sensitivity list of a level-sensitive behavior is not complete, some variable will not be assigned value under every condition of the inputs, implying the need for memory. In fact, the risk of failing to have a complete sensitivity list has been found to be so great that the language was expanded to include the wildcard token (*) to implicitly form a complete sensitivity list and relieve the designer of the responsibility for its completeness. The constructs @ (*) and @* can be used in place of an itemized sensitivity list for a level-sensitive behavior.

Signals that appear as operands on the right-hand side of any assignment in a level-sensitive cyclic behavior may not appear on the left-hand side of the expression. If this rule is not observed, the behavior has implicit feedback and will synthesize into a latch rather than feedback-free combinational logic [3].

End of Example 6.16

Synthesis Tip

A Verilog description of combinational logic must assign value to the outputs for all possible values of the inputs.

Verilog ***case*** statements and conditionals (***if***) that do not include all possible cases or conditions are *incompletely specified* and may lead to synthesis of unwanted latches in the design because they admit the possibility that a variable will not be assigned value under all possible conditions of the inputs (i.e., the description implies that the output should retain its residual value under the conditions that were left unspecified). Caution must be taken to ensure that ***case*** and conditional branching (***if***) statements are complete, either explicitly or by default. If an expression associated with a conditional operator in a continuous assignment assigns the target variable (left-hand side [LHS]) to itself,

the statement will synthesize a latch, but an incomplete conditional operator will cause a syntax error.[15]

Example 6.17

When a ***case*** statement is incompletely decoded, a synthesis tool will infer the need for a latch to hold the residual output when the select bits take the unspecified case item values. The latch is enabled by the event-or of the cases under which the assignment is explicitly made. In this example, the latch[16] is enabled *when* {*sel_a*, *sel_b*} == 2'*b*10 *or* {*sel_a*, *sel_b*} == 2'*b*01. Figure 6-23(a) shows a generic implementation, and Figure 6-23(b) shows an implementation using an actual cell library. The latter uses the library's 2-channel mux, and a model (*esdpupd*) for an electrostatic discharge pull-up/down device, which disables the hardware latch's active-low reset, rather than

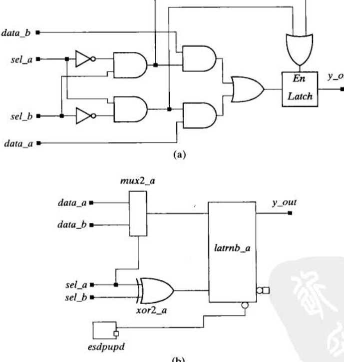


FIGURE 6-23 Two-channel mux with latched output synthesized from an incompletely specified case statement using (a) generic parts and (b) parts from a cell library.

[15]The syntax of the conditional operator (? . . . : . . .) requires a statement for the true condition and the false condition.

[16]Appendix G describes the various latches and flip-flops shown in synthesized circuits.

letting it float. Its internal structure consists of an instantiation of the ***pullup*** and ***pulldown*** primitives. Its outputs are the outputs of the two primitives. In this example the output of the ***pullup*** primitive is connected to the latch's reset input.

```
module mux_latch (output reg y_out, input data_a, data_b, sel_a, sel_b);
 always @ (sel_a, sel_b, data_a, data_b)
  case ({sel_a, sel_b})
    2'b10: y_out = data_a;
    2'b01: y_out = data_b;
  endcase
endmodule
```

End of Example 6.17

6.3.2 Intentional Synthesis of Latches

Threads of execution in a level-sensitive behavior that do not explicitly assign value to a register variable imply the need for the variable to retain the value it had before the behavior was activated. In general, a level-sensitive behavior may contain several such variables, and all of them will be synthesized as latches. The synthesis tool must identify the datapaths through the latches and identify their control signals. The control signal of a given latch will be the signal whose value controls the branching of the activity flow to the statements that do not assign value to the associated register variable. If the activity flow assigns value to a given register variable in all possible threads of the activity, a latch will be inferred only if a path assigns a variable its own value (i.e., self-feedback); otherwise, in the absence of feedback, the behavior does not imply a latch.

What determines the hardware for a latch? In synthesis, latches implement incompletely specified assignments to register variables in ***case*** and conditional branch (***if***) statements in a level-sensitive cyclic behavior. If a ***case*** statement has a ***default*** assignment with feedback (i.e., the variable is explicitly assigned to itself), the synthesis tool will form a mux structure with feedback. Likewise, if an ***if*** statement in a level-sensitive behavior assigns a variable to itself, the result will be a mux structure with feedback.

Memory is inferred when the conditional operator (? ... : ...) is implemented with feedback, but the actual hardware structure chosen by a synthesis tool depends on the context. If the conditional operator is used in a continuous assignment, the result will be a mux with feedback. If it is used in a level-sensitive cyclic behavior, the result will be a hardware latch. If the conditional operator is used in an edge-sensitive cyclic behavior, the result will be a register with a gated data path in a feedback configuration with the output of the register.

Example 6.18

The level-sensitive cyclic behavior in *latch_if*1 below has a complete sensitivity list, and assigns *data_out* with feedback in an ***if*** statement having feedback. The result of

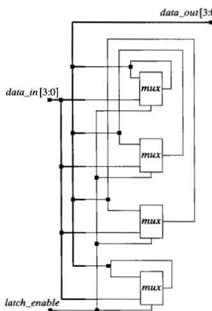


FIGURE 6-24 Latched circuit synthesized from *latch_if1*.

synthesizing the latch circuit is shown in Figure 6-24, which has the structure of a mux with feedback.[17]

```
module latch_if1(output reg [3: 0] data_out, input [3: 0] data_in, input latch_enable);
  always @ (*)
    if (latch_enable) data_out <= data_in;
      else data_out <= data_out;
endmodule
```

End of Example 6.18

[17]Note the use of the nonblocking assignment operator (<=). Intentional synthesis of a latch is one exception to the general rule of using blocking assignments (=) in level-sensitive behaviors and nonblocking assignments (<=) in edge-sensitive behaviors. The exception conforms to the IEEE Standard 1364.1 for Verilog Register Transfer Level Synthesis.

Synthesis Tip

An ***if*** statement in a level-sensitive behavior will synthesize to a latch if the statement assigns value to a register variable in some, but not all, branches (i.e., the statement is incomplete).

Example 6.19

The event control expression of the level-sensitive behavior in *latch_if2* is sensitive to both *latch_enable* and *data_in*, but the ***if*** statement is incomplete. This descriptive style maps preferentially to a hardware latch, shown in Figure 6-25, rather than a feedback-mux configuration.

```
module latch_if2 (output reg [3: 0] data_out, input [3: 0] data_in, input latch_enable);
  always @ (latch_enable, data_in)
    if (latch_enable) data_out = data_in;  // Incompletely specified
endmodule
```

End of Example 6.19

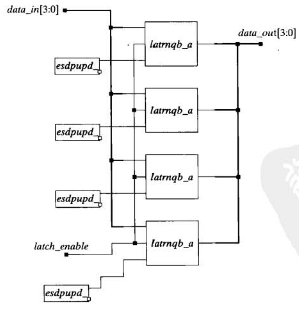


FIGURE 6-25 Latches synthesized from *latch_if2*, a description containing an incompletely specified conditional branch (*if*) statement.

The synthesis results in Figures 6-24 and 6-25 illustrate how a slight change in the behavioral description can influence the structure of the synthesized circuit. The structure in Figure 6-24 synthesized from a complete conditional branch with feedback, whereas the description in Figure 6-25 synthesized from an incomplete conditional branch. The circuits are equivalent in simulation, but will have different area/speed trade-offs in hardware. An ***if*** statement that is completed with feedback is equivalent to the following conditional assignment statement:

```
assign data_out [3: 0] = latch_enable ? data_in [3: 0] : data_out[3: 0];
```

This statement will synthesize to the same structure, and it is commonly used to describe a latch. Remember, the conditional operator must be completed with two expressions, one for the true condition, and the other for the false condition.

Example 6.20

An *sn54170* register file consists of an array of four 4-bit words. Two address busses, *wr_sel* and *rd_sel* provide addresses for write and read operations. Two enable lines, *wr_enb* and *rd_enb* control transparent-low latches. The level-sensitive behavior in *sn54170* has an incomplete branching statement. The synthesized circuit in Figure 6-26 has an array of latches holding *latched_data*.

```
module sn54170 (
  output [3: 0] data_out,
  input [3: 0] data_in, input [1: 0] wr_sel, rd_sel, input wr_enb, rd_enb
);
  reg [3: 0] latched_data;
  always @ (wr_enb, wr_sel, data_in) begin
   if (!wr_enb) latched_data[wr_sel] = data_in;
  end
  assign data_out = (rd_enb) ? 4'b1111 : latched_data[rd_sel];
endmodule
```

End of Example 6.20

Incomplete **case** and conditional (**if**) statements in an edge-sensitive behavior synthesize register variables to flip-flops; if the statements are completed with feedback, the result is a register whose output is fed back through a mux at its datapath. (If the cell library has a cell with a gated datapath, the tool will select that part.)

6.4 Synthesis of Three-State Devices and Bus Interfaces

Three-state devices allow buses to be shared among multiple devices. The preferred style for inferring a three-state bus driver uses a continuous assignment statement that has one branch set to a three-state logic value (*z*).

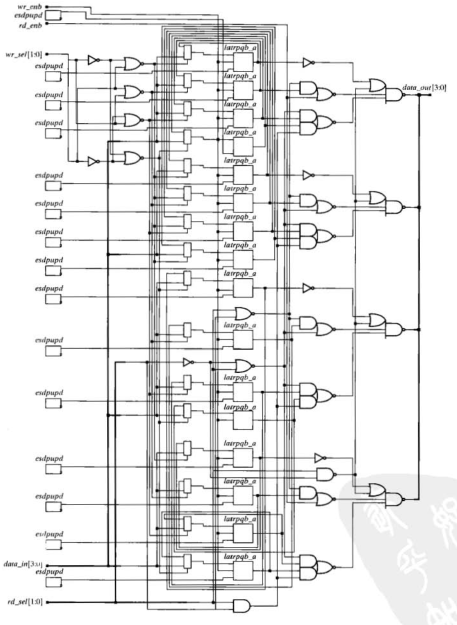


FIGURE 6-26 Circuit synthesized from a Verilog model of the *sn54170* register file.

Example 6.21

The circuit shown in Figure 6-27 is a typical configuration for integrating some core logic with a three-state unidirectional interface to a bus. Because an expression of the conditional assignment in *Uni_dir_bus* assigns *32'bz* to *data_to_bus*, a synthesis tool will infer a 32-bit wide output with three-state drivers.

```
module Uni_dir_bus (output [31: 0] data_to_bus, input bus_enabled);
  reg [31: 0]  ckt_to_bus;
  assign data_to_bus = (bus_enabled) ? ckt_to_bus : 32'bz;

// Description of core circuit goes here to drive ckt_to_bus

endmodule
```

End of Example 6.21

Example 6.22

The circuit shown in Figure 6-28 has a bidirectional interface with an external bus. The port of the circuit is declared to be bidirectional (***inout***) and a pair of continuous assignments is used to model the inbound and outbound datapaths.

```
module Bi_dir_bus (inout [31: 0] data_to_from_bus, input send_data, rcv_data);
  wire [31: 0] ckt_to_bus;
  wire [31: 0] data_from_bus;
  assign data_from_bus = (rcv_data) ? data_to_from_bus : 32'bz;
  assign data_to_from_bus = (send_data) ? ckt_to_bus : data_to_from_bus;
// Behavior using data_from_bus and generating ckt_to_bus goes here
endmodule
```

End of Example 6.22

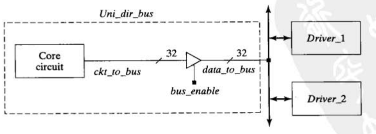


FIGURE 6-27 Unidirectional interface to a bus.

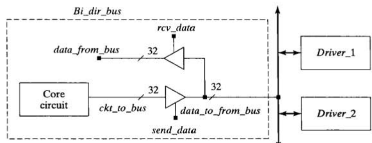


FIGURE 6-28 Bidirectional interface to a bidirectional bus.

6.5 Synthesis of Sequential Logic with Flip-Flops

Flip-flops are synthesized only from edge-sensitive cyclic behaviors, but not every register variable that is assigned value in an edge-sensitive behavior synthesizes to a flip-flop. What determines the outcome of synthesis from an edge-sensitive behavior? This section will address this and the following related questions: How does a synthesis tool infer the need for a flip-flop? Or when does a register variable that is assigned value within an edge-sensitive behavior automatically synthesize to a flip-flop? A flip-flop is synchronized by a clock signal—how does the synthesis tool distinguish the synchronizing signal from other signals, or must it be identified by a special word, like "clock"? If a design has multiple flip-flops, how can the model be written to ensure that they function concurrently?

A register variable in an edge-sensitive behavior will be synthesized as a flip-flop (1) if it is referenced outside the scope of the behavior, (2) if it is referenced within the behavior before it is assigned value, or (3) if it is assigned value in only some of the branches of the activity within the behavior. All of these situations imply the need for memory, i.e., the residual value of the variable. The fact that these conditions occur in an edge-sensitive behavior dictates that the memory be a flip-flop, rather than a latch.

Recall that an incomplete conditional statement (i.e., a ***if*** ... ***else*** statement or a ***case*** statement) in a level-sensitive cyclic behavior will synthesize to a latch. However, if the behavior is *edge*-sensitive, these types of statements will *not* create latches, but they will synthesize logic that implements a "clock enable," because the *incomplete* statements imply that the affected variables should not change under the conditions implied by the logic, even though the clock makes a transition.

The sequence in which signals are decoded in the statement that follows the event-control expression of an edge-sensitive cyclic behavior determines which of the edge-sensitive signals are control signals and which is the clock (i.e., the synchronizing signal). If the event-control expression is sensitive to the edge of more than one signal, an ***if*** statement must be the first statement in the behavior. The control signals that appear in the event-control expression must be decoded explicitly in the branches of the ***if*** statement (e.g., decode the reset condition first). The synchronizing signal is not tested explicitly in the body of the ***if*** statement, but, by default, the last branch must describe the synchronous activity, independently of the actual names given to the signals.

Synthesis Tip

A variable that is referenced within an edge-sensitive behavior before it is assigned value in the behavior will be synthesized as the output of a flip-flop.

Example 6.23

The nonblocking assignments to *data_a* and *data_b* in *swap_synch* describe a synchronous data swapping mechanism. The statements assigning value to *data_a* and *data_b* execute nonblocking assignments concurrently, so both variables are sampled (referenced) before receiving value; both are synthesized as the output of a flip-flop in Figure 6-29. Note that *set1* and *set2* are explicitly decoded first. The last clause of the ***if*** statement assigns values to *data_a* and *data_b*. Those (nonblocking) assignments are synchronized to the rising edge of *clk*, which is not referenced explicitly in the ***if*** statement.

```
module swap_synch (output reg data_a, data_b, input set1, set2, clk);
 always @ (posedge clk) begin
  if (set1) begin data_a <= 1; data_b <= 0; end else
  if (set2) begin data_a <= 0; data_b <= 1; end
  else begin
   data_b <= data_a;
   data_a <= data_b;
  end
 end
endmodule
```

End of Example 6.23

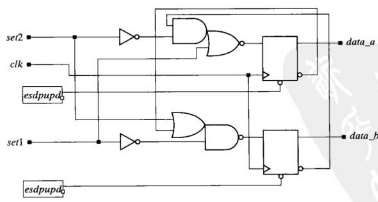


FIGURE 6-29 Circuit synthesized for *swap_synch*, with variables referenced before being assigned value.

Example 6.24

The functionality of a 4-bit parallel-load data register is described by *D_reg4_a*. The positive edge of *reset* appears in the event control expression and appears in the first clause of the ***if*** statement; the positive edge of *clock* also appears in the event-control expression, but is not explicitly decoded by a branch of the "*if*" statement that follows the event-control expression. This enables the synthesis tool to correctly infer the need for a resettable flip-flop, active on the positive edge of *clock*. The value of *Data_out* is synchronized to the positive edge of *clock*, so the synthesis tool creates the 4-bit array of flip-flops shown in Figure 6-30.

```
module D_reg4_a (output reg [3: 0] Data_out, input [3: 0] Data_in, input clock, reset);
 always @ (posedge clock, posedge reset) begin
  if (reset == 1'b1) Data_out <= 4'b0;
  else Data_out <= Data_in;
 end
endmodule
```

End of Example 6.24

In general, the event-control expression of a cyclic behavior describing sequential logic must be synchronized to a single edge (***posedge*** or ***negedge***, but not both) of a single clock (synchronizing signal). Multiple behaviors need not have the same synchronizing signal, or be synchronized by the same edge of the same signal, but the optimization process requires that all of the synchronizing signals (clocks) have the same period. Otherwise, it would not be possible to optimize the performance of the logic.

Synthesis Tip

A variable that is assigned value by a cyclic behavior before it is referenced within the behavior, but is not referenced outside the behavior, will be eliminated by the synthesis process.

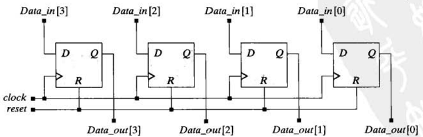


FIGURE 6-30 Synthesis circuit for *D_reg4_a*, a 4-bit parallel-load data register.

It is important to realize that not every register variable that is assigned value in a cyclic behavior synthesizes to a hardware storage device. The 4-input OR gate that was described by a Verilog behavior (*or4_behav*) in Example 6.16 has a register variable, *k*, that is used within the behavior, but is not referenced outside the behavior. It is not referenced before it is assigned value. The variable *k* merely provides an index within the algorithm and does not require hardware memory—it does not have a life beyond the computation performed by the behavior. It is eliminated by synthesis. The behavior correctly synthesizes to a hardware OR gate, without a memory element.

Synthesis Tip

A variable that is assigned value by an edge-sensitive behavior and is referenced outside the behavior will be synthesized as the output of a flip-flop.

Example 6.25

The behavior in *empty_circuit* assigns value to the register variable *D_out*, but *D_out* is not referenced outside the scope of the behavior. Consequently, a synthesis tool will eliminate *D_out*. If *empty_circuit* is modified to declare *D_out* as an output port, *D_out* will be synthesized as the output of a flip-flop.

```
module empty_circuit (input D_in, clk);
 reg D_out;
 always @ (posedge clk) begin
   D_out <= D_in;
 end
endmodule
```

End of Example 6.25

6.6 Synthesis of Explicit State Machines

Explicit state machines have an explicitly declared state register and explicit logic that governs the evolution of the state under the influence of the inputs. In this section, we will present a recommended style for describing explicit state machines, will use that style to write Verilog models of the explicit state machines that were designed by manual methods in Chapter 3, and then synthesize their implementation in hardware. It is recommended, for clarity, that explicit machines be described by two behaviors, an edge-sensitive behavior that synchronizes the evolution of the state and a level-sensitive behavior that describes the next state and output logic.

6.6.1 Synthesis of a BCD-to-Excess-3 Code Converter

The circuit for a Mealy-type binary-coded decimal (BCD)-to-Excess-3 code converter was designed by manual methods in Example 3.2 to illustrate the basic steps of (1) forming a STG, (2) defining a state table, (3) choosing a state assignment, (4) encoding the next state and output table, (5) developing and minimizing the Karnaugh maps for the encoded state bits and output, and (6) creating the circuit's implementation (schematic). Here we will revisit the code converter and demonstrate the simplicity of modeling and synthesizing it with Verilog.

Beginning with the STG (see Figure 3-19), we write the behavioral model, *BCD_to_Excess_3b*, and verify that it has the same functionality as the previous design. Note that a significant amount of work is required merely to change the state assignment in the original (manual) design, but the behavioral model requires only a change to the declared parameters defining the state codes. A synthesis tool will automatically reflect the change in the resulting gate-level structure.

Synthesis Tip

Use two cyclic behaviors to describe an explicit state machine: (1) a level-sensitive behavior to describe the combinational logic for the next state and outputs and (2) an edge-sensitive behavior to synchronize the state transitions.

The model, *BCD_to_Excess_3b*, has two cyclic behaviors.[18] An edge-sensitive behavior describes the state transitions and a level-sensitive behavior describes the next state and output logic. Note that the procedural assignments in the edge-sensitive behavior are nonblocking and that those in the level-sensitive behavior are blocked assignments. The Verilog language specifies that nonblocking assignments and blocking assignments that are scheduled to occur in the same time step of simulation execute in a particular order. The nonblocking assignments are sampled first, at the beginning of the time step (before any assignments are made) and then the blocked assignments are executed. After the blocked assignments are executed, the nonblocking assignments are completed by assigning to the left-hand side of the statements the values that were determined by the sampling at the beginning of the time step. This mechanism ensures that nonblocking assignments execute concurrently, independent of their order, and that race conditions cannot propagate

[18]It is sometimes convenient to use continuous assignments to describe the output combinational logic.

through blocked assignments and thereby affect the nonblocking assignments. Nonblocking assignments describe concurrent synchronous register transfers in hardware.

Synthesis Tip

Use the blocking procedural assignment operator (=) in the level-sensitive cyclic behaviors describing the combinational logic of a FSM.

Matching simulation results between a behavioral model and a synthesized circuit does not guarantee that an implementation of the circuit is correct. The waveforms in Figure 6-31 were obtained by simulating *BCD_to_Excess_3b*; they match those of the manually designed gate-level model in Figure 3-23. However, note that *BCD_to_Excess_3b* does not include a ***default*** assignment in its ***case*** statement. This induces latches in the synthesized circuit shown in Figure 6-32(a). On the other hand, with *don't-care* default assignments to *next_state* and *B_out*, *BCD_to_Excess_3c* synthesizes to the circuit in Figure 6-32(b), without latches. Figure 6-33(c) shows the simulation results for both circuits. The waveforms of both circuits match those of the manual design (Figure 3-23) because the testbench exercises the circuit over only the allowable input sequences. The don't-care assignments of *BCD_to_Excess_3c* give greater flexibility to the synthesis tool than the implied latch structure of *BCD_to_Excess_3b*, thus it is advisable to include ***default*** assignments in all ***case***

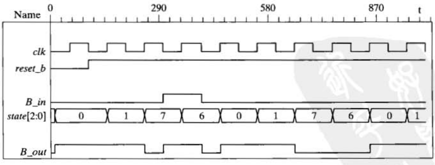


FIGURE 6-31 Results obtained from simulation of *BCD_to_Excess_3b*, a Verilog behavioral model of a BCD-to-Excess-3 code converter.

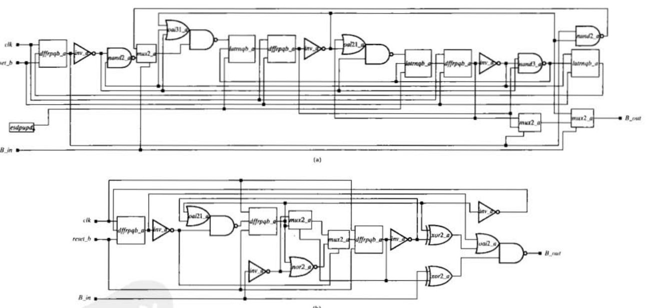


FIGURE 6-32 ASIC circuits synthesized from (a) *BCD_to_Excess_3b*, and (b) *BCD_to_Excess_3c*. Note that *BCD_to_Excess_3b* has latched circuitry due to omission of ***default*** assignments in the ***case*** statement.

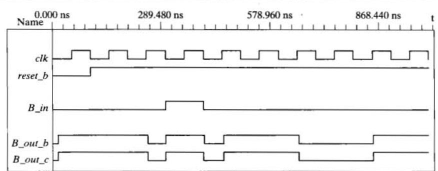


FIGURE 6-33 Post-synthesis simulation of the ASIC circuits synthesized from *BCD_to_Excess_3b* and *BCD_to_Excess_3c*.

statements. Additionally, the latches in *BCD_to_Excess_3b* waste hardware and silicon area.

> **Synthesis Tip**
>
> Use the nonblocking assignment operator (<=) in the edge-sensitive cyclic behaviors describing the state transitions of a FSM and the register transfers of the datapath of a sequential machine.

```
module BCD_to_Excess_3b (output reg B_out, input B_in, clk, reset_b);
 parameter              S_0 = 3'b000,              // State assignment
                        S_1 = 3'b001,
                        S_2 = 3'b101,
                        S_3 = 3'b111,
                        S_4 = 3'b011,
                        S_5 = 3'b110,
                        S_6 = 3'b010,
                        dont_care_state = 3'bx,
                        dont_care_out = 1'bx;
 reg         [2: 0]     state, next_state;

 always @ (posedge clk, negedge reset_b)
  if (reset_b == 0) state <= S_0; else state <= next_state;

 always @ (state, B_in) begin
  B_out = 0;                     // Default assignment; assign by exception
  case (state)
   S_0:     if (B_in == 0) begin next_state = S_1; B_out = 1; end
            else if (B_in == 1) next_state = S_2;

   S_1:     if (B_in == 0) begin next_state = S_3; B_out = 1; end
            else if (B_in == 1) begin next_state = S_4; end
```

```
        S_2:    begin next_state = S_4; B_out = B_in; end
        S_3:    begin next_state = S_5; B_out = B_in; end
        S_4:    if (B_in == 0) begin next_state = S_5; B_out = 1; end
                else if (B_in == 1) begin next_state = S_6; end
        S_5:    begin next_state = S_0; B_out = B_in; end
        S_6:    begin next_state = S_0; B_out = 1; end
        /* Omitted for BCD_to_Excess_3b version
           Included for BCD_to_Excess_3c version
        default: begin next_state = dont_care_state; B_out = dont_care_out; end
        */
        endcase
      end
    endmodule
```

Synthesis Tip

Decode all possible states in a level-sensitive behavior describing the combinational next state and output logic of an explicit state machine.

If the states listed as case items in a ***case*** statement do not decode all possible states in the level-sensitive behavior describing the next state and output logic of a state machine, the combinational logic describing the next state and the output will be synthesized as the outputs of latches, the circuit may have more hardware (wastes silicon) than needed, and it may not function as intended. This outcome can be prevented by making a default assignment to the next_state at the beginning of the level-sensitive behavior that describes the next state. Likewise, default assignments can be assigned to the outputs of the machine to avoid their being synthesized as the outputs of latched.

Synthesis tools impose some additional restrictions on how state machines can be modeled. The state register of an explicit state machine must be assigned value as an aggregate, i.e., bit select and part select assignments to the state register variable are not allowed by a synthesis tool. The entire register must be assigned value. Asynchronous control signals (e.g., set and reset) must be scalars in the event-control expression of the behavior. Lastly, for synthesis, the value that is assigned to the state register must be either a constant (e.g., *state_reg = start_state*) or a variable that evaluates to a constant after static evaluation (i.e., the state-transition diagram must specify a fixed relationship). The description of *BCD_to_Excess_3b* satisfies these constraints.

A behavior describing the synchronous activity of an explicit state machine may contain only one clock-synchronized event-control expression. This rule applies whether the same or some other behavior describes the machine's next state and output. The description of an explicit state machine will also include an explicitly declared state register variable of type ***reg***. Only one such register may be identified for a machine, which implies that each assignment to the state register must assign value to the whole register, rather than to a bit select or a part select. The constraints on procedural assignments to the same register ensure that it is possible to associate a fixed-state transition diagram with the behavior.

6.6.2 Design Example: Synthesis of a Mealy-Type NRZ-to-Manchester Line Code Converter

A serial line converter that converts a non-return-to-zero (NRZ) bit stream into a Manchester encoded bit stream was designed by manual methods as a Mealy-type finite state machine in Chapter 3 (see Section 3.7.1). The same state machine is described here by a Verilog behavioral model, *NRZ_2_Manchester_Mealy*, shown below.

```
module NRZ_2_Manchester_Mealy (output reg B_out, input B_in, clock, reset_b);
  reg [1: 0] state, next_state;
  parameter       S_0 = 2'd0,
                  S_1 = 2'd1,
                  S_2 = 2'd2, // 2'd3 is unused bit pattern
                  dont_care_state = 2'bx,
                  dont_care_out = 1'bx;
  always @ (negedge clock, negedge reset_b)
   if (reset_b == 0) state <= S_0; else state <= next_state;
  always @ (state, B_in ) begin
   B_out = 0; // Default assignment
   case (state) // Note: State register is partially decoded
     S_0:          if (B_in == 0) next_state = S_1;
                   else if (B_in == 1) begin next_state = S_2; B_out = 1; end
     S_1:          begin next_state = S_0; B_out = 1; end
     S_2:          begin next_state = S_0; end
     default:      begin next_state = dont_care_state; B_out = dont_care_out; end
   endcase
  end
endmodule
```

The simulation results shown in Figure 6-34 match those shown in Figure 3-30 for the gate level (manual) design of the Mealy-type code converter.[19] We note again that *B_in* is switching on active edges of *clock_1* in Figure 6-34, which coincides with alternate active edges of *clock_2*. For those edges, *B_in* changes at the same time as the state. As a general rule, avoid having the inputs to a synchronous machine change at the same time that its state changes, unless it happens that the inputs are treated as don't-cares at those edges, as they are in this example. The netlist[20] and schematic of the circuit synthesized from *NRZ_2_Manchester_Mealy* are shown in Figure 6-35.

[19]Problem 2 at the end of this chapter addresses the postsynthesis verification step in which the functionality of the synthesized circuit is shown to match that of the behavioral model.

[20]The synthesis tool (i.e., Synopsys) generates names of wires and module instances using a more general naming convention supported by Verilog's *escaped identifiers*. These identifiers begin with a backslash (\) and end with white space; any printable ASCII character can be used in an escaped identifier. Here the instance names of the flip-flops correspond to the bits of the machine's state.

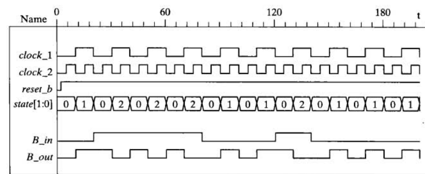


FIGURE 6-34 Results obtained from simulation of *NRZ_2_Manchester_Mealy*, a behavioral model of a Mealy-type FSM of an NRZ-to-Manchester serial line converter.

```
module NRZ_2_Manchester_Mealy (B_out, B_in, clock, reset_);
input B_in, clock, reset;
output B_out;
wire \next_state<1>, \next_state<0>, \state<1>, \state<0>, n80, n81, n82, n83;
  buff101 U26 (.A1(n81), .O(n80));
  norf201 U27 (.A1(n81), .B1(n82), .O(\next_state<1>));
  norf201 U28 (.A1(B_in), .B1(n80), .O(\next_state<0>));
  blf00101 U29 (.A1(n83), .B2(\state<1>), .(C2(n82), .O(B_out));
  nanf251 U30 (.A1(\state<1>), .B2(n83), .O(n81));
  invf101 U31 (.A1(B_in), .O(n82));
  invf101 U32 (.A1(\state<0>), .O(n83));
  dfrf301 \state_reg<1> (.DATA1(\next_state<1>), .CLK2(clock), .RST3(
     reset), .Q(\state<1>));
  dfrf301 \state_reg<0> (.DATA1(\next_state<0>), .CLK2(clock), .RST3(
     reset), .Q(\state<0>));
endmodule
```

(a)

B_in
clock
reset_b
dffrf301
nanf251
buff101
norf201
dffrf301
invf101
invf101
norf201
blf00101
B_out

(b)

FIGURE 6-35 ASIC circuit synthesized from *NRZ_2_Manchester_Mealy*: (a) netlist (shown with port connections made by names of formal and actual signals, and (b) schematic.

6.6.3 Design Example: Synthesis of a Moore-Type NRZ-to-Manchester Line Code Converter

The explicit FSM describing the Moore-type NRZ-to-Manchester serial line code converter that was designed in Chapter 3 (see the STG in Figure 3-31) has the Verilog behavioral description below. The waveforms in Figure 6-36, produced by simulating *NRZ_2_Manchester_Moore*, are identical to those of the gate-level model in Figure 3-35. The circuit synthesized from *NRZ_2_Manchester_Moore* is shown in Figure 6-37.[21]

```
module NRZ_2_Manchester_Moore (output reg B_out, input B_in, clock, reset_b);
 reg [1: 0] state, next_state;
 parameter      S_0 = 2'd0,
                S_1 = 2'd1,
                S_2 = 2'd2,
                S_3 = 2'd3;

 always @ (negedge clock, negedge reset_b)
  if (reset_b == 0) state <= S_0; else state <= next_state;

 always @ (state, B_in ) begin
  B_out = 0;                                    // Default assignment
  case (state)                                  // Fully decoded
   S_0: begin if (B_in == 0) next_state = S_1; else next_state = S_3; end

   S_1: begin next_state = S_2; end

   S_2: begin B_out = 1; if (B_in == 0) next_state = S_1; else next_state = S_3; end

   S_3: begin B_out = 1; next_state = S_0; end
  endcase
 end
endmodule
```

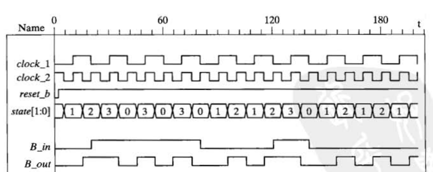


FIGURE 6-36 Results obtained from simulation of *NRZ_2_Manchester_Moore*, a Verilog behavioral model of a Moore-type NRZ-to-Manchester serial line converter.

[21]Problem 3 at the end of this chapter addresses the postsynthesis verification step in which the functionality of the synthesized circuit is shown to match that of the behavioral model.

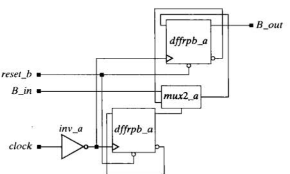


FIGURE 6-37 ASIC circuit synthesized from *NRZ_2_Manchester_Moore*.

6.6.4 Design Example: Synthesis of a Sequence Recognizer

A sequence recognizer asserts an output, *D_out*, when a given pattern of consecutive bits has been received in its serial input stream, *D_in* [6]. The data is typically synchronized by the opposite edge of the clock whose active edge controls the state transitions of the machine (i.e., the synchronization is said to be antiphase). Thus, data would be applied on the rising edge of the clock if the state transitions are to occur on the falling edge of the clock, and vice versa. Sequence recognizers can be realized as explicit FSMs of the Mealy or Moore type.

We will follow two conventions to describe sequence recognizers. The first convention clarifies the semantics of how the machine receives input bits. It states that the output of a Mealy machine is valid immediately *before* the active edge of the clock controlling the machine, and successive values are received in successive clock cycles.[22] This has implications for interpreting when the output is valid. The output immediately before the active edge of the clock is valid, reflecting the sampled value of the input and the state of the machine before the edge.

The second convention distinguishes between resetting and nonresetting machines. A nonresetting machine continues to assert its output if the input bit pattern is overlapping (i.e., the specified sequence of m bits has a nonempty intersection with a pattern formed from bits that immediately follow the mth bit). For example, overlapping sequences of 1111_2 are present in the bit stream 001111110_2. A resetting machine that detects a pattern of length m embedded within a longer pattern must de-assert when the $m + 1$st bit arrives, independently of its value. It then begins a new cycle of detecting the next m bits.

[22]The data must be stable prior to the active edge of the clock for at least the setup time of any flip-flop driven by *D_in*.

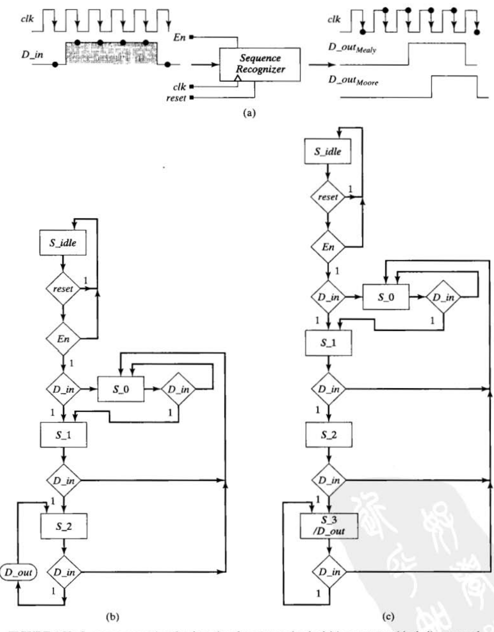


FIGURE 6-38 Sequence recognizer for detecting three successive 1s: (a) input–output block diagram and waveforms for Mealy- and Moore-type outputs, (b) ASM chart for a Mealy-type explicit FSM implementation, and (c) ASM chart for a Moore-type explicit FSM realization.

The sequence recognizer in Figure 6-38(a) is to sample the serial input, *D_in* on the falling edge of the clock and assert *D_out* if three successive samples are 1. The machine is to have a synchronous reset action and an enable signal, *En*, that controls whether the machine is active or not. The ASM chart in Figure 6-38(b) describes a nonresetting Mealy-type FSM implementing the desired behavior, and the chart in Figure 6-38(c) describes a Moore machine version.

In the Mealy version of the machine, the signal *reset* places the state of the machine in *S_idle*, where it resides until *En* is asserted.[23] After *En* is asserted the machine makes transitions to its other states, depending solely on *D_in*. Two successive samples of 1 will cause a transition to *S_2*, where *D_out* is asserted as long as *D_in* is held at 1. The structure of the ASM chart specifies that the Mealy machine will remain in *S_2* until a 0 is received (i.e., the machine is nonresetting); similarly, the Moore machine will remain in *S_3*. Note that the Moore machine has an extra state, because *D_out* in the Moore machine does not anticipate *D_in* and asserts *D_out* in the state reached *after* the third active edge of the clock (the Mealy machine anticipates *D_in* and asserts *D_out before* the third clock transition).

The Verilog models of the explicit state machines (given below) implement their output combinational logic with a continuous-assignment statement (see Chapter 4). To illustrate their response to two different formats for a serial line code converter (see Section 3.7), the testbench includes two instantiations of each machine. One machine has a bitstream for *D_in* encoded in an NRZ format and the other has a return-to-zero (RZ) format [7].

Simulation results are shown in Figure 6-39. First, compare the waveforms of *Mealy_NRZ* and *Moore_NRZ*. Note that *Mealy_NRZ* asserts in *S_2* (after two clocks) while *D_in* is *1*, and anticipates the third clock edge marking the end of the recognized pattern, but *Moore_NRZ* does not assert until after the third clock edge.

The importance of the convention stating that valid outputs of a Mealy machine are determined by the value of the inputs immediately prior to the active edge of the clock is illustrated by the waveform *Mealy_RZ*. Note that the Mealy machine has an invalid assertion of its output when the input has an RZ format, an apparent glitch. This assertion occurs immediately after the second clock and persists until *D_in* is possibly de-asserted in the next bit-time of the input. The valid output is 0, which is the value of *Mealy_RZ* immediately *before* the second clock. The value of *Mealy_RZ* immediately before the third clock is 1, which is a valid output. The processor that communicates with this machine would be responsible for interpreting *Mealy_RZ* correctly by detecting the value of the output immediately *before* the active edge of the clock.

The testbench also demonstrates the behavior of the machine if the input has a value of *1'bx* in simulation. The Verilog code was written to direct the machine to return to *S_idle* if the input is not a 0 or a 1. This situation cannot occur in the physical

[23]The structure of the ASM chart at *S_idle* implies that the reset action is synchronous, because *reset* is tested at only the active edges of the clock. For simplicity, the ASM chart omits showing that *reset* will cause a transition from any state to *S_idle*.

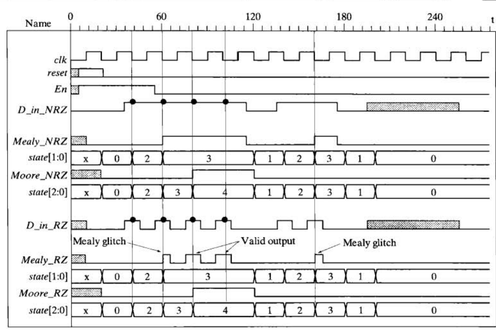


FIGURE 6-39 Sequence recognizer: simulation results for behavioral models of Mealy and Moore machines that detect three successive 1s in a serial bit stream encoded in NRZ and RZ formats.

machine, but the state could be assigned ***x*** in simulation. The circuits synthesized from the Mealy and Moore machines are shown in Figure 6-40.

```
module Seq_Rec_3_1s_Mealy (output D_out, input D_in, En, clk, reset);
 parameter    S_idle = 2'd0, // Binary code
              S_0 = 2'd1,
              S_1 = 2'd2,
              S_2 = 2'd3,
              S_3 = 2'd4;
 reg [1: 0] state, next_state;

 always @ (negedge clk)
  if (reset == 1) state <= S_idle; else state <= next_state;

 always @ (state, En, D_in) begin
  next_state = S_idle;
  case (state)
   S_idle: if ((En == 1) && (D_in == 1))          next_state = S_1;
              else if ((En == 1) && (D_in == 0)) next_state = S_0;
              else                               next_state = S_idle;
```

```
    S_0:  if (D_in == 0)                          next_state = S_0;
                 else if (D_in == 1)              next_state = S_1;
                 else                             next_state = S_idle;

    S_1:  if (D_in == 0)                          next_state = S_0;
                 else if (D_in == 1)              next_state = S_2;
                 else                             next_state = S_idle;

    S_2:  if (D_in == 0)                          next_state = S_0;
                 else if (D_in == 1)              next_state = S_2;
                 else                             next_state = S_idle;

    default:                                      next_state = S_idle;
   endcase
  end
  assign D_out = ((state == S_2) && (D_in == 1 )); // Mealy output
endmodule

module Seq_Rec_3_1s_Moore (output D_out, input D_in, En, clk, reset);
  parameter      S_idle = 3'd0,
                 S_0 = 3'd1,
                 S_1 = 3'd2,
                 S_2 = 3'd3,
                 S_3 = 3'd4;

  reg    [2: 0]  state, next_state;

  always @ (negedge clk)
   if (reset == 1) state <= S_idle; else state <= next_state;

  always @ (state, En, D_in) begin
   case (state)
    S_idle:      if ((En == 1) && (D_in == 1))     next_state = S_1; else
                 if ((En == 1) && (D_in == 0))     next_state = S_0;
                 else    next_state = S_idle;

    S_0:         if (D_in == 0)     next_state = S_0; else
                 if (D_in == 1)     next_state = S_1;
                 else    next_state = S_idle;

    S_1:         if (D_in == 0)     next_state = S_0; else
                 if (D_in == 1)     next_state = S_2;
                 else    next_state = S_idle;

    S_2, S_3:    if (D_in == 0)     next_state = S_0; else
                 if (D_in == 1)     next_state = S_3;
                 else    next_state = S_idle;
    default:     next_state = S_idle;
```

```
   endcase
  end

  assign D_out = (state == S_3);            // Moore output
 endmodule

 module t_Seq_Rec_3_1s ();
  reg D_in_NRZ, D_in_RZ, En, clk, reset;
  wire Mealy_NRZ;
  wire Mealy_RZ;
  wire Moore_NRZ;
  wire Moore_RZ;

  Seq_Rec_3_1s_Mealy M0 (Mealy_NRZ, D_in_NRZ, En, clk, reset);
  Seq_Rec_3_1s_Mealy M1 (Mealy_RZ, D_in_RZ, En, clk, reset);
  Seq_Rec_3_1s_Moore M2 (Moore_NRZ, D_in_NRZ, En, clk, reset);
  Seq_Rec_3_1s_Moore M3 (Moore_RZ, D_in_RZ, En, clk, reset);

  initial #275 $finish;
  initial begin #5 reset = 1; #22 reset = 0; end
  initial begin clk = 0; forever #10 clk = ~clk; end
  initial begin
   #5 En = 1;
   #50 En = 0;
  end
  initial fork
   begin #10 D_in_NRZ = 0; #25 D_in_NRZ = 1; #80 D_in_NRZ = 0; end
   begin #135 D_in_NRZ = 1; #40 D_in_NRZ = 0; end
   begin #195 D_in_NRZ = 1'bx; #60 D_in_NRZ = 0; end
   join
  initial fork
   #10 D_in_RZ = 0;
   #35 D_in_RZ = 1; #45 D_in_RZ = 0;
   #55 D_in_RZ = 1; #65 D_in_RZ = 0;
   #75 D_in_RZ = 1; #85 D_in_RZ = 0;
   #95 D_in_RZ = 1; #105 D_in_RZ = 0;
   #135 D_in_RZ = 1; #145 D_in_RZ = 0; #155 D_in_RZ = 1; #165 D_in_RZ = 0;
   #195 D_in_RZ = 1'bx; #250 D_in_RZ = 0;
  join
 endmodule
```

The data bits of the sequence recognizer in Figure 6-38(a) were used to control explicit state machines. This led to implementations of Mealy and Moore sequence recognizers having extra logic that forces the machines to *S_idle* if they enter an unused state, depending on the state assignment scheme. An alternative approach is to consider the sequence recognizer as a datapath unit in which the input bits are shifted through a register, with simple logic detecting whether the contents of the register match the pattern of 1s. The basic cores of two such implicit state machines are shown in Figure 6-41.

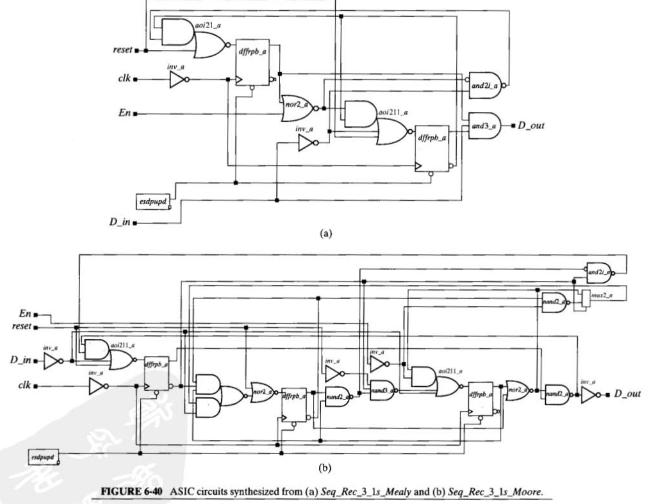


FIGURE 6-40 ASIC circuits synthesized from (a) *Seq_Rec_3_1s_Mealy* and (b) *Seq_Rec_3_1s_Moore*.

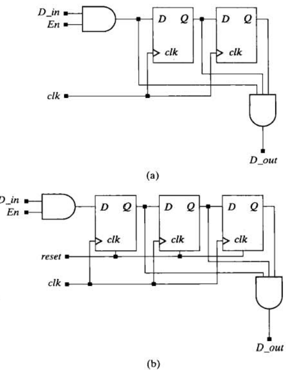


FIGURE 6-41 Partial implementations of shift register-based circuits for (a) Mealy and (b) Moore sequence recognizers for detecting three successive 1s in a serial bit stream.

Both machines gate the datapath to the shift register, which is a slight difference from the machines described by *Seq_Rec_3_1s_Mealy* and *Seq_Rec_3_1s_Moore*, which ignore *En* after their state has moved from *S_idle*. The Mealy-type machine in Figure 6-41 also gates *D_in* with the content of the register, has fewer states, and requires one less flip-flop than the Moore version.

The circuits shown in Figure 6-41 lack the logic required to fix (recirculate) the contents of the register when *En* is not asserted. Verilog models for the full implementations with shift registers are given below. They synthesize to the circuits shown in Figure 6-42, which are considerably simpler than those in Figure 6-40, where the machine's state is decoded to determine assertions of the output. In Figure 6-42 the data is stored and decoded directly. Thus, *an explicit state machine implementation of a sequence recognizer is not necessarily the most efficient implementation.* Note that the simulations results shown in Figure 6-42(c) match those in Figure 6-39.

```
module Seq_Rec_3_1s_Mealy_Shft_Reg (output D_out, input D_in, En, clk, reset);
  parameter Empty = 2'b00;
  reg [1: 0]   Data;

  always @ (negedge clk)
    if (reset == 1) Data <= Empty; else if (En == 1) Data <= {D_in, Data[1]};
  assign D_out = ((Data == 2'b11) && (D_in == 1 )); // Mealy output
endmodule

module Seq_Rec_3_1s_Moore_Shft_Reg (output D_out, input D_in, En, clk, reset);
  parameter Empty = 2'b00;
  reg [2: 0]   Data;

  always @ (negedge clk)
    if (reset == 1) Data <= Empty; else if (En == 1) Data <= {D_in, Data[2: 1]};
  assign D_out = (Data == 3'b111);   // Moore output
endmodule
```

6.7 Registered Logic

Variables whose values are assigned synchronously with a clock signal are said to be *registered*. Registered variables are updated at the active edges of the clock and are stable otherwise (i.e., they cannot glitch). The outputs of a Moore-type state machine are not registered, but they cannot glitch with changes at the machine's input.

Example 6.26

The output of *mux_reg* in Figure 6-43(a) is synchronized by the rising edge of *clock*, so the synthesis tool implements the combinational logic of a 4-channel mux with 8-bit datapaths, but registers the outputs of the mux in a bank of D-type flip-flops, as shown in Figure 6-43.

End of Example 6.26

Figure 6-44 shows structures for registering the outputs of Mealy and Moore-type state machines. The output of the storage register in the structures in Figure 6-44(a) and 6-44(b) lags the combinational values by one clock cycle (i.e., the output of the register corresponds to the state of the machine in the previous cycle). The structures in Figures 6-44(c) and 6-44(d) can be used if it is important that the registered outputs be formed in the same cycle as the state. The registered Mealy outputs are formed from the *next* state and inputs at the time of the active edge of the clock; the value stored in

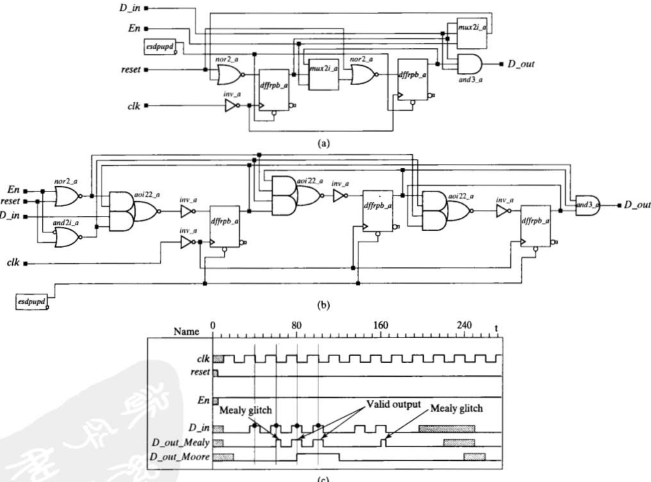


FIGURE 6-42 Synthesized shift register-based sequence recognizers for detecting 3 successive 1s in a serial bit stream: (a) Mealy machine, (b) Moore FSM, and (c) simulation results.

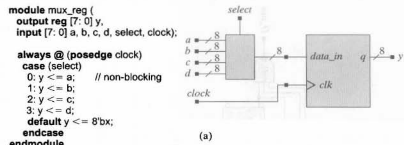


FIGURE 6-43 Multiplexer with registered output: (a) structural block diagram and (b) synthesized circuit.

the output register will correspond to the state reached at the clock transition and the inputs that caused the state transition. The registered Moore outputs are formed from the *next* state at the time of the active edge of the clock. The value stored in the output register will correspond to the state that is stored in the state register. The output is a registered Moore-type output.

Example 6.27

The sequence recognizers in Example 6.26 did not have registered outputs. The simulation results that were shown in Figure 6-42 had invalid assertions (glitches) of the output. The output of both machines can be registered. Include the following code in *Seq_Rec_3_1s_Mealy*[24]:

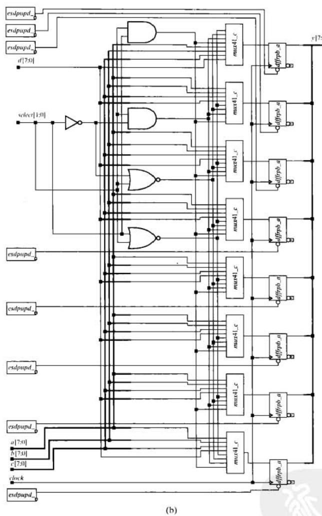
```
reg D_out_reg;
always @ (negedge clk)
  if (reset == 1) D_out_reg <= 0;
  else D_out_reg <= ((state == S_2) && (next_state == S_2) && (D_in == 1 ));
```

Notice that the clause (*state* = = *S_2*) is included in the logic to prevent a premature assertion while the state of the machine is *S_1* (see the ASM chart in Figure 6-38(b)).

Include the following code in *Seq_Rec_3_1s_Moore*:

```
reg D_out_reg;
always @ (negedge clk)
 if (reset == 1) D_out_reg <= 0; else D_out_reg <= (next_state == S_3);
```

[24]The port declarations of each machine must be modified to include the registered output.



(b)

FIGURE 6-43 Continued

The waveforms shown in Figure 6-45 show both registered and unregistered outputs for NRZ and RZ formatted serial inputs to the machine. Note that the unregistered output of the Mealy machine changes with the input, but the registered output does not, and that the value of the registered Mealy output corresponds to the value implied by the input and next state at the active (falling) edge of the clock. The unregistered

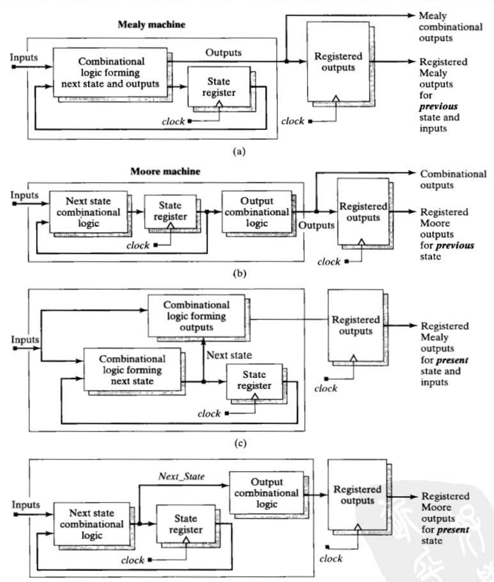


FIGURE 6-44 Registered outputs in (a) a Mealy machine, and (b) a Moore machine, (c) Mealy machine registered on the next-state function, and (d) Moore machine registered on the next-state function.

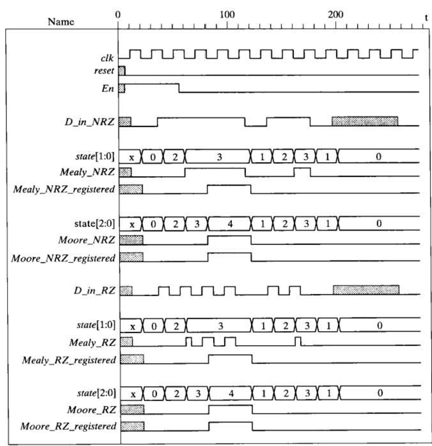


FIGURE 6-45 Simulations results showing registered and unregistered outputs of Mealy and Moore sequence recognizers that detect three successive 1s.

output anticipates the clock; the output of the registered machine does not. The waveforms of the registered and unregistered Moore outputs are identical. In the case of the unregistered machine, the output is formed by combinational logic; in the case of the registered machine, the output is the output of a register.

End of Example 6.27

Example 6.28

The waveform produced by a line converter that generates a NRZI waveform from a bit stream was shown in Figure 3-24 for Mealy and Moore machines. The Verilog model and testbench for a Mealy NRZI code converter is given below, and its state transition graph and simulation results are shown in Figure 6-46. The state machine is synchronized by *clock_2*, the fast clock, and its output is registered by *clock_1*, the clock that synchronizes the line bits. The testbench generates a bit sequence matching that of *B_in* shown in Figure 3-24. Notice that the signal *B_out* in Figure 6-46(b) does not match the waveform for $NRZI_{Mealy}$ shown in Figure 3-24. Observe, however, that the value of *B_out* in Figure 3-24 immediately before the rising edge of *clock_1* does produce the bit values corresponding to $NRZI_{Mealy}$ (denoted by the dots). The waveforms

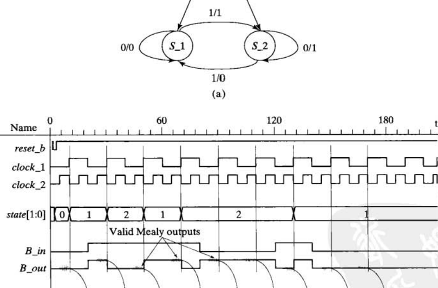


FIGURE 6-46 Line converter for generating a NRZI waveform: (a) state transition graph and (b) simulation results.

do not match because *B_out* in Figure 6-46(b) exhibits glitches produced by changes in the value of *B_in*. The Verilog model includes an additional output, *B_out_reg*, which is obtained by registering *B_out* at the rising edges of *clock_1*. The waveform of *B_out_reg* matches that shown in Figure 3-24, but with a latency. In a given state-time of the waveform, the output value is obtained from samples of the state and input immediately before the clock (i.e., the previous state and input as in Figure 6-44 (a)).

```
module NRZI_Mealy (output reg B_out, B_out_reg,
  input B_in, clock_1, clock_2, reset_b);
  reg [1: 0]        state, next_state;
  parameter         S_0 = 0,
                    S_1 = 1,
                    S_2 = 2,
                    dont_care_state = S_0,
                    dont_care_out = 0;

  always @ (posedge clock_1 or negedge reset_b)
    if (reset_b == 0) state <= S_0; else state <= next_state;

  always @ (posedge clock_1) B_out_reg <= B_out;   // Registered output

  always @ (state or B_in ) begin
    B_out = 0;
    case (state)
      S_0:          if (B_in == 0) begin next_state = S_1; B_out = 0; end
                    else begin next_state = S_2; B_out = 1; end

      S_1:          if (B_in == 0) begin next_state = S_1; B_out = 0; end
                    else begin next_state = S_2; B_out = 1; end

      S_2:          if (B_in == 1) begin next_state = S_1; B_out = 0; end
                    else begin next_state = S_2; B_out = 1; end

      default:      begin next_state = dont_care_state; B_out = dont_care_out; end
    endcase
  end
endmodule

module t_NRZI_Mealy ();
  wire B_out, B_out_reg;
  reg B_in, clock_1, clock_2, reset_b;
   parameter half_cycle_1 = 10, half_cycle_2 = 5;
   parameter cycle_1 = 2*half_cycle_1;

  NRZI_Mealy M0 (B_out, B_out_reg, B_in, clock_1, clock_2, reset_b);

  initial #500 $finish;
  initial begin clock_1 = 0; forever #half_cycle_1 clock_1 = !clock_1; end
  initial begin clock_2 = 0; forever #half_cycle_2 clock_2 = !clock_2; end

  initial begin #1 reset_b = 1; #1 reset_b = 0; #1 reset_b = 1; end
```

```
initial begin
  B_in = 0;
  #(cycle_1) B_in = 1;
  #(3*cycle_1) B_in = 0;
  #(2*cycle_1) B_in = 1;
  #(cycle_1) B_in = 0;
 end
endmodule
```

End of Example 6.28

6.8 State Encoding

The design of a sequential machine uses a set of flip-flops to represent the states of the machine and assigns a unique binary code to each state. State encoding determines the number of flip-flops that are required to hold the state, and influences the complexity of the combinational logic used to realize the next state and output of a synthesized state machine. The task of assigning a code to the states of a machine is called *state assignment* or *state encoding*. Because the number of different state assignments grows exponentially with the dimension of the state, it is not feasible to enumerate the possible state assignments of any but the simplest machines. Various algorithms are embedded within a synthesis tool to search for a good state assignment. Alternatively, the designer can assign state codes manually.

There are some general guidelines for manually assigning a state code: (1) if two states have the same next state for a given input, give them logically adjacent state assignments, (2) assign logically adjacent state codes to the next states of a given state, and (3) assign logically adjacent state codes to states that have the same output for a given input. These guidelines increase the possibility, but do not guarantee, that the combinational logic required to implement the output and next-state functions will be minimal [6].

The number of flip-flops in a finite state machine must be sufficient to represent the number of states as a binary number (e.g., a machine with eight states would require at least three flip-flops, and the state could be represented as a binary number). Other encodings are possible. The designer can choose the state assignment or let the synthesis tool optimize the assignment. Alternatively, the design of the state machine can be taken out of the hands of the general synthesis tool and passed to a special state machine design tool and companion optimizer. Table 6-1 lists several common state-assignment codes.

If the designer assigns the state code, the synthesis tool will treat the state machine like it treats random logic, and make no attempt to find an optimal assignment. A state machine with N states will require at least $\log_2 N$ flip-flops to store the encoded representation of the state, but it could have more. For example, a machine

TABLE 6-1 Commonly used state-assignment codes.

#	Binary	One-Hot	Gray	Johnson
0	0000	0000000000000001	0000	00000000
1	0001	0000000000000010	0001	00000001
2	0010	0000000000000100	0011	00000011
3	0011	0000000000001000	0010	00000111
4	0100	0000000000010000	0110	00001111
5	0101	0000000000100000	0111	00011111
6	0110	0000000001000000	0101	00111111
7	0111	0000000010000000	0100	01111111
8	1000	0000000100000000	1100	11111111
9	1001	0000001000000000	1101	11111110
10	1010	0000010000000000	1111	11111100
11	1011	0000100000000000	1110	11111000
12	1100	0001000000000000	1010	11110000
13	1101	0010000000000000	1011	11100000
14	1110	0100000000000000	1001	11000000
15	1111	1000000000000000	1000	10000000

with 64 states will require at least 8 flip-flops to encode the state. A BCD format simply adds one to a code to obtain the next code in the sequence. It uses the minimal number of flip-flops, but does not necessarily lead to an optimal realization of the combinational logic used to decode the next state and output of the machine. If a machine has more than 16 states, a binary code will result in a relatively large amount of next-state logic; the machine's speed will also be slower than alternative encodings. A Gray code uses the same number of bits as a binary code, but has the feature that two adjacent codes differ by only one bit, which can reduce the electrical noise in a circuit. A Johnson code has the same property, but uses more bits.

A code that changes by only one bit between adjacent codes will reduce the simultaneous switching of adjacent physical signal lines in a circuit, thereby minimizing the possibility of electrical crosstalk. These codes also minimize transitions through intermediate states, when state changes occur in the operation of the actual hardware. The problem of intermediate transitions arises because flip-flops in the state register do not change simultaneously. When more than one bit changes to make a state transition and the bits do not switch simultaneously, an intermediate state is present momentarily in the state register. This could have undesirable consequences (see Section 9.7).

A popular design methodology called *one-hot encoding* (for active-high logic, one-cold for active-low logic) uses more than the minimum number of flip-flops; in fact, it uses one for each state. The decoding logic in a one-hot machine uses fewer gates because the machine has to decode only a single bit of a register rather than a vector pattern.

One-hot state encoding uses more flip-flops than other forms of encoding, but it usually leads to simpler (fewer levels) decoding logic for the next state and the output of the machine. The decoding logic for one-hot machines does not become more complex

as more states are added to the design. Thus, the speed at which the machine can operate is not limited by the time to decode the state. One-hot machines can be faster, and the silicon area required by the extra flip-flops can be offset by the area saved by using simplified decoding logic. It is also quite easy to modify a one-hot design, because adding or removing a state does not affect the encoding of the other states. The design effort is reduced, too, because there is no need to encode a state-transition table. The STG is sufficient.

One-hot encoding used with a ***case*** statement might not produce the same results as one-hot encoding with an ***if*** statement. A ***case*** statement implicitly references all of the bits in the case expression, so a one-hot encoding with an ***if*** statement that tests individual bits might provide simpler decoding logic.

One-hot encoding usually does not correspond to the optimal state assignment, but it has overriding merit in some applications. For example, programmable logic, such as a FPGA, will have a fixed amount of flip-flop and combinational logic resources. Saving them does not necessarily provide a benefit. In the Xilinx architecture discussed in Chapter 8, a configurable logic block (CLB) uses lookup tables to implement combinational logic. An application that requires more decoding logic than that available in a single CLB will have to use additional CLBs. An attractive alternative might be to use one-hot state assignment to reduce the number of CLBs used by the machine and to reduce the need to use interconnect resources between CLBs. One-hot encoding is more reliable than binary encoding because fewer bits make transitions. Be aware, though, that in large machines, one-hot encoding will have several unused states, in addition to requiring more registers than alternative encodings.[25] Gray encoding is recommended for machines with more than 32 states because it requires fewer flip-flops than one-hot encoding and is more reliable than binary encoding because fewer bits change simultaneously.

A word of caution: If a state assignment does not exhaust the possibilities of a code, then additional logic will be required to detect and recover from transitions into unused states. Such transitions should not occur, but noise could cause the state to change to an unexpected value. It is essential that the machine be able to recover from such a state and resume operation. This additional logic will have an impact on the overall area required to realize the design.

6.9 Synthesis of Implicit State Machines, Registers, and Counters

An implicit state machine has one or more clock-synchronized (i.e., edge-sensitive) event-control expressions in a behavior. The synchronous behavior of an explicit FSM can contain only one such event-control expression, but an implicit FSM can contain multiple edge-sensitive event-control expressions in the same behavior. The clock edges of an implicit FSM define the boundaries of the machine's state transitions (i.e., the machine is in a fixed state between clock transitions). It is essential, for synthesis,

[25]This is not an issue in register-rich FPGAs.

that the multiple event-control expressions of an implicit FSM be synchronized to the same edge of the same clock, either ***posedge*** or ***negedge*** but not both.

6.9.1 Implicit State Machines

The description of an implicit FSM does not represent the value of its state by an explicitly declared register variable (***reg***). Instead, the state is defined implicitly by the evolution of activity within a cyclic (***always***...) behavior. Implicit FSMs may contain multiple clock-synchronized event-control expressions within the same behavior, and are considered to be a more general style of design than explicit FSMs. These machines have a major limitation—*each state may be entered from only one other state,* because states are determined by the evolution of the behavior from clock cycle to clock cycle, and a clock cycle can be entered only from the immediately preceding clock cycle. Thus, the ASM charts of the sequence recognizers in Figure 6-38 cannot be implemented as implicit state machines, but the counters and registers described in Chapter 5 can all be described as single-cycle implicit state machines. Likewise, the Verilog models of the shift-register-based sequence recognizers in Figure 6-41 are implicit state machines. *Any sequential machine with an identical activity flow in every cycle is a one-cycle implicit state machine, and its activity can be described by one state, S_running.* The simplest example of such a machine is a D-type flip-flop. Typically, an implicit state machine can be described with fewer statements than a corresponding explicit machine, which must have an elaborate, explicit, STG description. The STG of an implicit machine is implicit, and could be constructed from the behavioral description, if necessary.

Synthesis tools infer the existence of an implicit multi-cycle FSM when a cyclic (***always***) behavior has more than one embedded, clock-synchronized, event-control expression. The multiple event-control expressions within an implicit FSM separate the activity of the behavior into distinct clock cycles of the machine. For example, the behavior below has register assignments to *reg_a* and *reg_c* in the first clock cycle, and to *reg_g* and *reg_m* in the second clock cycle. Both cycles must execute before the activity flow returns to the beginning of the behavior. Note that the event-control expressions that are embedded within the behavior are not accompanied by the ***always*** keyword, which declares a behavior and cannot be nested. The role of these embedded event-control expressions is to suspend execution of the simulation until the active edge of the clock. They condition activity on the specified clock edge.

```
always @ (posedge clk)  // Synchronized event before first assignment
 begin
   reg_a <= reg_b;            // Executes in first clock cycle
   reg_c <= reg_d;            // Executes in first clock cycle.
   @ (posedge clk)            // Begins second clock cycle.
    begin
      reg_g <= reg_f;         // Executes in second clock cycle.
      reg_m <= reg_r;         // Executes in second clock cycle.
   end
 end
```

The states of an implicit FSM are not enumerated a priori. Each edge-sensitive transition determines a state transition. A synthesis tool that has the capability of synthesizing

an implicit state machine will use this information to determine the size of a physical register that will be synthesized to represent the state (e.g., the synthesized circuit will contain registers designated as "multiple wait states"). The tool will also extract and optimize the combinational logic that governs the state transitions in the physical machine.

6.9.2 Synthesis of Counters

A machine that is completely described by a single event-control expression is also an implicit state machine. At each significant clock event, the machine executes register operations, but does not make explicit state transitions. Synthesis tools easily synthesize a variety of counters and shift registers as single-cycle implicit state machines. Several were presented in Chapter 5. Even a ripple counter can be described and synthesized as a cascade of individual implicit state machines (one for each cell).

Example 6.29

A 4-bit ripple counter can be implemented with toggle (T-type) flip-flops. This type of counter has limited practical application because it takes excessive time to propagate changes through the cascaded chain of flip-flops, especially for long counters. The output count is also subject to glitches during the transitions. The Verilog description *ripple_counter* uses four behaviors to model the rippling effect, with successive stages of the counter triggered by the output of their immediately previous stage. The toggling action is controlled by the input *toggle*. No ASM chart is developed because devices trigger on signals other than the clock. The circuit simulates correctly and synthesizes. The wires *c0*, *c1*, and *c2* are required because the event-control expression must be a simple variable (not a bit-select) to comply with the style sheet for the synthesis tool that produced the result. The structure of the counter and the synthesized result are shown in Figures 6-47 and 6-48, respectively.

```
module ripple_counter (output reg [3: 0] count, input toggle, clock, reset);
  wire c0, c1, c2;
  assign c0 = count[0];
  assign c1 = count[1];
  assign c2 = count[2];

  always @ ( posedge reset, posedge clock)
    if (reset == 1'b1) count[0] <= 1'b0; else
    if (toggle == 1'b1) count[0] <= ~count[0];

  always @ ( posedge reset, negedge c0)
    if (reset == 1'b1) count[1] <= 1'b0; else
    if (toggle == 1'b1) count[1] <= ~count[1];

  always @ ( posedge reset, negedge c1)
    if (reset == 1'b1) count[2] <= 1'b0; else
    if (toggle == 1'b1) count[2] <= ~count[2];

  always @ ( posedge reset, negedge c2)
    if (reset == 1'b1) count[3] <= 1'b0; else
    if (toggle == 1'b1) count[3] <= ~count[3];
endmodule
```

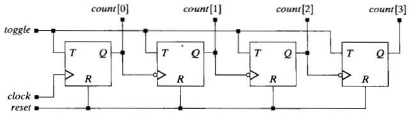


FIGURE 6-47 Structure of a 4-bit ripple counter.

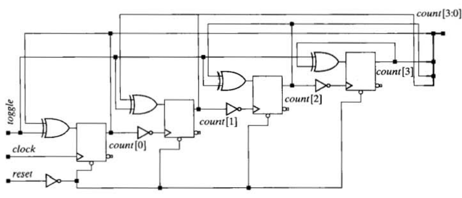


FIGURE 6-48 Synthesized circuit for a 4-bit ripple counter.

End of Example 6.29

Example 6.30

The ring counter presented in Example 5.40 is a single-cycle implicit state machine. Its description is greatly simplified compared to an alternative machine based on an elaboration of all possible states of an 8-bit register storing an explicit state.

End of Example 6.30

6.9.3 Synthesis of Registers

Storage elements in a sequential machine can be implemented with flip-flops or with latches, depending on the clocking scheme used by the machine. We will use the term

register to mean a memory structure formed by a group of D-type flip-flops with a common clock.[26]

Example 6.31

The shift register described by *shifter_1* below includes combinational logic forming the register variable *new_signal*. Since *new_signal* receives value within a synchronous behavior and is referenced outside the behavior, it will be synthesized as the output of a flip-flop, with the structure shown in Figure 6-49.

```
module shifter_1 (output reg sig_d, new_signal, input Data_in, clock, reset);
 reg sig_a, sig_b, sig_c;

 always @ (posedge reset, posedge clock) begin
   if (reset == 1'b1) begin
     sig_a <= 0;
     sig_b <= 0;
     sig_c <= 0;
     sig_d <= 0;
     new_signal <= 0;
   end
   else begin
     sig_a <= Data_in;
     sig_b <= sig_a;
     sig_c <= sig_b;
     sig_d <= sig_c;
     new_signal <= (~ sig_a) & sig_b;
   end
 end
endmodule
```

End of Example 6.31

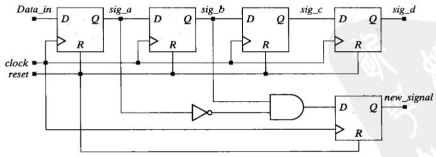


FIGURE 6-49 Generic structure of a shift register with registered combinational logic.

[26]Asynchronous registers using latches will be discussed in Chapter 7.

Example 6.32

In *shifter_2, new_signal* is formed outside of the behavior in a continuous assignment and is synthesized as the combinational logic shown in Figure 6-50.

```
module shifter_2 (output reg sig_d, output new_signal, input Data_in, clock, reset);
 reg sig_a, sig_b, sig_c;

 always @ (posedge reset, posedge clock) begin
  if (reset == 1'b1) begin
    sig_a <= 0;
    sig_b <= 0;
    sig_c <= 0;
    sig_d <= 0;
  end
  else begin
    sig_a <= Data_in;
    sig_b <= sig_a;
    sig_c <= sig_b;
    sig_d <= sig_c;
  end
 end

 assign new_signal = (~ sig_a) & sig_b;
endmodule
```

End of Example 6.32

Example 6.33

An accumulator is an important part of an ALU of a digital machine. Here, an accumulator forms a running sum of the samples of an input. Two versions, *Add_Accum_1* and *Add_Accum_2* are shown below. *Add_Accum_1* forms *overflow_1* one cycle after storing the results of an overflow condition, as shown in the simulation results in Figure 6-51(a).

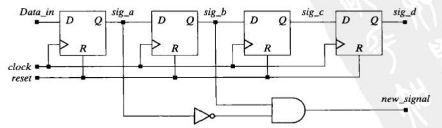


FIGURE 6-50 A shift register with separate, unregistered combinational logic.

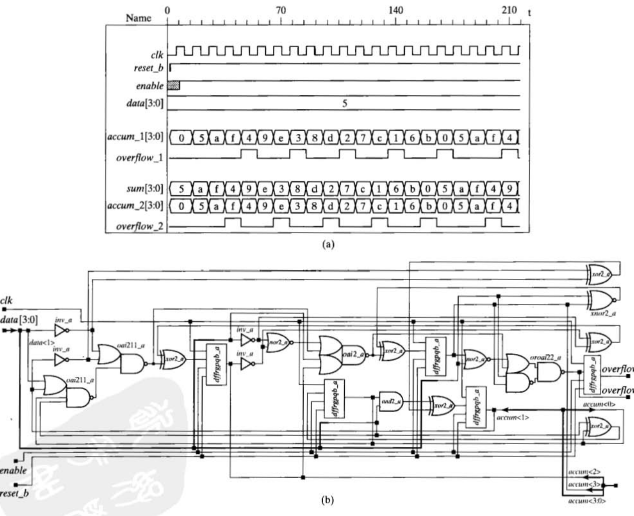


FIGURE 6-51 Synthesis of an accumulator for a 4-bit wide datapath: (a) simulation results and (b) synthesized circuit.

Add_Accum_2 forms *overflow_2* as an unregistered Mealy output. The machines synthesize differently, too, as shown in Figure 6-51(b), where *overflow_1* is formed in *Add_Accum_1 as a registered version of overflow_2*, which is formed in *Add_Accum_2*.

```
module Add_Accum_1 (
  output reg [3: 0] accum, output reg overflow,
  input [3: 0] data, input enable, clk, reset_b
);
  always @ (posedge clk, negedge reset_b)
    if (reset_b == 0) begin accum <= 0; overflow <= 0; end
    else if (enable) {overflow, accum} <= accum + data;
endmodule

module Add_Accum_2 (
  output reg [3: 0] accum, output overflow,
  input [3: 0] data, input enable, clk, reset_b
);
  wire [3: 0]  sum;
  assign {overflow, sum} = accum + data;

  always @ (posedge clk, negedge reset_b)
    if (reset_b == 0) accum <= 0;
    else if (enable) accum <= sum;
endmodule
```

End of Example 6.33

6.10 Resets

Every sequential module in a design should have a reset signal. Otherwise, the initial state of the machine cannot be controlled. A machine without a controllable initial state cannot be tested for manufacturing defects and cannot be operated predictably. Special care must be taken to describe the reset action of an implicit state machine that contains more than one event-control expression. Such machines must be disabled by an external agent. The first statement in the behavior that is associated with a reset signal must be a conditional statement that terminates execution of the behavior if the reset signal is asserted. Be careful: The ***disable*** statement must ensure that the machine begins executing at the top of the behavior when the reset is de-asserted. Incomplete resets will cause extra logic to be synthesized. Worse yet, the machine will reset to a different state, depending on when the reset is asserted.

Asynchronous reset signals can glitch, so it is recommended that asynchronous reset inputs be synchronized.[27] This can be done with a separate synchronizer.

[27]FPGAs commonly conserve routing resources by having a global set/reset that is automatically wired to all of its sequential devices. These signal paths may be slower than routed signals in more advanced technologies (e.g., the Xilinx Virtex parts).

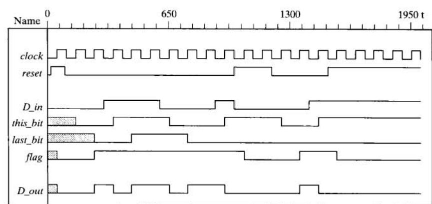


FIGURE 6-52 Simulation results for *Seq_Rec_Moore_imp.*

Example 6.34

A Moore-type sequence recognizer that asserts *D_out* after two successive samples of *D_in* are both either 1 or 0 is described by an implicit-state machine, and illustrates the care that must be taken with reset signals. The Verilog model, *Seq_Rec_Moore_imp*, uses a two-stage shift register to hold samples of the input bit stream, and generates the waveforms in Figure 6-52. Additional logic, in the form of a state machine, must prevent the output from asserting prematurely (i.e., before two samples have been received). A variable, *flag*, will be set after two samples have been received. We will illustrate start-up of the machine and explore the consequences of partially resetting the shift register.

If only the last stage of the shift register is flushed under the action of *reset*, the description synthesizes to the circuit shown in Figure 6-53. The synthesis tool forms a state

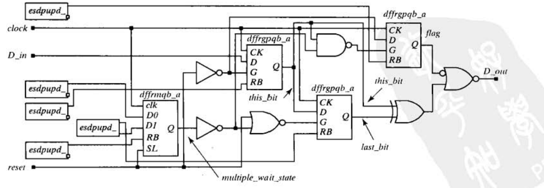


FIGURE 6-53 Circuit synthesized from *Seq_Rec_Moore_imp.*

register, *multiple_wait_state*. The machine flushes *last_bit* or loads *this_bit* at the first active edge of *clock*, depending on *reset*. An assertion of *reset* terminates the activity of *machine* until *reset* is de-asserted. Then *this_bit* is loaded with the first sample of the bit stream, and the machine enters a loop in which data is shifted through the pipeline. *flag* is asserted at the second active edge of *clock*, and it remains asserted until the activity of the named block *machine* is terminated by *reset*. Signal *flag* is used in the continuous assignment to *D_out* to ensure that the machine will not assert prematurely after a reset condition. The entire behavior must be encapsulated as the named block *wrapper_for_synthesis* to enable the synthesis tool to create an implementation of the circuit.

```
module Seq_Rec_Moore_imp (output D_out, input D_in, clock, reset);
 reg last_bit, this_bit, flag;
 always begin: wrapper_for_synthesis
  @ (posedge clock /* or posedge reset*/) begin: machine
   if (reset == 1) begin
    last_bit <= 0;
    // this_bit <= 0;
    // flag <= 0;
    disable machine; end
   else begin
    // last_bit <= this_bit;
    this_bit <= D_in;
    forever
     @ (posedge clock /* or posedge reset */) begin
      if (reset == 1) begin
      // last_bit <= 0;
      // this_bit <= 0;
      flag <= 0;
      disable machine; end
     else begin
      last_bit <= this_bit;
      this_bit <= D_in;
      flag <= 1; end      // second edge
     end
    end
   end // machine
  end // wrapper_for_synthesis

 assign D_out = (flag && (this_bit == last_bit));
endmodule
```

In Figure 6-53 two gated-input flip-flops (*dffrgpqb_a*) form a pipeline for *last_bit* and *this_bit*. When the gate input (G) of this type of flip-flop is low, the Q output is connected to the D input through internal feedback, while ignoring the external D input. Otherwise, the external D input is the input. A third gated-input flip-flop holds *flag*, which is gated together with the difference of *last_bit* and *this_bit* to form *out_bit*. The active-low input (RB) of all of the flip-flops is disabled by the *esdpupd* device. The synthesis tool inserts a D-type flip-flop with multiplexed input (*dffrmpqb_a*) to hold *multiple_wait_state* (created by the synthesis tool) indicating whether two samples

have been received or not. The active-low *RB* (reset) input of the flip-flop is disabled (for synchronous operation), and the active-low *SL* (set) is wired to *reset*. The *D0* and *D1* inputs are wired to power and ground, respectively, through *esdpupd* and the active-low *SL* input, which is connected to *reset*. When *SL* is low (*reset* is not asserted) *D0* is selected, and when *SL* is high (*reset* is asserted), *D1* is selected.

Now consider the action of *reset*. While *reset* is asserted, its inverted value causes *this_bit* and *last_bit* to hold their value (through internal feedback); it also drives the NAND-gate at the input to *flag* to get the value of its external input, which is held to 0. Thus, the reset conditions specified by the behavioral description are met for *this_bit, last_bit,* and *flag.*

At the first clock after *reset* is de-asserted, the *multiple_wait_state* gets 1, setting up the datapath from *this_bit* to *last_bit* on subsequent clocks. Also, *this_bit* gets *in_bit* after *reset* is de-asserted.

Note that the first reset statement within the loop in *Seq_Rec_Moore_imp* sets only *flag* to 0, not *this_bit* and *last_bit*. This implies that *this_bit* and *last_bit* are to remain unchanged by *reset* (i.e., the pipeline is not flushed). If the comments in the model are removed to cause the reset action to flush the pipeline, the description synthesizes to the simpler realization shown in Figure 6-54. *When registers are not flushed*

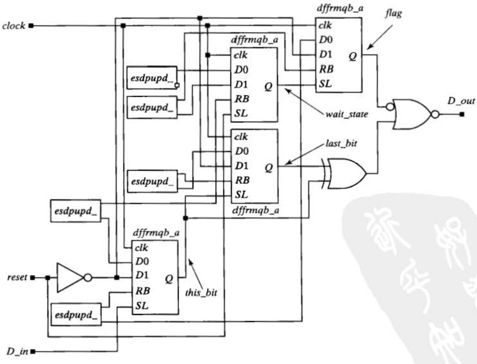


FIGURE 6-54 Circuit synthesized from *Seq_Rec_Moore_imp* with comments removed to flush the pipeline at reset.

on reset, additional logic is required to feed their outputs back to their inputs to retain their state under the action of the clock. This additional logic can be avoided by driving the register to a known value on reset. In this example, the flag register prevents undesirable consequences of not flushing the registers and not fully loading the pipeline, but this style leads to needless logic of extra muxes and/or more complicated flip-flop cells.

End of Example 6.34

6.11 Synthesis of Gated Clocks and Clock Enables

Designers avoid using gated clocks because they can lead to problematic timing behavior of the host circuit. On the other hand, low-power designs deliberately disable clocks to reduce or eliminate power wasted by useless switching of transistors. Improperly gated clocks can add skew to the clock path and cause the clock signal to violate a flip-flop's constraint on the minimum width of the clock pulse. The recommended way to write a Verilog description that synthesizes a gated clock is shown below.

```
module best_gated_clock (output reg Q, input data, data_gate, clock, reset_);
  always @ (posedge clock, negedge reset_)
   if (reset_ == 0) Q <= 0; else if (data_gate) Q <= data;        // Infers storage
endmodule
```

The description multiplexes the data with the output of a flip-flop. When the signal *data_gate* is asserted, the data is presented to the input of the flip-flop. When *data_gate* is de-asserted, the output of the flip-flop is unchanged. This description synthesizes into the circuit shown in Figure 6-55. The circuit is synchronized by *clock*, but *data_gate* gates the action of *clock*. Cell libraries may contain an encapsulation of this structure as a library cell for a flip-flop. Note that the synthesis tool selected an inverter and a negative-edge-sensitive flip-flop based on the availability of cells in the library.

Whether a synthesis tool infers a clock enable circuit from a Verilog description depends on the style of coding. For example, the cyclic behavior given below implies

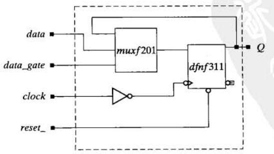


FIGURE 6-55 Synthesis result for the recommended structure of a gated clock.

that *q_out* gets value conditionally, depending on *enable*. This will synthesize logic to implement a clock enable.

```
always @ (posedge clock)
 if (enable == 1) q_out <= Data_in;
```

6.12 Anticipating the Results of Synthesis

It is advisable to anticipate what the synthesis process will produce and then examine the results against those expectations. There are more details about synthesis than can be covered here, but this section will cover some of the basic rules that will help the designer anticipate the results of synthesis and write Verilog descriptions that infer the desired result [8]. Each vendor's tool operates differently, so it is advisable to experiment with a synthesis tool to learn how it handles particular styles of coding.

6.12.1 Synthesis of Data Types

A synthesis tool will retain primary inputs or outputs in the design, but it may eliminate internal nets. By eliminate, we mean that a synthesized circuit may not have a structural connection (i.e., wire) that was in the Verilog model from which the circuit was synthesized. Integers are stored as 32-bit data objects, so use sized numbers (e.g., 8'*b*0110_1110) to reduce the size of the register required to hold a parameter. Do not use explicit values of *x* or *z* in logical tests (e.g., $A == 4'bx$). They have no hardware counterpart.

6.12.2 Operator Grouping

All of the predefined Verilog operators may be used in expressions forming a binary or Boolean value. Some operators may be treated in a special way by the technology mapper that is part of a synthesis tool. For example, the Verilog operators +, −, <, >, and = may be mapped directly to a library element if it is available. Otherwise, the synthesis tool will convert the operator into an equivalent set of Boolean equations that will be optimized. Be aware that the operands of some Verilog operators must be restricted for successful synthesis. Shift operators (<<, >>) within a behavior are synthesizable, provided that the shift is by a constant number of bits. The reduction, bitwise, and logical connective operators (see Appendix D) are each equivalent to operations performed by a logic gate. Thus, these operators are translated into a set of equivalent Boolean equations and synthesized into combinational logic. The synthesis engine will optimize these equations and then map the generic description into the target technology library.

The conditional operator (? ... :) synthesizes into library muxes or into gates that implement the functionality of a mux. The expression to the left of *?* is formed as control logic for the mux. A conditional operator must be complete—an expression must be given for both the true and false conditions. When an expression has multiple operators, the architecture of the synthesized result will reflect the parsing of the compiler (i.e., left-to-right) and the precedence of the operators. The designer can influence the outcome by using parentheses to form sub-expressions.

Example 6.35

The continuous assignment to *sum1* in *operator_group* is equivalent to the continuous assignment to *sum2*, but *sum2* will synthesize to a faster circuit.

```
module operator_group (output [4: 0] sum1, sum2, input a, b, c, d);
 assign sum1 = a + b + c + d;
 assign sum2 = (a + b) + (c + d);
endmodule
```

End of Example 6.35

The structures of the synthesized circuits are shown in Figure 6-56. The architectural improvement created by the grouping of terms leads to trade-offs in the synthesized result. The logic for *sum1* has three levels, compared to two for *sum2*, so *sum2* will be approximately 30% faster. The longest path forming *sum1* goes through three adders. If power is a consideration, input *d* forming *sum1* could be used for the signal that changes more frequently. It can also be used to accommodate a late-arriving signal by having to pass through only one adder. Within these overall structures, the synthesis tool can also optimize the implementation of the individual adders.

6.12.3 Expression Substitution

Synthesis tools perform expression substitution to determine the outcome of a sequence of procedural (blocking) assignments in a behavior. The designer can often

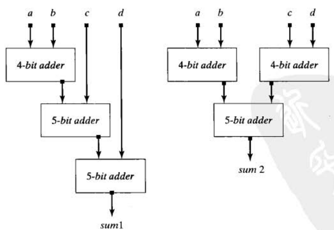


FIGURE 6-56 Architectural improvement resulting from operator grouping.

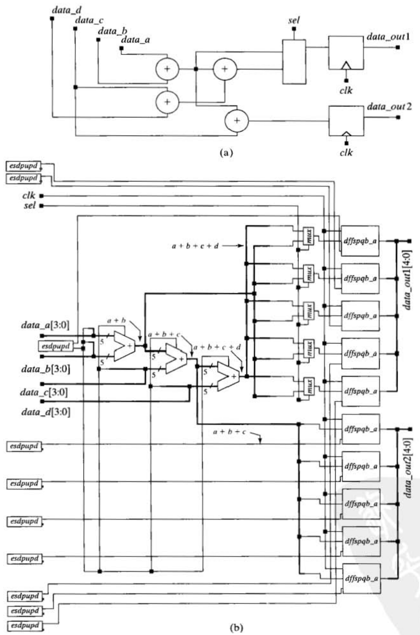


FIGURE 6-57 Dataflow structure and synthesized circuit resulting from (a) *multiple_reg_assign* and (b) *expression_sub*.

write an alternative and more readable description of the same functionality. If it is necessary for procedural assignments to be used, be aware that expression substitution will affect the result.

Example 6.36

The assignments in *multiple_reg_assign* execute sequentially, with immediate changes to the target register variables, so *data_a* + *data_b* is substituted into the expression for *data_out2* and used in the subsequent assignment to *data_out1*. Figure 6-57(a) shows the effective data flow implemented by the functionality. The behavior of *expression_sub* is equivalent to the behavior of *multiple_reg_assign*, but the former style makes the effect of expression substitution more apparent. Both versions synthesize to the circuit in Figure 6-57(b). A recommended style is given by *expression_sub_nb*, which implements equivalent logic with the nonblocking operator (<=)

```
module multiple_reg_assign (
  output reg [4: 0] data_out1, data_out2,
  input [3: 0] data_a, data_b, data_c, data_d, input sel, clk
);
  always @ (posedge clk) begin
    data_out1 = data_a + data_b ;
    data_out2 = data_out1 + data_c;
    if (sel == 1'b0)
      data_out1 = data_out2 + data_d;
  end
endmodule

module expression_sub (
  output reg [4: 0] data_out1, data_out2,
  input [3: 0] data_a, data_b, data_c, data_d, input sel, clk
);
  always @ (posedge clk) begin
    data_out2 = data_a + data_b + data_c;
    if (sel == 1'b0) data_out1 = data_a + data_b + data_c + data_d;
    else data_out1 = data_a + data_b;
  end
endmodule

module expression_sub_nb (output reg [4: 0] data_out1nb, data_out2nb,
  input [3: 0] data_a, data_b, data_c, data_d, input sel, clk
);
  always @ (posedge clk) begin
    data_out2nb <= data_a + data_b + data_c;
    if (sel == 1'b0) data_out1nb <= data_a + data_b + data_c + data_d;
    else data_out1nb <= data_a + data_b;
  end
endmodule
```

End of Example 6.36

6.13 Synthesis of Loops

A loop in a cyclic behavior is said to be *static*, or *data-independent*, if the number of its iterations can be determined by the compiler *before* simulation (i.e., the number of iterations is fixed and independent of the data). A loop is said to be data-dependent if the number of iterations depends on some variable during operation. In addition to having a dependency on data, a loop may have a dependency on embedded timing controls (i.e., an event-control expression). Figure 6-58 shows possible loop structures. In principle, static loops can be synthesized using ***repeat***, ***for***, ***while***, and ***forever*** loop constructs, but a given vendor might choose to confine the descriptive style of a static loop to a particular construct. The most likely form is that of a ***for*** loop. Non-static loops that do not have internal timing controls are problematic—they cannot be synthesized.

6.13.1 Static Loops without Embedded Timing Controls

If a loop has no internal timing controls and no data dependencies, its computational activity is implicitly combinational. The mechanism of the loop is artificial—the computations of the loop can be performed without memory instantaneously. The iterative

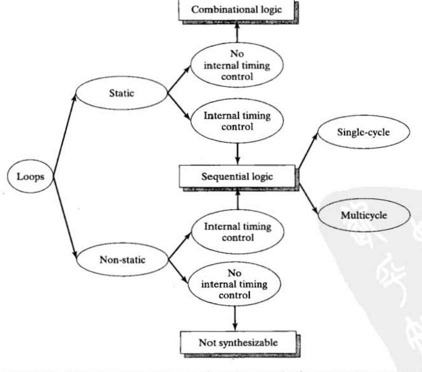


FIGURE 6-58 Possible loop structures that can be formed by procedural statements in a cyclic behavior.

computational sequence has a non-iterative counterpart that can be obtained by unrolling the loop, and the operations in the unrolled loop can occur at a single time step of the simulator.

Example 6.37

The loop in *for_and_loop_comb* does not depend on the data and does not have embedded event controls. It iterates for a fixed, predetermined number of steps and terminates. The description synthesizes to the anticipated combinational circuit in Figure 6-59.

```
module for_and_loop_comb (output reg [3: 0] out, input [3: 0] a, b);
 reg [2: 0] i;

 always @ (a, b) begin
  for (i = 0; i <= 3; i = i+1)
   out[i] = a[i] & b[i];
 end
endmodule
```

The unrolled loop in Example 6.37 is equivalent to the following assignments:

```
out[0] = a[0] & b[0];
out[1] = a[1] & b[1];
out[2] = a[2] & b[2];
out[3] = a[3] & b[3];
```

which correspond to a bitwise-and of two 4-bit datapaths. There are no dependencies between the assignments, so the order in which the statements are evaluated does not affect the outcome of the evaluations.

End of Example 6.37

A tool that supports synthesis of a static ***repeat*** loop with no internal timing controls will replace it by equivalent, synthesized combinational logic. The behavior of a ***for*** loop with static range is equivalent to a ***repeat*** loop with the same range, so some tools support only the ***for*** loop.

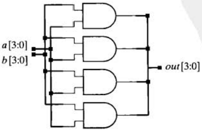


FIGURE 6-59 Synthesis of bitwise-and operations in a static ***for*** loop.

Example 6.38

Consider a sequential machine that receives a word of data in parallel format and then asserts an output that encodes the number of 1s in the word. The functionality can be implemented by combinational logic, and, if the hardware is fast enough, can execute in one clock cycle. The loop in the Verilog model *count_ones_a* below has no internal timing controls and is static—it does not depend on *data*. *The order in which the statements in the cyclic behavior execute is important*, and the algorithm in the model uses the blocked assignment operator (=).

Note that *bit_count* asserts after the loop has executed, within the same clock cycle in which the loop executes. Simulation results are shown in Figure 6-60(a). Be

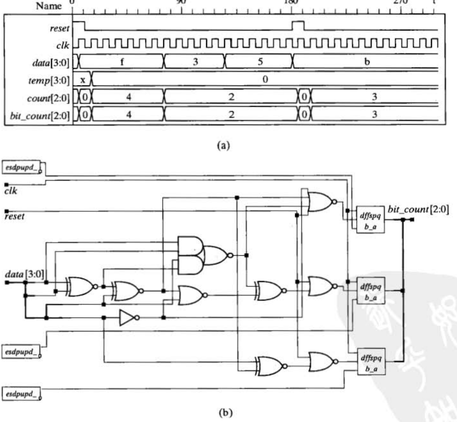


FIGURE 6-60 Results for (a) simulation of *count_ones_a*, and (b) circuit synthesized from *count_ones_a*.

careful in interpreting the results, for the loop executes in one time step of simulation, virtually instantaneously. Consequently, the displayed values of *temp, count*, and *bit_count* are final values that result after any values generated in intermediate simulation cycles have been overwritten. This version of the machine synthesizes into clock-compatible combinational logic (i.e., the outputs are stable within one cycle of the clock) with registered outputs.

The synthesized circuit is shown in Figure 6-60(b). The contents of register variables *index* and *temp* do not have a lifetime outside of the cyclic behavior in which they are assigned value (i.e., they are not referenced elsewhere). Both variables are eliminated by the synthesis tool. The only synthesized register is *bit_count*. Signal *bit_count* is registered because its value is assigned within an edge-sensitive cyclic behavior; *bit_count* is also an output port. The reset action is synchronous, and the reset inputs of the selected D-type flip-flops are hard wired to 1 to disable them.

```
module count_ones_a #( parameter data_width = 4, count_width = 3)(
 output reg [count_width -1: 0] bit_count,

  input [data_width -1: 0] data,
  input clk, reset

);
  reg     [count_width-1: 0] count, index;
  reg     [data_width-1: 0]  temp;

  always @ (posedge clk)
    if (reset) begin count = 0; bit_count = 0; end
    else begin
      count = 0;
      bit_count = 0;
      temp = data;
      for (index = 0; index < data_width; index = index + 1) begin
        count = count + temp[0];
        temp = temp >> 1;
      end
     bit_count = count;
   end
endmodule
```

End of Example 6.38

6.13.2 Static Loops with Embedded Timing Controls

If a static loop has an embedded edge-sensitive event-control expression, the computational activity of the loop is synchronized and distributed over one or more cycles of the clock. As a result, the behavior is that of an implicit state machine in which each

iteration of the loop occurs after a clock edge. The behavior may include additional computational activity that is placed in the cycle that immediately follows the loop's expiration.

Example 6.39

As an alternative to the static loop without embedded timing controls, we will now consider three equivalent versions of the machine that counts the 1s in a word of data, with each machine using a different static loop structure and having embedded timing controls. Machines *count_ones_b0*, *count_ones_b1*, and *count_ones_b2* use ***forever***, ***while***, and ***for*** loops, respectively. Each loop structure has an embedded event-control expression that is synchronized by an external clock signal. The loops execute for a fixed number of clock cycles, independently of the data. The loops can be unrolled and controlled by an FSM (sequential logic) whose state transitions correspond to the iterations of the loop. The simulation results shown in Figure 6-61 for a 4-bit data path demonstrate that the machines have identical functionality. Signals *bit_count_0*, *bit_count_1*, and *bit_count_2* are the outputs of the machines *count_ones_b0*, *count_ones_b1*, and *count_ones_b2*, respectively. Only the styles in *count_ones_b0* and *count_ones_b1* were supported by the synthesis tool.[28] Only *count_ones_b2* has the

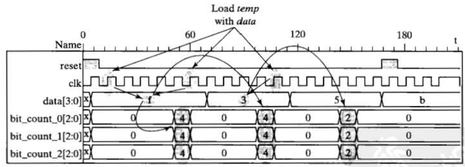


FIGURE 6-61 Simulation of three versions of *count_ones_b* showing start-up with reset, and reset on-the-fly.

[28]Synopsys Design Compiler™ was used to obtain the synthesized circuits. It is common for EDA tools to support a restricted descriptive style.

same functionality if nonblocking assignments replace the procedural assignments in the models (see Problem 45 at the end of this chapter).

```
module count_ones_b0 #( parameter data_width = 4, count_width = 3)(
  output reg [count_width -1: 0]    bit_count,
  input  [data_width -1: 0]         data,
  input                             clk, reset
);

  reg     [count_width-1: 0]        count;
  reg     [data_width-1: 0]         temp;
  integer                           index;

  always begin: wrapper_for_synthesis
   @ (posedge clk) begin: machine
    if (reset) begin bit_count = 0; disable machine; end
    else
     count = 0; bit_count = 0; index = 0; temp = data;
     forever @ (posedge clk)
      if (reset) begin bit_count = 0; disable machine; end
      else if (index < data_width-1) begin
       count = count + temp[0];
       temp = temp >> 1;
       index = index + 1;
      end
      else begin
       bit_count = count + temp[0];
       disable machine;
      end
     end // machine
    end // wrapper_for_synthesis
endmodule

module count_ones_b1 #( parameter data_width = 4, count_width = 3)(
  output reg [count_width -1: 0]    bit_count,
  input   [data_width -1: 0]        data,
  input                             clk, reset

);
  reg [count_width-1: 0]            count;
  reg [data_width-1: 0]             temp;
  integer                           index;

  always begin: wrapper_for_synthesis
   @ (posedge clk) begin: machine
    if (reset) begin bit_count = 0; disable machine; end
    else begin
     count = 0; bit_count = 0; index = 0; temp = data;
     while (index < data_width) begin
      if (reset) begin bit_count = 0; disable machine; end
```

```
        else if ((index < data_width) && (temp[0] ))
          count = count + 1;
          temp = temp >> 1;
          index = index +1;
          @ (posedge clk);
        end
      if (reset) begin bit_count = 0; disable machine; end
      else bit_count = count;
        disable machine;
      end
    end // machine
  end // wrapper_for_synthesis
endmodule

module count_ones_b2 #( parameter data_width = 4, count_width = 3)(
  output reg [count_width -1: 0]      bit_count,
  input      [data_width -1: 0]       data,
  input                               clk, reset
);
  reg [count_width-1: 0]              count;
  reg [data_width-1: 0]               temp;
  integer                             index;

  always begin: machine
    for (index = 0; index <= data_width; index = index +1) begin
      @ (posedge clk)
      if (reset) begin bit_count = 0; disable machine; end
      else if (index == 0) begin count = 0; bit_count = 0; temp = data; end
      else if (index < data_width) begin count = count + temp[0]; temp = temp >> 1; end
      else bit_count = count + temp[0];
    end
  end // machine
endmodule
```

End of Example 6.39

6.13.3 Nonstatic Loops without Embedded Timing Controls

The number of iterations to be executed by a loop having a data dependency cannot be determined before simulation. If the loop does not have embedded timing control, the behavior can be simulated, *but it cannot be synthesized.* Under the action of a simulator, the behavior is virtually sequential and can be simulated, but hardware cannot execute the computation of the loop in a single cycle of the clock. We will demonstrate this by the following example.

Example 6.40

The computational activity of counting the 1s in a word of data is wasted after the last 1 is found. The data-dependent loop in *count_ones_c* implements a more efficient machine than *count_ones_b*. At the first active edge of the clock after reset, the machine loads *data* into *temp*, and counts the number of 1s in *temp* by repeatedly adding the value of the least significant bit (LSB) of *temp* to *count*, and shifting the word. This continues as long as the word is not empty of 1s (i.e., until the reduction-or of *temp*, | *temp*, is false). So the data word 0001_2 will execute in fewer iterations than 1000_2 because the reduction-or of the word that results from the first right-shift in *count_ones_c* is empty of 1s. The computational activity occurs in a single cycle of the clock, so the efficiency would be apparent in simulation with long words of data. The simulation results in Figure 6-62 show the value of *index* at the end of the loop. Note that the final results (i.e., those at the end of the clock cycle of computation) are displayed. Although this behavior is attractive for simulation, it cannot be synthesized. The task of counting the 1s in a word is fundamentally combinational, but combinational logic cannot perform the sequential steps of the loop in one cycle of the clock, and at the same time terminate the activity if the word becomes empty of 1s. The loop cannot be unrolled statically because its length is data-dependent.

```
module count_ones_c #( parameter data_width = 4, count_width = 3)(
  output reg [count_width -1: 0]    bit_count,
  input [data_width -1: 0]          data,
  input                             clk, reset
);
  reg [count_width-1: 0]    count, index;
  reg [data_width-1: 0]     temp;

  always @ (posedge clk)
    if (reset) begin count = 0; bit_count = 0; end
    else begin
      count = 0;
      temp = data;
      for (index = 0; | temp; index = index + 1) begin
        if (temp[0] ) count = count + 1;
          temp = temp >> 1;
        end
        bit_count = count;
      end
endmodule
```

End of Example 6.40

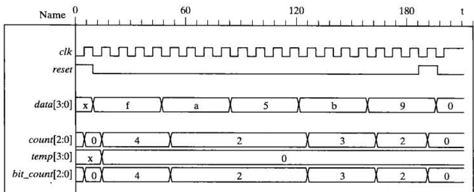


FIGURE 6-62 Simulation of *count_ones_c*, which has a data-dependent loop, and cannot be synthesized.

6.13.4 Nonstatic Loops with Embedded Timing Controls

A nonstatic loop may implement a multicycle operation. The data dependency alone is not a barrier to synthesis because the activity of the loop can be distributed over multiple cycles of the clock. However, the iterations of a nonstatic loop must be separated by a synchronizing edge-sensitive event control expression in order to be synthesized.

Example 6.41

The cyclic behavior in *count_ones_d* has edge-sensitive timing controls within a nonstatic ***while*** loop. The sequential activity of the loop is distributed over multiple cycles of the clock. First, the data is loaded into a shift register. Then it shifts the data through the register on successive clock cycles. After all of the data have been shifted, one more cycle elapses before *bit_count* is ready. Simulation results are presented in Figure 6-63.[29] Note that when $data = 3_H = 0011_2$, the loop terminates after the second cycle.

```
module count_ones_d #( parameter data_width = 4, count_width = 3)(
  output reg [count_width -1: 0]    bit_count,
  input [data_width -1: 0]          data,
  input                             clk, reset
```

[29]See Problem 6.13 for an exercise requiring synthesis of ***count_ones_d***.

```
);
 reg [count_width -1: 0]            count;
 reg [data_width -1: 0]             temp;

 always begin: wrapper_for_synthesis
  @ (posedge clk)
  if (reset) begin count = 0; bit_count = 0; end
  else begin: bit_counter
   count = 0;
   temp = data;
   while (temp)
   @ (posedge clk)
   if (reset) begin
    count = 2'b0;
    disable bit_counter;
   end
   else begin
    count = count + temp[0];
    temp = temp >> 1;
   end
   @ (posedge clk)
   if (reset) begin
    count = 0;
    disable bit_counter;
   end
   else bit_count = count;
  end // bit_counter
 end    // wrapper_for_synthesis
endmodule
```

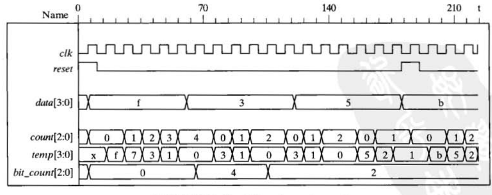


FIGURE 6-63 Simulation of *count_ones_d*.

The next version of a machine that counts the 1s in a word of data, *count_ones_SD*, adds signals *start* and *done* to the port structure, and eliminates the data dependency in the loop. Asserting *start* launches counting.[30]

```
module count_ones_SD #(parameter data_width = 4, count_width = 3)(
  output reg [count_width -1: 0]     bit_count,
  output reg                         done,
  input [data_width -1: 0]   data,
  input                      start, clk, reset
);
  reg [count_width-1: 0]     count, index;
  reg [data_width-1: 0]      temp;

  always @ (posedge clk) begin: bit_counter
   if (reset == 1'b1) begin count = 0; bit_count = 0; done = 0; end
   else if (start == 1'b1) begin
    done = 0;
    count = 0;
    bit_count = 0;
    temp = data;
    for (index = 0; index < data_width; index = index + 1)
     @ (posedge clk)
     if (reset == 1'b1) begin
      count = 0;
      bit_count = 0;
      done = 0;
      disable bit_counter; end
   else begin
    count = count + temp[0];
    temp = temp >> 1;
   end
   @ (posedge clk)  // Required for final register transfer
   if (reset == 1'b1) begin count = 0; bit_count = 0; done = 0;
    disable bit_counter; end
   else begin
    bit_count = count;
    done = 1; end
   end
  end
 endmodule
```

Note in the simulation results in Figure 6-64 that *start* asserts for one cycle. Signal *index* increments at each edge of the clock until all bits have been counted. Then *done* is asserted. When *reset* is asserted in the middle of a counting sequence, the machine re-initializes the registers and restarts the sequence.

End of Example 6.41

[30]See Problem 14 at the end of this chapter for an exercise requiring the use of alternative loop structures in *count_ones_SD*.

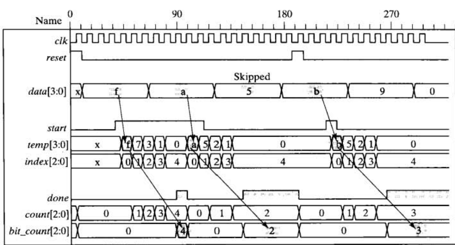


FIGURE 6-64 Simulation of *count_ones_SD*.

6.13.5 State-Machine Replacements for Unsynthesizable Loops

Synthesis tools cannot support nonstatic loops that do not have embedded timing controls. Such machines are not directly synthesizable, but their loop structures can be replaced by equivalent synthesizable sequential behavior. The key is to describe the behavior by an explicit finite state machine.

Example 6.42

The ASMD chart in Figure 6-65 describes a state machine that counts the 1s in a word and terminates activity as soon as possible. The machine remains in its reset state, *S_idle*, until an external agent asserts *start*. This action asserts the Mealy output, *load_temp*, which will cause *data* to be loaded into register *temp* when the state makes a transition to *S_counting* at the next active edge of *clk*. The machine remains in *S_counting* while *temp* contains a 1. Two actions occur concurrently at each subsequent clock: (1) *temp* is shifted toward its LSB and (2) *temp[0]* is added to *bit_count*. When *temp* finally has a 1 in only the LSB, the machine's state moves to *S_waiting*, where *done* is asserted as a Moore output. The state remains in *S_waiting* until *start* is de-asserted. The branches of the ASM and datapath (ASMD) chart show the control signals that are generated by the controller and are annotated with the register operations of the machine. Implicitly, those registers must remain unchanged for state transitions that traverse the other branches.

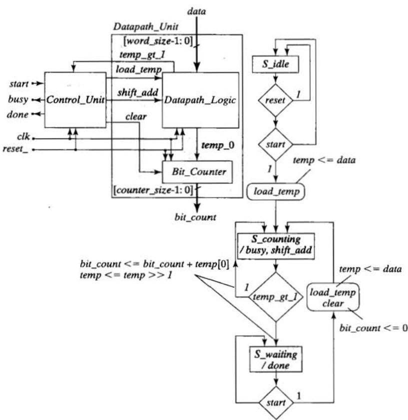


FIGURE 6-65 Block diagram and ASMD chart of *count_ones_SM*.

The Verilog sequential machine *count_ones_SM* avoids the problem of having to synthesize a nonstatic loop. The waveforms of the behavioral model, shown in Figure 6-66, demonstrate that the machine terminates as soon as *temp* is detected to be empty of 1s. The circuit synthesized from *count_ones_SM* is shown in Figure 6-67. The machine's datapath includes logic to hold and shift the data, and additional logic to count the ones in the data. The schematics shown in Figure 6-67 show the partition of the datapath unit to reduce the complexity of the drawings. Note that the signals *busy* and *shift_add* are identical outputs of the controller. This is a consequence of the ASMD chart's assertion of *busy* and *shift_add* in only state *S_counting*. Both signals were preserved in the synthesis process, even though they are identical, because both

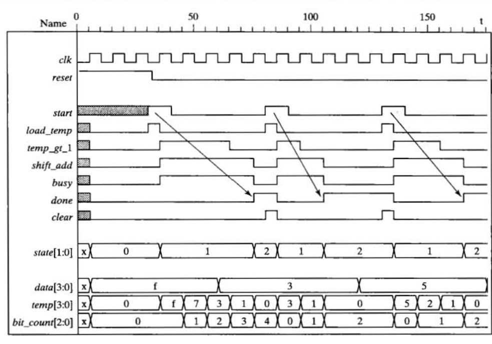


FIGURE 6-66 Simulation of *count_ones_SM* demonstrating behavior equivalent to a nonstatic loop.

are ports of the controller. The machine can be simplified to remove one of the signals from the port.

```
module count_ones_SM #( parameter word_size = 4, counter_size = 3)(
  output [counter_size -1: 0]        bit_count,
  output                             busy, done,
  input  [word_size -1: 0]           data,
  input                              start, clk, reset
);
  wire   load_temp, shift_add, clear;
  wire   temp_gt_1;

  Control_Unit    M0 (busy, load_temp, shift_add, clear, done, start, temp_gt_1, clk,
    reset);
  Datapath_Unit   M1 (bit_count, temp_gt_1, data, load_temp, shift_add, clear, clk,
    reset);
endmodule

module Control_Unit (
  output reg busy, load_temp, shift_add, clear, done,
  input start, temp_gt_1, clk, reset
);
  parameter state_size = 2'd2;
  parameter S_idle = 2'd0;
```

```
parameter S_counting = 2'd1;
parameter S_waiting = 2'd2;
reg bit_count;
reg [state_size-1 : 0] state, next_state;

always @ (posedge clk) // state transitions
 if (reset) state <= S_idle; else state <= next_state;

always @ (state, start, temp_gt_1) begin
 load_temp = 0;                          // defaults - assign by exception
 shift_add = 0;
 clear = 0;
 done = 0;
 busy = 0;
 next_state = S_idle;
 case (state)
  S_idle:       if (start) begin next_state = S_counting; load_temp = 1; end
  S_counting:   begin busy = 1;
                    if (temp_gt_1) begin
                    next_state = S_counting;
                    shift_add = 1; end
                  else begin next_state = S_waiting; shift_add = 1; end
                end
  S_waiting:    begin
                  done = 1;
                  if (start) begin next_state = S_counting; load_temp = 1; clear = 1; end
                  else next_state = S_waiting;
                end
  default:      begin clear = 1; next_state = S_idle; end
 endcase
end
endmodule

module Datapath_Unit #(parameter word_size = 4, counter_size = 3)(
 output reg [counter_size -1: 0] bit_count,
 output temp_gt_1,
 input [word_size -1: 0] data,
 input load_temp, shift_add, clear, clk, reset);
 reg [word_size-1: 0]      temp;
 assign temp_gt_1 = (temp > 1);
 wire   temp_0 = temp[0];

 always @ (posedge clk) // register transfers for datapath logic
  if (reset) begin temp <= 0; end
  else begin
   if (load_temp) temp <= data;
   if (shift_add) begin temp <= temp >>1; end
  end

 always @ (posedge clk) // counter of ones
  if (reset == 1'b1 || clear == 1'b1) bit_count <= 0;
  else bit_count <= bit_count + temp_0;
endmodule
```

End of Example 6.42

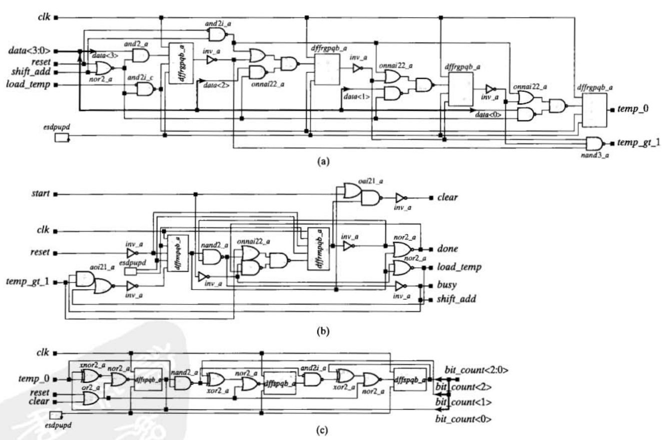


FIGURE 6-67 Circuit synthesized from a partitioned version of *count_ones_SM*: (a) datapath logic, (b) control unit, and (c) counter logic.

Note that in the previous example all of the outputs in the level-sensitive behavior were assigned value at the beginning of the behavior. Then only changes were assigned in subsequent states. This style makes the code more readable and prevents synthesis of unwanted latches, which will happen if a variable in a level-sensitive behavior is assigned value in some, but not all, of the paths of the activity flow through the behavior.

Example 6.43

Our final example of a machine that counts the 1s in a word of data is described by an implicit state machine. Its behavior (see Figure 6-68) is equivalent to that of a nonstatic loop, but it is synthesizable (see Problem 42 at the end of this chapter).

```
module count_ones_IMP #(parameter counter_size = 3, word_size = 3)(
  output reg [counter_size -1: 0] bit_count, output reg start, done,
  input [word_size -1: 0] data, input data_ready, clk, reset
);
  parameter                                   state_size = 2;
  reg          [state_size-1 : 0]             state, next_state;
  reg          [word_size-1: 0]               temp;

  always @ (posedge clk)
    if (reset) begin temp<= 0; bit_count <= 0; done <= 0; start <= 0; end
    else if (data_ready && data && !temp) begin
      temp <= data; bit_count <= 0; done <= 0; start <= 1; end
    else if (data_ready && (!data) && done) begin bit_count <= 0; done <= 1; end
    else if (temp == 1)
    begin bit_count <= bit_count + temp[0]; temp <= temp >> 1; done <= 1; end
    else if (temp && !done)
    begin start <= 0; temp <= temp >> 1; bit_count <= bit_count + temp[0]; end
endmodule
```

End of Example 6.43

6.14 Design Traps to Avoid

In general, avoid referencing the same variable in more than one cyclic (***always***) behavior. When variables are referenced in more than one behavior, there can be races in the software, and the post-synthesis simulated behavior may not match the pre-synthesis behavior. Never assign value to the same variable in multiple behaviors.

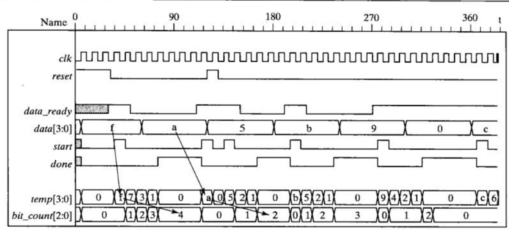


FIGURE 6-68 Simulation results for *count_ones_IMP*, an implicit state machine having behavior equivalent to a nonstatic loop.

6.15 Divide and Conquer: Partitioning a Design

Very large-scale integrated (VLSI) circuits commonly contain more gates (several million) than a synthesis tool can accommodate. It is standard practice to partition such circuits hierarchically into small functional units, which have a manageable complexity. This partitioning is done top-down, across one or more layers of hierarchy. Synthesis tools commonly synthesize circuits with about 10,000 to 50,000 gates, but beyond that the return diminishes as the run-time becomes large. Following decomposition, the lowest-level modules of the design can be synthesized individually. It is easier to modify the lower-level modules to make the design more amenable to synthesis and then to work with large modules.

Structural modeling composes a circuit by connecting primitive gates to create a specified functionality (e.g., an adder) just as parts are connected on a chip or a circuit board. But gate-level models are not necessarily the most convenient or understandable models of a circuit, especially when the design involves more than a few gates. Many modern ASICs have several million gates on a single chip! Also, truth tables become unwieldy when the circuit has several inputs, limiting the utility of UDPs. Architectural partitioning forms a structural model, but the functional units in the architecture have much more complexity than basic combinational logic gates and are modeled behaviorally.

Partitioning is not done randomly. Skill and experience play a role, and the designer's choice of hierarchical boundaries can have a strong impact on the quality of the synthesis product, and the cost of the effort. Partitioning the design into smaller

functional units can improve the readability of the description, improve the synthesis result, shorten the optimization cycle, and simplify the synthesis process.

In general, a design should be partitioned along functional lines into smaller functional units, each with a common clock domain, and each of which is to be verified separately. The hierarchy of the design should separate the clock domains, thereby clarifying the interaction between multiple clocks and revealing the need for synchronizer circuits. The logic in each clock domain can be verified separately, before integrating the system.

Functionally related logic should be grouped within a partition, so that the synthesis tool will be able to exploit opportunities for sharing logic, with a minimum of routing between blocks. If a module is used in multiple places in the design, it should be optimized separately for area and then instantiated as needed. This strategy will result in an overall design that is very efficient in its use of area.

It is also recommended that a module contain no more than one state machine. This will allow the synthesis tool to optimize the logic for a machine without the influence of extraneous logic. Logic in different clock domains (e.g., with interacting state machines) should be encapsulated in separate blocks of the partition. Synchronizers should be used where signals cross between the domains.

The partition of a design should group registers and their logic, so that their control logic might be implemented efficiently. Otherwise, splitting registers and logic across boundaries of the partition might lead to extra/duplicate control logic. Place the combinational logic driving the datapath of a register in the same module as the destination register. Likewise, any glue logic between module boundaries should be included within a module. If glue logic sits outside the modules, it cannot be absorbed by either of them.

Module boundaries are preserved in synthesis (i.e., the contents of the modules are optimized separately) so combinational logic should not be distributed between modules. Placing the logic in a single module will allow the synthesis tool to achieve the maximum exploitation of common logic. Do not include clocks trees, input–output pads, and test registers (see Chapter 11) in a design that is to be synthesized. Add them to the design after synthesis.

REFERENCES

1. De Micheli G. *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.
2. Gajski D, et al. *High-Level Synthesis*. Boston, MA: Kluwer, 1992.
3. Bartlett K, et al. "Multilevel Logic Minimization Using Implicit Don't-Cares." *IEEE Transactions on Computer Aided Design of Integrated Circuits, CAD-5*, 723–740, 1986.
4. Brayton RK, et al. "MIS: A Multiple-Level Interactive Logic Optimization System." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, CAD-6*, 1062–1081, 1987.

5. Brayton RK, et al. *Logic Minimization Algorithms for VLSI Synthesis*. Boston, MA: Kluwer, 1984.
6. Katz RH., Borriello, G. *Contemporary Logic Design*. Upper Saddle River, NJ: Prentice-Hall, 2004.
7. Wakerly JF. *Digital Design Principles and Practices*, 3rd ed. Upper Saddle River, NJ: Prentice-Hall, 2000.
8. Ciletti MD. *Modeling, Synthesis and Rapid Prototyping with the Verilog HDL*. Upper Saddle River, NJ: Prentice-Hall, 1999.

PROBLEMS

1. Synthesize the universal shift register that was presented in Example 5.45. Verify that the waveforms produced by simulation of the synthesized circuit match those of the behavioral model.
2. Synthesize *NRZ_2_Manchester_Mealy* (see Section 6.6.2) and verify that the post-synthesis simulation results match those shown in Figure 6-34 for the behavioral model. Note the time delays that result from the physical cells.
3. Synthesize *NRZ_2_Manchester_Moore* (see Section 6.6.3) and verify that the post-synthesis simulation results match those shown in Figure 6-37 for the behavioral model. Note the time delays that result from the physical cells.
4. The sequence recognizers described by the ASM charts in Figure 6-38 are *nonresetting*—they assert after three successive 1s are received and continue to assert until a 0 is detected. Develop ASM charts for *resetting* Mealy and Moore machines that will detect three successive 1 in a serial bit stream, assert their output, and then return to the reset state, *S_idle*, where the outputs are de-asserted, before processing additional bits. Develop and verify a Verilog model of each machine. Synthesize the machines and verify that the functionality of each synthesized machine matches that of its behavioral model.
5. Develop ASM charts for resetting Mealy and Moore machines that will detect the pattern 10101010_2 in a serial bit stream (with the LSB arriving first), assert their output, and then return to the reset state, *S_idle*, where the outputs are de-asserted, before processing additional bits. Develop and verify a Verilog model of each machine. Synthesize the machines and verify that the functionality of each synthesized machine matches that of its behavioral models.
6. Develop ASM charts for nonresetting Mealy and Moore machines that will detect the pattern 0010_2 in a serial bit stream, with the LSB arriving first. Synthesize the machines and verify that the functionality of each synthesized machine matches that of its behavioral model.
7. Develop ASM charts for nonresetting Mealy and Moore machines that will detect the patterns 0111_2 or 1000_2 in a serial bit stream, with the LSB arriving first. Synthesize the machines and verify that the functionality of each synthesized machine matches that of its behavioral model.
8. Develop an ASM chart for a nonresetting sequential machine that implements the majority function. The machine is to assert *D_out* if the serial input, *D_in*, contains two or more 1s in the last 3 bits. Synthesize the machines and verify that the functionality of each synthesized machine matches that of its behavioral model.

9. Critique the following code.

```
module clock_Prog (clk, Pulse_Width, Latency, Offset);
 input Pulse_Width, Latency, Offset;
 output clk;
 reg clk, Pulse_Width, Latency, Offset;
 parameter Pulse_Width = 5;
 parameter Latency = 5;
 parameter Offset = 10;
 parameter a_cycle = Pulse_Width;
 //parameter max_time=1000;
 initial
  clk = 0;
 always begin
  #a_cycle clk = ~clk;
  end
  //initial
  // #max_time $finish;
endmodule
```

10. Explain whether the circuit described below implements combinational logic with, or without, a latched output.

```
module or4_something #(parameter word_length = 4)(
 output reg y,
 input [word_length - 1: 0] x_in
);
 integer k;

 always @ x_in begin
  y = 0;
  if (x_in[0] == 1) y = 1;
  else if (x_in[1] == 1) y = 1;
  else if (x_in[2] == 1) y = 1;
 end
endmodule
```

11. The machine *Seq_Rec_3_1s_Mealy* in Example 6.26 uses a binary code for the state assignment. Synthesize an alternative machine using a one-hot code. Compare the new machine to the original machine. Discuss the trade-offs.
12. The machine *Seq_Rec_3_1s_Moore* in Section 6.6.4 partially decodes the possible state codes and assigns *next_state* = *S_idle* by default. Synthesize an alternative machine using the default assignment *next_state* = 3'bx. Compare the result to the original machine and determine whether assigning *next_state* = 3'bx actually reduced the logic that was synthesized.
13. Synthesize *count_ones_b0, count_ones_b1 and count_ones_d* (see Examples 6.39 and 6.41) and compare the results.

14. Use (a) ***while***, and (b) ***forever*** loops to develop implementations of *count_ones_SD* (see Example 6.41). Will either of these synthesize?
15. Develop, verify, and synthesize *count_ones_max_string*, a sequential machine whose output is the length of the largest string of consecutive 1s in 32 bits of serial data. Action is to begin when *Start* is asserted; *Done* is to assert when the string is found. Consider the possibility of early termination of the search if the length of a string of found consecutive 1s exceeds the size of the sub-word of remaining bits. The machine is to be partitioned into a datapath and a controller. Provide an ASMD chart and simulation results (pre- and post-synthesis).
16. Develop, verify, and synthesize *count_gray_bin*, a 4-bit counter that can count in Gray or in binary code, depending on an input mode.
17. Design and synthesize a sequential machine whose inputs are *clk* and *reset*, and whose outputs are *clk_by_4* and *clk_by_8*, (i.e., clock divider outputs that divide *clk* by 4 and 8, respectively).
18. Determine whether a synthesis tool detects and removes equivalent states from the sequential machine described by the STG in Figure P6-18.

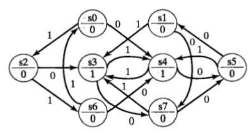


FIGURE P6-18

19. Synthesize *pipe_2stage*, described by the ASMD chart in Example 5.39. Verify that the synthesized circuit has the correct behavior.
20. Synthesize a circuit that will detect an illegal BCD-encoded word.
21. Digital switches are used in telephone central offices to sample local analog voice signals, convert them to digital signals, and multiplex them with other similar signals for digital transmission on the global phone network [7]. Figure P6-21(a) shows a configuration in which analog signals are sampled at a rate of 8000 samples per second, with each sample represented as an 8-bit word. To achieve a substantial savings in copper wire and other circuitry, thirty-two 64-kbps voice channels are multiplexed onto a serial channel having a bandwidth of 2.048 Mbps. Each bit of a sample has a time of 488 ns, the period of the signal *clock_488ns*, and the words of the sampled signals are interleaved in a 32-byte frame, with each frame having a length of 125 μs, as shown in Figure P6-21(b). A frame is synchronized by a pulse of *frame_synch*, which occurs in the bit time immediately before the first byte slot of the frame. At the receiving end of the serial line an 8-bit shift register receives the digitally encoded bits of *Serial_in* sequentially, beginning with the LSB and ending with the MSB. Then the byte is loaded into a holding register that drives a demultiplexer In the "double-buffered" scheme shown in Figure P6-21(a) the output of the demultiplexer has a latency of 9 cycles of *clock_488ns*—8 cycles to load a shift register with a byte

of data, and one cycle to load the byte into the output register driving the demultiplexer. Holding the output in a register keeps the data available for 8 cycles of *clock_488ns* instead of for only one cycle. Because the period of the frame synchronizer is 125 μs and the total bit time of 32 bytes is 124.92 μs, there is a gap of 72 μs between the end of the last byte and the end of the frame. The action of *frame_synch* is to keep the machine synchronized to the time

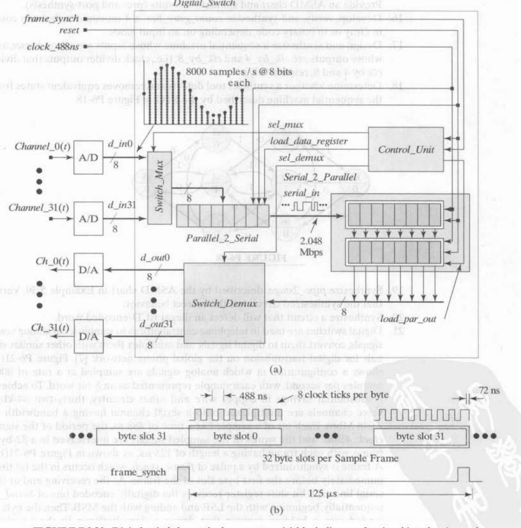


FIGURE P6-21 Digital switch for a telephone system: (a) block diagram for signal interleaving and (b) synchronized frame format.

reference of the inbound data, and to eliminate the effect of the gap. With synchronization, the last cycle of *clock_488ns* is elongated by 72 ns. The registers in the machine have active-high asynchronous reset. Each frame of data transmission is to be launched by the negative edge of *frame_synch*. Note: the action of *reset* will reset the registers of the machine, but will not synchronize the activity of the machine because *frame_synch* is used to reset the control unit.

Develop, verify (before and after synthesis), and synthesize a Verilog module that encapsulates the functionality shown in Figure P6-21(a), where the outputs of the A/D converters are inputs to a module that interleaves the sample bytes, with separate submodules for the control unit, the mux, the demux, the parallel to serial converter, and the serial to parallel converter. Define additional interface signals as needed to complete the design. Model the multiplexer so that its outputs will be registered. Carefully document your work.

22. Synthesize a Verilog description of the combinational logic described by the following Boolean function. Compare the schematic of the synthesized circuit to (a) that of the original circuit and to (b) a simplified version of the function obtained by using Karnaugh maps.
    $f(a, b, c, d) = \Sigma\, m(0, 2, 5, 7, 8, 10, 13, 15)$
23. Develop a behavioral model that implements the functionality described by *Divide_by_11* (see Problem 4.18). Synthesize the circuit and compare the result to the circuit synthesized from the structural model shown in Figure P4-18.
24. Under what conditions will a synthesis tool create combinational logic?
25. Under what conditions will a synthesis tool create a circuit that implements a transparent latch?
26. Under what conditions will a synthesis tool create an edge-triggered sequential circuit?
27. Discuss how to describe a synchronous reset condition using Verilog.
28. Synthesize and verify a cell-based implementation of the circuit described by *compare_4_32_CA* (see Problem 12 in Chapter 5).
29. Synthesize and verify a cell-based implementation of the circuit described by *compare_4_32_ALGO* (see Problem 13 in Chapter 5).
30. Synthesize and verify a cell-based implementation of the ring counter described in Problem 24a in Chapter 5.
31. Synthesize and verify a cell-based implementation of the ring counter described in Problem 24b in Chapter 5.
32. Synthesize and verify a cell-based implementation of the sequential machine described in Problem 26 in Chapter 5.
33. Synthesize and verify a cell-based implementation of the 8-bit ALU described in Problem 27 of Chapter 5.
34. Synthesize and verify a cell-based implementation of the sequential machine described in Problem 28 in Chapter 5.
35. Synthesize and verify a cell-based implementation of the sequential machine described by the ASMD chart in Figure 5.24.
36. Synthesize and verify a cell-based implementation of the keypad scanner described in Problem 32 in Chapter 5.
37. Synthesize and verify a cell-based implementation of the programmable pattern generator described in Problem 34 in Chapter 5.
38. Synthesize and verify a cell-based implementation of *Hamming_Encoder_MD3*, described in Problem 35 in Chapter 5.

39. Synthesize and verify a cell-based implementation of the synchronizer circuit described in Problem 36 in Chapter 5.
40. Synthesize and verify a cell-based implementation of the transparent latch described in Problem 37 in Chapter 5.
41. Develop, verify, and synthesize *Binary_Counter_Imp*, an implicit-state machine implementing a 4-bit counter by executing a register transfer operation (*count* <= *count* + 1) conditionally in every clock cycle, depending on an input signal *enable*.
42. Synthesize *count_ones_IMP_gates* and verify that the functionality of the synthesized circuit matches that of the behavioral model, *count_ones_IMP*, given in Example 6.42.
43. A token ring local area network (LAN) consists of a set of computers organized in a ring topology, with each machine connected to a bus by a LAN adapter (see Figure P7-3). LAN-based machines communicate by transmitting and receiving packets of bits containing encoded source and destination addresses, and a message. The LAN adaptor executes handshaking protocols by decoding the address in a received packet and determining whether its host machine is the recipient of the message or whether to pass the message on to the next machine on the LAN. It also transmits packets from the host machine to a target machine. The protocol for the network specifies how the start of a packet will be recognized, and how the source and destination addresses are encoded. The adaptor must recognize the start sequence, decode the address, and acknowledge that it is the recipient.

    The protocol for the token ring shown in Figure P6-43 marks the start of a packet by two successive 0s. Suppose a LAN adapter is hardwired to recognize either the address 100_2 or 010_2. The adapter is to assert a Moore-type output, *P_IN*, if a packet arrives and decodes to the adaptor's addresses. Develop

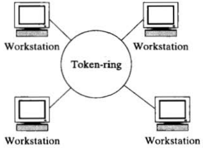


FIGURE P6-43

an ASM chart for the packet detection and address decoding circuit, write and verify a Verilog model of the machine, and synthesize the machine. Note whether the synthesis process has detected and/or removed any equivalent states. Explore alternative state assignments.

44. The rotor of a conventional motor rotates continuously when power is applied. The rotor of a stepper motor can be controlled to move a prescribed number of steps or to rotate continuously at a prescribed speed. Stepper motors are widely used in computer numerical control, where they control precision machine tools, and in personal computers, where they control floppy disk drives and control the paper feed of line printers, and in numerous other applications in which accurate positioning without feedback control is a requirement. A simple stepper motor has the configuration shown in Figure P6-44, in which a set of fixed stator coils form electromagnets located symmetrically about the perimeter of the motor assembly. A permanently magnetized rotor[31] is free to rotate on a central axis under the influence of the magnetic fields that are created when the coils are energized. The angular position of the rotor is controlled by sequentially energizing the coils to create a rotating magnetic field that exerts force on the poles of the rotor magnets. The stator coils can be energized sequentially to make the rotor move to a desired angular position or to rotate continuously at a selected speed. The angular acceleration of the rotor can be controlled by varying the spacing between the pulses that energize the coils. The motor in Figure P6-44 has only four stator coils, so the rotor can be held stable in any of eight different positions (states), with an angular resolution of 45 degrees, (the motor steps in increments of 45 degrees). Increasing the number of coils increases the resolution of the position of the rotor.[32] The coils are energized individually, to hold the rotor at the position of the energized stator, or in an adjacent/pairwise fashion, to orient the stator at an angle midway between two adjacent stators. For example, if coils *C0* and *C1* are both energized, the rotor will move from the position of *C0* to the position P1. Successively energizing (1) *C0*, (2) *C0* and *C1*, (3) *C1*, (4) *C1* and *C2*, and (5) *C2* will rotate the rotor clockwise to the position of *C2*. The speed at which the motor rotates is determined by the time interval between the inputs. Design and synthesize a Moore-type state machine to control the direction, speed, and angular acceleration of the motor's rotor, with inputs that program the machine to (1) advance by a specified number of steps in a given direction, (2) spin at a specified rate, and (3) accelerate from rest to a specified angular velocity (with stepwise increments in speed). Pulse rates from 1500 to 2500 pulses per second are allowed.

[31]Another class of stepper motors has a variable reluctance.

[32]Commercial motors have angular resolution ranging from 0.72 to 90 degrees.

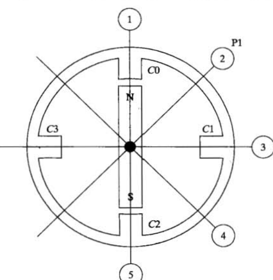


FIGURE P6-44 Rotor and stator configuration of a four-pole stepper motor.

45. The scheme shown in Figure 6-44 (c) generates registered Mealy outputs by forming them from the *next_state* rather than from the state of the machine. Modify the Verilog Model in Example 6.28 by replacing its level-sensitive cyclic behavior with two level-sensitive cyclic behaviors. One is to form next_state from state and *B_in*; the other is to form *B_out* from next_state and *B_in*. Verify that the output of the modified model is registered and matches the waveform of NRZI_Mealy in Figure 3-24 (with latency). Annotate the waveforms to show that the values of the registered output correspond to the present state and the input that caused the transition to the present state.
46. Explain the consequences of replacing procedural assignments by nonblocking assignments in Example 6.39 and Example 6.41.

CHAPTER 7

Design and Synthesis of Datapath Controllers

Digital systems range from those that are control-dominated to those that are data-dominated. Control-dominated systems are reactive systems responding to external events; data-dominated systems are shaped by the requirements of high-throughput data computation and transport, as in telecommunications and signal processing [1]. Consequently, sequential machines are commonly classified and partitioned into datapath units and control units. Examples in the previous chapters have included both.

Most datapaths include arithmetic units, such as arithmetic and logic units (ALUs), adders, multipliers, shifters, and digital signal processors, but some do not, such as graphics coprocessors. Datapath units consist of computational resources (e.g., ALUs and storage registers), logic for moving data through the system and between the computation units and the internal registers, and datapaths for moving data to and from the external environment. The datapath unit in Figure 7-1 is controlled by a finite-state machine (FSM) that coordinates the execution of instructions that perform operations on the datapath. Datapath units are characterized by repetitive operations on different sets of data, as in signal processing, image processing, and multimedia [2]. Architectures that are dominated by control units will generally have a significant amount of random (irregular) logic, together with some regular structures, like multiplexers for steering signals and comparators.

7.1 Partitioned Sequential Machines

Partitioning a sequential machine into a datapath and a controller clarifies the architecture and simplifies the design of the system. The process by which the machine is designed is said to be *application driven* because the sequence of operations that must be performed by the datapath unit in a particular application determines the resources

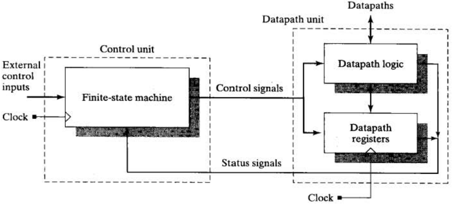


FIGURE 7-1 State-machine controller for a datapath.

composing its architecture, the set of instructions that must be executed by the datapath, and, ultimately, the FSM that controls the datapath.

The sequence of steps in an application-driven design process is illustrated in Figure 7-2. Once the architecture of the datapath unit has been selected to support the instruction set of an application, sequences of operations (control states) that support the instruction set can be identified. The control states are used to schedule assertions of the signals that control the movement and manipulation of data as the machine executes instructions. Then an FSM can be designed to generate the control signals. In this section, we will illustrate the design of datapath controllers for some simple functional units to prepare for the design of a stored-program reduced instruction-set computer (RISC) in the next section.

Control units orchestrate, coordinate, and synchronize the operations of datapath units. The control unit of a machine generates the signals that load, read, clear, and shift the contents of storage registers; fetch instructions and data from memory; store data in memory; steer signals through muxes; control three-state devices; and control the operations of ALUs and other complex datapath units. In simple synchronous machines, a common clock[1] synchronizes the activities of the controller and datapath functional units. Note that the control unit in Figure 7-1 is implemented as an FSM, and is itself controlled by external input signals (primary input signals) and by status signals from the datapath unit. The FSM produces the signals that control the operation of the datapath unit.

Datapath units are commonly described by dataflow graphs; control units are commonly modeled by state transition graphs and/or algorithmic-state machine (ASM) charts for FSMs. Partitioned sequential machines can be modeled by an FSM and datapath (FSMD), a combined control-dataflow graph, which expresses datapath operations

[1]More complex machines may have multiple clocks that synchronize particular activities of the machine.

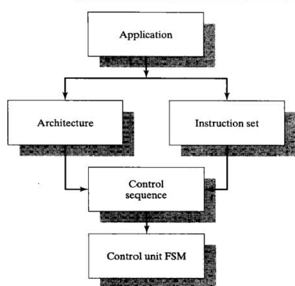


FIGURE 7-2 Application-driven architecture, instruction set, and control sequence for a datapath controller.

in the context of a state-transition graph (STG). As noted in Chapter 5, we favor using an ASM and datapath (ASMD) chart, which likewise links an ASM chart for a control unit to the operations of the datapath that it controls.

7.2 Design Example: Binary Counter

Consider a synchronous 4-bit binary counter that is to be incremented by a count of 1 at each active edge of the clock, and whose count is to wrap around to 0 when the count reaches 1111_2. As a first view of the machine, we could describe the counter by an implicit state machine, *Binary_Counter_Imp*, executing a register transfer operation (*count* <= *count* + 1) conditionally, in every clock cycle, depending on *enable*, and then synthesize a hardware realization directly.[2] A second approach is to partition the machine into an architecture of separate datapath and control units, as shown in Figure 7-3 for *Binary_Counter_Arch*, with a specific architecture for the datapath.

The functional elements of the architecture of the datapath unit consist of (1) a 4-bit register to hold *count*, (2) a mux that steers either *count* or the sum of *count* and 0001_2 to the input of the register, and (3) a 4-bit adder to increment *count*. The signal *enable* must be asserted for counting to occur, and the signal *rst* overrides all activity and drives *count* to a value of 0000_2. The input *rst* must be de-asserted and *enable* must be asserted for the machine to begin counting and to continue counting. The control

[2]See Example 5.41.

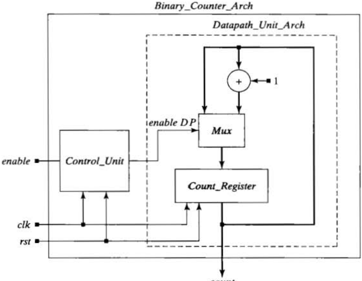


FIGURE 7-3 Architecture for a synchronous 4-bit binary counter.

unit for this simple machine passes *enable* directly to the datapath unit as the signal *enable_DP*. The signals *clk* and *rst* are shown in Figure 7-3 to accommodate a controller for a more general specification of a counter.[3]

Next, consider a third alternative in which the counter itself is an explicit-state machine, *Binary_Counter_STG*, having state *count* (the contents of the counter), and inputs *enable, clk*, and *rst*. The simplified STG of the machine is shown in Figure 7-4, with *count* entered as the state label within each node. (The reset-directed arcs are not shown, nor are the arcs that return to the current state if *enable* is not asserted.) The STG can be used to develop an explicit state machine with two cyclic behaviors, one defining the next-state/output combinational logic and the other synchronizing the state transitions. Note that this approach to the design of the counter/machine can become unwieldy, because the size of the graph increases with the width of the datapath. It is often true that the number of states of the datapath registers is enormous compared to the number of states of the control unit. Partitioning the design eliminates the need to consider the state of the datapath registers, except to generate status signals that are fed back to the control unit (e.g., a signal indicating that *count* has a particular value).

The preceding three views of a 4-bit binary counter illustrate how partitioning a sequential machine into a datapath and a controller can reduce the size of the state that needs to be considered in the design and simplify the control unit of the machine. Because the machine is simple, the task of writing and synthesizing a behavioral model is easy; managing the design of more complex machines becomes unwieldy unless the overall

[3]See Problem 7-4 at the end of the chapter.

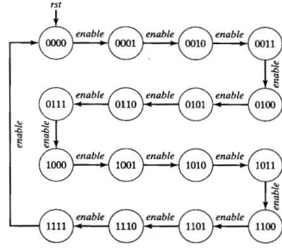


FIGURE 7-4 Simplified STG for a synchronous 4-bit binary counter.

machine is partitioned into a datapath unit and a control unit. In this example, the implicit state machine has the simplest description; it expresses datapath operations in terms of language-based operators and suppresses explicit structural detail of the datapath, leaving it to the synthesis tool. The partitioned machine has the most structural detail, a simple controller, and a datapath architecture having a register whose state does not influence the design; the STG-based approach required a detailed STG and led to a state machine with 16 states, because the state of the machine was the state of the register holding *count*. This approach is unattractive because even a simple resizing of the datapath requires redesigning the controller. The relative complexity of the Verilog models and the synthesized hardware are trade-offs between the equivalent, but alternative machines.[4]

Yet another view of the binary counter is based on the counter's computational *activity*. The machine is described by mixing the ASM chart with datapath operations, as shown in Figure 7-5. *Binary_Counter_ASM* has one state, *S_running*.[5] At every clock, *enable* is tested and a transition is made back to *S_running*; if *enable* is asserted, a conditional register operation that increments the counter is executed concurrently with the state transition. The ASM chart describes the activity of more complex counters and other single-cycle machines that have a different relationship governing the register operations. For example, the function *next_count* could describe a Johnson counter.[6] Note that the description does not require an explicit state register because it remains in the same state. In fact, it reduces to the implicit state machine described above. *Avoid this practice of mixing datapath operations with an ASM chart* because (1) the control signals that cause the datapath operations are not explicitly identified and (2) the practice can

[4]See Problems 10 and 11 at the end of this chapter.
[5]This machine is actually the same as *Binary_Counter_Imp* mentioned above.
[6]See Problem 17 in Chapter 5.

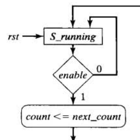


FIGURE 7-5 ASM chart for a synchronous 4-bit binary counter.

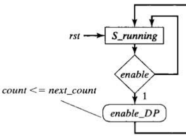


FIGURE 7-6 ASMD chart for a datapath unit, a synchronous 4-bit binary counter, controlled by a state machine.

create confusion by suggesting that the datapath operation is part of the controller. A more clear distinction between the control unit and the datapath unit can be made.

A fifth approach to designing the binary counter partitions the machine into a control unit and a datapath unit, but designs a register transfer level (RTL) behavioral model for the datapath unit, rather than a structural model (as we did in Figure 7-3). This style separates the design of the control unit from the design (and synthesis) of the datapath unit and simplifies the task of describing the datapath unit.

We separate the unit that determines *what* happens to the data from the unit that determines *when* it happens. This effort might seem like overkill for this counter (and it is), but the style is critical to successful design and synthesis of more complex machines and is easily implemented in Verilog. Figure 7-6 shows an ASMD chart for *Binary_Counter_Part_RTL*, a counter partitioned into a datapath unit and a control unit, with signal *enable_DP* linking the controller to the datapath. The model's control unit passes *enable* through to the datapath unit. The state machine of the controller consists of only the pass-through logic. (In general, a controller consisting of only combinational logic can be absorbed by the datapath unit to form an implicit state machine.) The ASM chart identifies the inputs and outputs of the controller and is annotated with the register operations that are caused by the output signals, i.e., assertion of the Mealy output signal *enable_DP* controls the operation *count* <= *next_count*. The operation executes at the clock edges that cause the state transition of the machine.

Example 7.1

The Verilog description *Binary_Counter_Part_RTL* has two nested modules: *Control_Unit* and *Datapath_Unit*. The datapath has been described with flexibility to implement codes for other counters, independently of the control unit.[7] The simulation results in Figure 7-7 show that counting begins at the first rising edge of *clk* after *enable* is asserted, that counting continues while *enable* is asserted, and that *enable_DP* replicates *enable*. The machine also recovers from a reset on-the-fly condition.

```
module Binary_Counter_Part_RTL #(parameter size =4) (
  output [size -1: 0]        count,
  input                      enable,
  input                      clk, rst
);
  wire                       enable_DP;

  Control_Unit M0_Controller (enable_DP, enable, clk, rst);
  Datapath_Unit M1_Datapath (count, enable_DP, clk, rst);
endmodule

module Control_Unit (output enable_DP, input enable, clk, rst); // clk, reset not needed
   assign enable_DP = enable;
endmodule
```

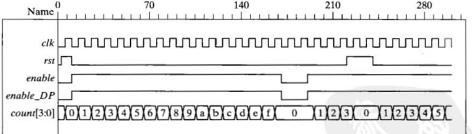


FIGURE 7-7 Simulation results for *Binary_Counter_RTL*, a synchronous 4-bit binary counter controlled by a state machine.

[7]We illustrate here the use of descriptive module instance names (*M0_Controller*, *M1_Datapath*) for the controller and the datapath. These names will appear explicitly in the design hierarchy displayed by a simulation or synthesis tool, making it easier to locate signals. However, for our simpler examples, we will use simpler names (e.g., *M0* and *M1*).

```
module Datapath_Unit #(parameter size = 4)(
 output reg [size-1: 0]    count,
 input                     enable,
 input                     clk, rst
);

 always @ (posedge clk)
  if (rst == 1) count <= 0;
   else if (enable == 1) count <= next_count(count);

 function        [size-1: 0]        next_count;
  input          [size-1: 0]        count;
  begin
   next_count = count + 1;
  end
 endfunction
endmodule
```

Next, we will redesign the counter to form *Binary_Counter_Part_RTL_by_3*, which increments its count every third clock cycle. Only the control unit must change. One approach is to model the control unit by the implicit Moore machine shown below. The simulated activity of the machine is shown in Figure 7-8.

```
module Control_Unit_by_3 (output reg enable_DP, input enable, clk, rst);
 always begin: Cycle_by_3
  @ (posedge clk) enable_DP <= 0;
  if ((rst == 1) || (enable != 1)) disable Cycle_by_3; else
   @ (posedge clk)
   if ((rst == 1) || (enable != 1)) disable Cycle_by_3; else
    @ (posedge clk)
    if ((rst == 1) || (enable != 1)) disable Cycle_by_3;
    else enable_DP <= 1;
 end // Cycle_by_3
endmodule
```

End of Example 7.1

The Verilog module *Binary_Counter_Part_RTL_by_3*, with the modified control unit, is synthesizable. We anticipate that two flip-flops will be needed to implement the implicit-state machine of the control unit, because the state evolves through three embedded clock cycles. One flip-flop will be needed to register *enable_DP*, and four flip-flops will be needed to implement the register holding *count* in the datapath unit. The synthesis results in Figure 7-9 confirm this use of resources for the control unit. However, the post-synthesis behavior of *Binary_Counter_Part_RTL_by_3* is problematic.[8]

[8]See Problem 10 at the end of this chapter.

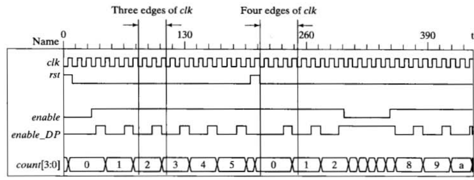


FIGURE 7-8 Simulation results for *Binary_Counter_Part_RTL_by_3* with a control unit to increment the datapath counter every third cycle.

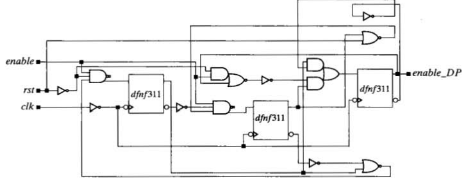


FIGURE 7-9 Circuit synthesized for *Control_Unit_by_3*, the control unit of *Binary_Counter_Part_RTL_by_3*, a 4-bit binary counter partitioned into a control unit and a datapath unit, with the counter incrementing every third clock cycle.

7.3 Design and Synthesis of a RISC Stored-Program Machine

RISCs are designed to have a small set of instructions that execute in short clock cycles, with a small number of cycles per instruction. RISC machines are optimized to achieve efficient pipelining of their instruction streams [2]. In this section, we will model a simple RISC machine. Our companion Web site (www.pearsonhighered.com/ciletti) includes the machine's source code and an assembler that can be used to develop programs for student projects. The machine also serves as a starting point for developing architectural variants and a more robust instruction set.

Designers make high-level trade-offs in selecting an architecture that serves an application. Once an architecture has been selected, a circuit that has sufficient performance (speed) must be synthesized to implement the machine. Hardware description languages (HDLs) play a key role in this process by modeling the system and serving as a descriptive medium that can be used by a synthesis tool.

As an example, the overall architecture of a simple RISC is shown in Figure 7-10. *RISC_SPM* is a stored-program RISC-architecture machine [3, 4]—its instructions are contained in a program stored in memory.

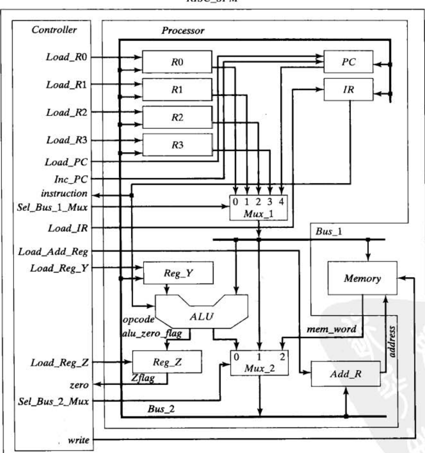


FIGURE 7-10 Architecture of *RISC_SPM*, an RISC stored-program machine (SPM).

The machine consists of three functional units: a processor (datapath unit), a controller (control unit), and memory. Program instructions and data are stored in memory. In program-directed operation, instructions are fetched synchronously from memory, decoded, and executed to (1) operate on data within the arithmetic and logic unit (*ALU*), (2) change the contents of storage registers, (3) change the contents of the program counter (*PC*), instruction register (*IR*), and the address register (*ADD_R*), (4) change the contents of memory, (5) retrieve data and instructions from memory, and (6) control the movement of data on the system busses. The instruction register contains the instruction that is currently being executed; the program counter contains the address of the next instruction to be executed; and the address register holds the address of the memory location that will be addressed next by a read or write operation.

7.3.1 RISC SPM: Processor

The processor includes registers, datapaths, control signals, and an ALU capable of performing arithmetic and logic operations on its operands, subject to the opcode held in the instruction register. A multiplexer, *Mux_1*, determines the source of data that is bound for *Bus_1*, and a second mux, *Mux_2*, determines the source of data bound for *Bus_2*. The input datapaths to *Mux_1* are from four internal general-purpose registers (*R0, R1, R2, R3*), and from the PC. The contents of *Bus_1* can be steered to the ALU, to memory, or to *Bus_2* (via *Mux_2*). The input datapaths to *Mux_2* are from the *ALU*, *Mux_1*, and the memory unit. Thus, an instruction can be fetched from memory, placed on *Bus_2*, and loaded into the instruction register. A word of data can be fetched from memory and steered to a general-purpose register or to the operand register (*Reg_Y*) prior to an operation of the ALU. The result of an *ALU* operation can be placed on *Bus_2*, loaded into a register, and subsequently transferred to memory. A dedicated register (*Reg_Z*) holds a flag indicating that the result of an *ALU* operation is 0.[9]

7.3.2 RISC SPM: ALU

For the purposes of this example, the ALU has two operand datapaths, *data_1* and *data_2*, and its instruction set is limited to the following instructions:

Instruction	Action
ADD	Adds the datapaths to form *data_1* + *data_2*
SUB	Subtracts the datapaths to form *data_1* − *data_2*
AND	Takes the bitwise-and of the datapaths, *data_1* & *data_2*
NOT	Takes the bitwise Boolean complement of *data_1*

7.3.3 RISC SPM: Controller

The timing of all activity in the machine is determined by the controller. The controller must steer data to the proper destination, according to the instruction being executed. Thus, the design of the controller is strongly dependent on the specification of the

[9]This can be used to monitor a loop index.

machine's ALU and datapath resources and the clocking scheme available. In this example, a single clock will be used, and execution of an instruction is initiated on a single edge of the clock (e.g., the rising edge). The controller monitors the state of the processing unit and the instruction to be executed and determines the value of the control signals. The controller's input signals are the instruction word and the zero flag from the *ALU*. The signals produced by the controller are identified as follows:

Control Signal	Action
Load_Add_Reg	Loads the address register
Load _PC	Loads *Bus_2* to the program counter
Load_IR	Loads *Bus_2* to the instruction register
Inc_PC	Increments the program counter
Sel_Bus_1_Mux	Selects among the *Program_Counter, R0, R1, R2*, and *R3* to drive *Bus_1*
Sel_Bus_2_Mux	Selects among *Alu_out, Bus_1*, and memory to drive *Bus_2*
Load_R0	Loads general-purpose register *R0*
Load_R1	Loads general-purpose register *R1*
Load_R2	Loads general-purpose register *R2*
Load_R3	Loads general-purpose register *R3*
Load_Reg_Y	Loads *Bus_2* to the register *Reg_Y*
Load Reg_Z	Stores output of *ALU* in register *Reg_Z*
write	Loads *Bus_1* into the *SRAM* memory at the location specified by the address register

The control unit (1) determines when to load registers, (2) selects the path of data through the multiplexers, (3) determines when data should be written to memory, and (4) controls the three-state busses in the architecture.

7.3.4 RISC SPM: Instruction Set

The machine is controlled by a machine language program consisting of a set of instructions stored in memory. So, in addition to depending on the machine's architecture, the design of the controller depends on the processor's instruction set (i.e., the instructions that can be executed by a program). A machine language program consists of a stored sequence of 8-bit words (bytes). The format of an instruction of *RISC_SPM* can be long or short, depending on the operation.

Short instructions have the format shown in Figure 7-11(a). Each short instruction requires 1 byte of memory. The word has a 4-bit opcode, a 2-bit source register address, and a 2-bit destination register address. A long instruction requires 2 bytes of memory. The first word of a long instruction contains a 4-bit opcode. The remaining 4 bits of the word can be used to specify addresses of a pair of source and destination registers, depending on the instruction. The second word contains the address of the memory word that holds an operand required by the instruction. Figure 7-11(b) shows the 2-byte format of a long instruction.

(a)

Opcode				Source		Destination	
0	0	1	0	0	1	1	0

(b)

Opcode				Source		Destination	
0	1	1	0	1	0	don't care	don't care
Address							
0	0	0	1	1	1	0	1

FIGURE 7-11 Instruction format of (a) a short instruction and (b) a long instruction.

The instruction mnemonics and their actions are listed below.

Single-Byte Instruction	Action
NOP	No operation is performed; all registers retain their values. The addresses of the source and destination register are don't-cares, they have no effect.
ADD	Adds the contents of the source and destination registers and stores the result into the destination register.
AND	Forms the bitwise-and of the contents of the source and destination registers and stores the result into the destination register.
NOT	Forms the bitwise complement of the content of the source register and stores the result into the destination register.
SUB	Subtracts the content of the source register from the destination register and stores the result into the destination register.

Two-Byte Instruction	Action
RD	Fetches a memory word from the location specified by the second byte and loads the result into the destination register. The source register bits are don't-cares (i.e., unused).
WR	Writes the contents of the source register to the word in memory specified by the address held in the second byte. The destination register bits are don't-cares (i.e., unused).
BR	Branches the activity flow by loading the program counter with the word at the location (address) specified by the second byte of the instruction. The source and destination bits are don't-cares (i.e., unused).
BRZ	Branches the activity flow by loading the program counter with the word at the location (address) specified by the second byte of the instruction if the zero flag register is asserted.

The *RISC_SPM* instruction set is summarized in Table 7-1.

The program counter holds the address of the next instruction to be executed. When the external reset is asserted, the program counter is loaded with 0, indicating that the bottom of memory holds the next instruction that will be fetched. Under the action of the clock, for single-cycle instructions, the instruction at the address in the program counter is loaded into the instruction register and the program counter is incremented. An instruction decoder determines the resulting action on the datapaths and the ALU. A long instruction is held in 2 bytes and an additional clock cycle is

TABLE 7-1 Instruction set for the *RISC_SPM* machine.

Instr	Instruction Word			Action
	opcode	src	dest	
NOP	0000	??	??	none
ADD	0001	src	dest	dest <= src + dest
SUB	0010	src	dest	dest <= dest − src
AND	0011	src	dest	dest <= src & dest
NOT	0100	src	dest	dest <= ~src
RD*	0101	??	dest	dest <= memory [Add_R]
WR*	0110	src	??	memory[Add_R] <= src
BR*	0111	??	??	PC <= memory[Add_R]
BRZ*	1000	??	??	PC <= memory [Add_R]
HALT	1111	??	??	Halts execution until reset

*Requires a second word of data; ? denotes a don't-care.

required to execute the instruction. In the second cycle of execution, the second byte is fetched from memory at the address held in the program counter and then the instruction is completed. Intermediate contents of the ALU may be meaningless when two-cycle operations are being executed.

7.3.5 RISC SPM: Controller Design

The machine's controller will be designed as an FSM. Its states must be specified, given the architecture, instruction set, and clocking scheme used in the design. This can be accomplished by identifying what steps must occur to execute each instruction. We will use an ASM chart to describe the activity within the machine, *RISC_SPM*, and to present a clear picture of how the machine operates under the command of its instructions.

The machine has three phases of operation: *fetch*, *decode*, and *execute*. Fetching retrieves an instruction from memory, decoding decodes the instruction, manipulates datapaths, and loads registers; execution generates the results of the instruction. The fetch phase will require two clock cycles—one to load the address register and one to retrieve the addressed word from memory. The decode phase is accomplished in one cycle. The execution phase may require zero, one, or two more cycles, depending on the instruction. The *NOT* instruction can execute in the same cycle that the instruction is decoded; single-byte instructions, such as *ADD*, take one cycle to execute, during which the results of the operation are loaded into the destination register. The source register can be loaded during the decode phase. The execution phase of a 2-byte instruction will take two cycles (e.g., *RD*)—one to load the address register with the

second byte and one to retrieve the word from the memory location addressed by the second byte and load it into the destination register. The controller for *RISC_SPM* has the 11 states listed below, with the control actions that must occur in each state.

S_idle	State entered after reset is asserted. No action.
S_fet1	Load the address register with the contents of the program counter. (*Note*: *PC* is initialized to the starting address by the reset action.) The state is entered at the first active clock after reset is de-asserted, and is revisited after a *NOP* instruction is decoded.
S_fet2	Load the instruction register with the word addressed by the address register and increment the program counter to point to the next location in memory, in anticipation of the next instruction or data fetch.
S_dec	Decode the instruction register and assert signals to control data-paths and register transfers.
S_ex1	Execute the *ALU* operation for a single-byte instruction, conditionally assert the zero flag, and load the destination register.
S_rd1	Load the address register with the second byte of a *RD* instruction, and increment the *PC*.
S_rd2	Load the destination register with the memory word addressed by the byte loaded in *S_rd1*.
S_wr1	Load the address register with the second byte of a *WR* instruction, and increment the *PC*.
S_wr2	Write the data stored in the source register filed of the instruction register to the memory location loaded in the address register in *S_wr1*.
S_br1	Load the address register with the second byte of a *BR* instruction, and increment the *PC*.
S_br2	Load the program counter with the memory word addressed by the byte loaded in *S_br1*.
S_halt	Default state to trap failure to decode a valid instruction.

The partitioned ASM chart for the controller of *RISC_SPM* is shown in Figure 7-12, with the states numbered for clarity. Once the ASM charts have been built, the designer can write the Verilog description of the entire machine, for the given architectural partition. This process unfolds in stages. First, the functional units are declared according to the partition of the machine. Then their ports and variables are declared and checked for syntax. Then the individual units are described, debugged, and verified. The last step is to integrate the design and verify that it has correct functionality.

The top-level Verilog module *RISC_SPM* integrates the modules of the architecture of Figure 7-10 and will be presented first. Three modules are instantiated: *Processing_Unit, Control_Unit*, and *Memory_Unit*, with instance names *M0_Processor, M1_Controller*, and *M2_Mem*, respectively. The parameters declared at this level of the hierarchy size the datapaths between the three structural/functional units.

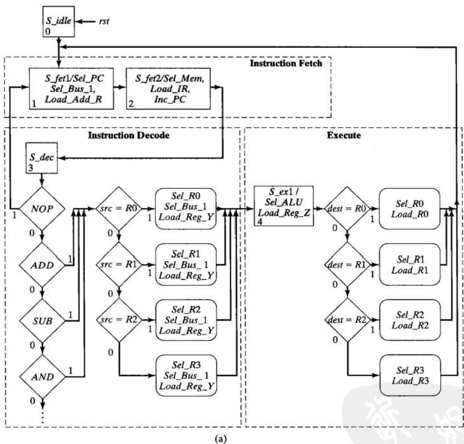
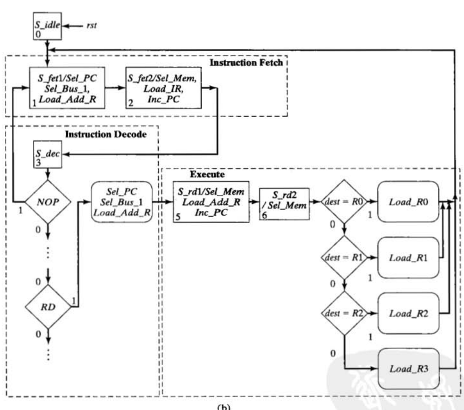
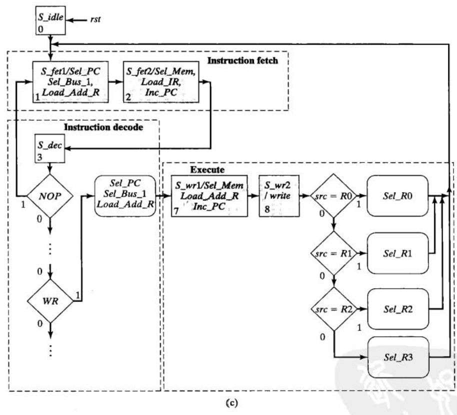


FIGURE 7-12 ASM charts for the controller of a processor that implements the *RISC_SPM* instruction set: (a) *NOP*, *ADD*, *SUB*, *AND*, (b) *RD*, (c) *WR*, (d) *NOT*, and (e) *BR*, *BRZ*.



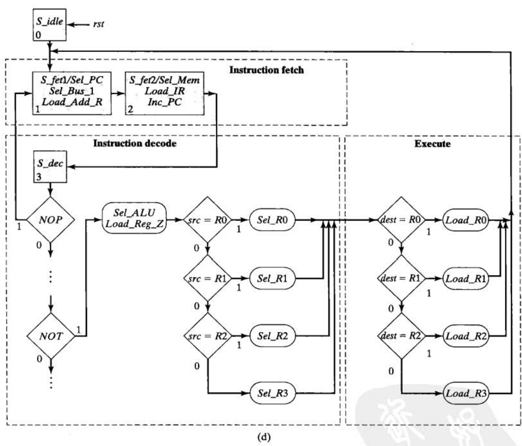
(b)

FIGURE 7-12 Continued



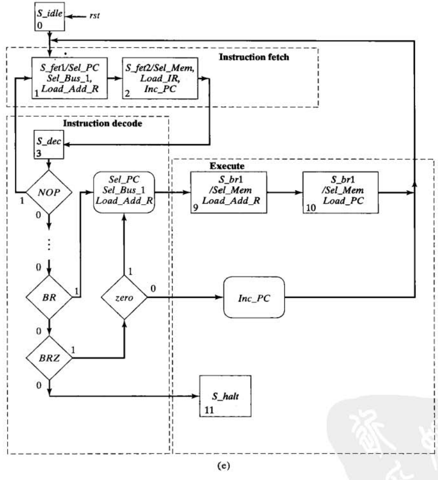
(c)

FIGURE 7-12 Continued



(d)

FIGURE 7-12 Continued



(e)

FIGURE 7-12 Continued

```
module RISC_SPM #(parameter word_size = 8, Sel1_size = 3, Sel2_size = 2)(
 input clk, rst
);
 // Data Nets
 wire [Sel1_size -1: 0] Sel_Bus_1_Mux;
 wire [Sel2_size -1: 0] Sel_Bus_2_Mux;
 wire zero;
 wire [word_size -1: 0] instruction, address, Bus_1, mem_word;

// Control Nets
 wire    Load_R0, Load_R1, Load_R2, Load_R3, Load_PC, Inc_PC; Load_IR,
         Load_Add_R, Load_Reg_Y, Load_Reg_Z, write;

 Processing_Unit M0_Processor (instruction, address, Bus_1, zero, mem_word,
   Load_R0, Load_R1, Load_R2, Load_R3, Load_PC, Inc_PC, Sel_Bus_1_Mux,
    Sel_Bus_2_Mux, Load_IR, Load_Add_R, Load_Reg_Y, Load_Reg_Z, clk, rst);

 Control_Unit M1_Controller (Sel_Bus_2_Mux, Sel_Bus_1_Mux, Load_R0,
   Load_R1, Load_R2, Load_R3, Load_PC, Inc_PC, Load_IR, Load_Add_R,
    Load_Reg_Y, Load_Reg_Z, write, instruction, zero, clk, rst);

 Memory_Unit M2_MEM (
  .data_out(mem_word),
  .data_in(Bus_1),
  .address(address),
  .clk(clk),
  .write(write) );
endmodule
```

The Verilog model of the machine's processor will describe the architecture, register operations, and datapath operations that are represented by the functional units shown in Figure 7-10. The processor instantiates several other modules, which must be declared too.

```
module Processing_Unit #(parameter
   word_size = 8, op_size = 4, Sel1_size = 3, Sel2_size = 2)(
   output [word_size -1: 0]    instruction, address, Bus_1,
   output                      Zflag,
   input [word_size -1: 0]     mem_word,
   input                       Load_R0, Load_R1, Load_R2, Load_R3, Load_PC,
                                Inc_PC,
   input [Sel1_size -1: 0]     Sel_Bus_1_Mux,
   input [Sel2_size -1: 0]     Sel_Bus_2_Mux,
   input                       Load_IR, Load_Add_R, Load_Reg_Y, Load_Reg_Z,
   input                       clk, rst
 );

   wire [word_size -1: 0]      Bus_2;
   wire [word_size -1: 0]      R0_out, R1_out, R2_out, R3_out;
```

```
  wire [word_size -1: 0]     PC_count, Y_value, alu_out;
  wire                       alu_zero_flag;
  wire [op_size -1 : 0]      opcode = instruction [word_size-1: word_size-op_size];

  Register_Unit              R0       (R0_out, Bus_2, Load_R0, clk, rst);
  Register_Unit              R1       (R1_out, Bus_2, Load_R1, clk, rst);
  Register_Unit              R2       (R2_out, Bus_2, Load_R2, clk, rst);
  Register_Unit              R3       (R3_out, Bus_2, Load_R3, clk, rst);
  Register_Unit              Reg_Y    (Y_value, Bus_2, Load_Reg_Y, clk, rst);
  D_flop                     Reg_Z    (Zflag, alu_zero_flag, Load_Reg_Z,
                                        clk, rst);
  Address_Register           Add_R    (address, Bus_2, Load_Add_R, clk, rst);
  Instruction_Register       IR       (instruction, Bus_2, Load_IR, clk, rst);
  Program_Counter            PC       (PC_count, Bus_2, Load_PC, Inc_PC,
                                        clk, rst);
  Multiplexer_5ch            Mux_1    (Bus_1, R0_out, R1_out, R2_out,
                                        R3_out, PC_count,
                                      Sel_Bus_1_Mux);
  Multiplexer_3ch            Mux_2    (Bus_2, alu_out, Bus_1, mem_word,
                                      Sel_Bus_2_Mux);
  Alu_RISC                   ALU      (alu_out, alu_zero_flag, Y_value, Bus_1,
                                        opcode);
endmodule

module Register_Unit #(parameter word_size = 8) (
 output reg [word_size-1: 0]  data_out,
 input [word_size -1: 0]      data_in,
 input                        load, clk, rst
);
 always @ (posedge clk, negedge rst)
  if (rst == 1'b0) data_out <= 0; else if (load) data_out <= data_in;
endmodule

module D_flop (output reg data_out, input data_in, load, clk, rst);
 always @ (posedge clk, negedge rst)
  if (rst == 1'b0) data_out <= 0; else if (load == 1'b1) data_out <= data_in;
endmodule

module Address_Register #(parameter word_size = 8)(
 output reg    [word_size -1: 0]  data_out,
 input         [word_size -1: 0]  data_in,
 input                            load, clk, rst

);
 always @ (posedge clk, negedge rst)
  if (rst == 1'b0) data_out <= 0; else if (load == 1'b1) data_out <= data_in;
endmodule

module Instruction_Register #(parameter word_size = 8)(
 output reg    [word_size -1: 0]  data_out,
 input         [word_size -1: 0]  data_in,
 input                            load, clk, rst
```

```
);
  always @ (posedge clk, negedge rst)
   if (rst == 1'b0) data_out <= 0; else if (load == 1'b1) data_out <= data_in;
endmodule

module Program_Counter #(parameter word_size = 8)(
  output reg   [word_size -1: 0]  count,
  input        [word_size -1: 0]  data_in,
  input                           Load_PC, Inc_PC,
  input                           clk, rst
);

  always @ (posedge clk, negedge rst)
    if (rst == 1'b0) count <= 0;
     else if (Load_PC == 1'b1) count <= data_in;
       else if (Inc_PC == 1'b1) count <= count +1;
endmodule

module Multiplexer_5ch #(parameter word_size = 8)(
  output  [word_size -1: 0]         mux_out,
  input   [word_size -1: 0]         data_a, data_b, data_c, data_d, data_e,
  input   [2: 0]                    sel
);
  assign mux_out = (sel == 0)       ? data_a: (sel == 1)
                                    ? data_b : (sel == 2)
                                    ? data_c: (sel == 3)
                                    ? data_d : (sel == 4)
                                    ? data_e : 'bx;
endmodule

module Multiplexer_3ch #(parameter word_size = 8)(
  output              [word_size -1: 0]  mux_out,
  input               [word_size -1: 0]  data_a, data_b, data_c,
  input               [1: 0]             sel
);
  assign mux_out = (sel == 0) ? data_a: (sel == 1) ? data_b : (sel == 2) ? data_c: 'bx;
endmodule
```

The ALU is modeled as combinational logic described by a level-sensitive cyclic behavior that is activated whenever the datapaths or the select bus changes. Parameters are used to make the description more readable and to reduce the likelihood of a coding error.

```
/*ALU Instruction     Action
ADD                   Adds the datapaths to form data_1 + data_2.
SUB                   Subtracts the datapaths to form data_1 - data_2.
AND                   Takes the bitwise-and of the datapaths, data_1 & data_2.
NOT                   Takes the bitwise Boolean complement of data_1.
*/
// Note: the carries are ignored in this model.
```

```
module Alu_RISC #(parameter word_size = 8,op_size = 4,
 // Opcodes
 NOP    = 4'b0000,
 ADD    = 4'b0001,
 SUB    = 4'b0010,
 AND    = 4'b0011,
 NOT    = 4'b0100,
 RD     = 4'b0101,
 WR     = 4'b0110,
 BR     = 4'b0111,
 BRZ    = 4'b1000
)(
 output reg  [word_size-1: 0]      alu_out,
 output                            alu_zero_flag,
 input      [word_size -1: 0]      data_1, data_2,
 input      [op_size -1: 0]        sel
);
 assign alu_zero_flag = ~|alu_out;
 always@ (sel,data_1,data_2)
  case (sel)
   NOP:            alu_out = 0;
   ADD:            alu_out = data_1 + data_2; // Reg_Y + Bus_1
   SUB:            alu_out = data_2 - data_1;
   AND:            alu_out = data_1 & data_2;
   NOT:            alu_out = ~ data_2;              // Gets data from Bus_1
   default:        alu_out = 0;
  endcase
endmodule
```

The control unit is rather large, but its design has a simple form, and its development follows directly from the ASM charts in Figure 7-12. First, declarations are made for the ports and variables needed to support the description. Then the datapath multiplexers are described with nested continuous assignments using the conditional (? . . . :) operator. Two cyclic behaviors are used: a level-sensitive behavior describes the combinational logic of the outputs and the next state, and an edge-sensitive behavior synchronizes the clock transitions.

```
module Control_Unit #(parameter
 word_size = 8, op_size = 4, state_size = 4,
 src_size = 2, dest_size = 2, Sel1_size = 3, Sel2_size = 2)(
 output [Sel2_size -1: 0] Sel_Bus_2_Mux,
 output [Sel1_size-1: 0] Sel_Bus_1_Mux,
 output reg Load_R0, Load_R1, Load_R2, Load_R3,Load_PC, Inc_PC,
            Load_IR, Load_Add_R, Load_Reg_Y, Load_Reg_Z, write,
 input [word_size -1: 0] instruction,
 input zero,clk, rst
);

// State Codes
 parameter S_idle = 0, S_fet1 = 1, S_fet2 = 2, S_dec = 3,
```

```
            S_ex1 = 4, S_rd1 = 5, S_rd2 = 6,
            S_wr1 = 7, S_wr2 = 8, S_br1 = 9, S_br2 = 10, S_halt = 11;
// Opcodes
  parameter NOP = 0, ADD = 1, SUB = 2, AND = 3, NOT = 4,RD = 5, WR = 6,
    BR = 7, BRZ = 8;
// Source and Destination Codes
  parameter R0 = 0, R1 = 1, R2 = 2, R3 = 3;

  reg [state_size-1: 0] state, next_state;
  reg Sel_ALU, Sel_Bus_1, Sel_Mem;
  reg Sel_R0, Sel_R1, Sel_R2, Sel_R3, Sel_PC;
  reg err_flag;
  wire [op_size -1: 0] opcode = instruction [word_size-1: word_size - op_size];
  wire [src_size -1: 0] src = instruction [src_size + dest_size -1: dest_size];
  wire [dest_size -1: 0] dest = instruction [dest_size -1: 0];

  // Mux selectors
  assign Sel_Bus_1_Mux[Sel1_size -1: 0] = Sel_R0 ? 0:
                                Sel_R1 ? 1:
                                Sel_R2 ? 2:
                                Sel_R3 ? 3:
                                Sel_PC ? 4: 3'bx; // 3-bits, sized number

  assign Sel_Bus_2_Mux[Sel2_size-1: 0] = Sel_ALU ? 0:
                                Sel_Bus_1 ? 1:
                                Sel_Mem ? 2: 2'bx;

  always @ (posedge clk, negedge rst) begin: State_transitions
    if (rst == 0) state <= S_idle; else state <= next_state; end

  /* always @ (state, instruction, zero) begin: Output_and_next_state
```

Note: The above sensitivity list leads to incorrect operation. The state transition causes the activity to be evaluated once, then the resulting instruction change causes it to be evaluated again, but with the residual value of *opcode*. On the second pass the value seen is the value *opcode* had before the state change, which results in *Sel_PC = 0* in state 3, which will cause a return to state 1 at the next clock. Finally, *opcode* is changed, but this does not trigger a re-evaluation because it is not in the event control expression. So, the caution is to be sure to use *opcode* in the sensitivity list. That way, the final execution of the behavior uses the value of *opcode* that results from the state change, and leads to the correct value of *Sel_PC*.

```
*/

    always @ (state, opcode, src, dest, zero) begin: Output_and_next-state
      Sel_R0 = 0;  Sel_R1 = 0;   Sel_R2 = 0;          Sel_R3 = 0;    Sel_PC = 0;
      Load_R0 = 0; Load_R1 = 0;  Load_R2 = 0;         Load_R3 = 0;   Load_PC = 0;
      Load_IR = 0; Load_Add_R = 0; Load_Reg_Y = 0; Load_Reg_Z = 0;
      Inc_PC = 0;
      Sel_Bus_1 = 0;
      Sel_ALU = 0;
      Sel_Mem = 0;
      write = 0;
      err_flag = 0;        // Used for de-bug in simulation
```

```
next_state = state;
 case (state)  S_idle:      next_state = S_fet1;
               S_fet1:      begin
                              next_state = S_fet2;
                              Sel_PC = 1;
                              Sel_Bus_1 = 1;
                              Load_Add_R = 1;
                             end
               S_fet2:      begin
                              next_state = S_dec;
                              Sel_Mem = 1;
                              Load_IR = 1;
                              Inc_PC = 1;
                             end
               S_dec:       case (opcode)
                            NOP: next_state = S_fet1;
                            ADD, SUB, AND: begin
                             next_state = S_ex1;
                             Sel_Bus_1 = 1;
                             Load_Reg_Y = 1;
                             case (src)
                               R0:              Sel_R0 = 1;
                               R1:              Sel_R1 = 1;
                               R2:              Sel_R2 = 1;
                               R3:              Sel_R3 = 1;
                               default          err_flag = 1;
                             endcase
                            end // ADD, SUB, AND
                            NOT: begin
                             next_state = S_fet1;
                             Load_Reg_Z = 1;
                             Sel_ALU = 1;
                             case (src)
                               R0:              Sel_R0 = 1;
                               R1:              Sel_R1 = 1;
                               R2:              Sel_R2 = 1;
                               R3:              Sel_R3 = 1;
                               default          err_flag = 1;
                             endcase
                            case (dest)
                               R0:              Load_R0 = 1;
                               R1:              Load_R1 = 1;
                               R2:              Load_R2 = 1;
                               R3:              Load_R3 = 1;
                               default          err_flag = 1;
                             endcase
                            end// NOT
                            RD: begin
                              next_state = S_rd1;
                              Sel_PC = 1; Sel_Bus_1 = 1; Load_Add_R = 1;
```

```
                         end // RD
                         WR: begin
                           next_state = S_wr1;
                           Sel_PC = 1; Sel_Bus_1 = 1; Load_Add_R = 1;
                         end // WR
                         BR: begin
                           next_state = S_br1;
                           Sel_PC = 1; Sel_Bus_1 = 1; Load_Add_R = 1;
                         end // BR
                         BRZ: if (zero == 1) begin
                           next_state = S_br1;
                           Sel_PC = 1; Sel_Bus_1 = 1; Load_Add_R = 1;
                         end // BRZ
                         else begin
                           next_state = S_fet1;
                           Inc_PC = 1;
                         end
                         default: next_state = S_halt;
                         endcase // (opcode)
      S_ex1:             begin
                           next_state = S_fet1;
                           Load_Reg_Z = 1;
                           Sel_ALU = 1;
                           case (dest)
                             R0: begin Sel_R0 = 1; Load_R0 = 1; end
                             R1: begin Sel_R1 = 1; Load_R1 = 1; end
                             R2: begin Sel_R2 = 1; Load_R2 = 1; end
                             R3: begin Sel_R3 = 1; Load_R3 = 1; end
                             default: err_flag = 1;
                           endcase
                         end
      S_rd1:             begin
                           next_state = S_rd2;
                           Sel_Mem = 1;
                           Load_Add_R = 1;
                           Inc_PC = 1;
                         end
      S_wr1:             begin
                           next_state = S_wr2;
                           Sel_Mem = 1;
                           Load_Add_R = 1;
                           Inc_PC = 1;
                         end
      S_rd2:             begin
                           next_state = S_fet1;
                           Sel_Mem = 1;
                           case (dest)
                             R0:          Load_R0 = 1;
                             R1:          Load_R1 = 1;
                             R2:          Load_R2 = 1;
```

```
                                R3:             Load_R3 = 1;
                                default         err_flag = 1;
                              endcase
                            end
            S_wr2:          begin
                              next_state = S_fet1;
                              write = 1;
                              case (src)
                                R0:             Sel_R0 = 1;
                                R1:             Sel_R1 = 1;
                                R2:             Sel_R2 = 1;
                                R3:             Sel_R3 = 1;
                                default         err_flag = 1;
                              endcase
                            end
            S_br1:          begin next_state = S_br2; Sel_Mem = 1;
                             Load_Add_R = 1; end
            S_br2:          begin next_state = S_fet1; Sel_Mem = 1;
                             Load_PC = 1; end
            S_halt:         next_state = S_halt;
            default:        next_state = S_idle;
    endcase
  end
endmodule
```

For simplicity, the memory unit of the machine is modeled as an array of D-type flip-flops. An alternative would be to use an external static RAM.

```
module Memory_Unit #(parameter word_size = 8, memory_size = 256)(
 output [word_size -1: 0] data_out,
 input [word_size -1: 0] data_in,
 input [word_size -1: 0] address,
 input clk, write
);
 reg [word_size -1: 0] memory [memory_size-1: 0];

 assign data_out = memory[address];
 always @ (posedge clk)
 if (write) memory[address] <= data_in;
endmodule
```

7.3.6 RISC SPM: Program Execution

A testbench for verifying that *RISC_SPM* executes a stored program[10] is given in page 374. *test_RISC_SPM* defines probes to display individual words in memory, uses a one-shot (***initial***) behavior to flush memory, and loads a small program and data into separate

[10]An assembler for the machine is located at the website for this book, and can be used to generate programs for use in embedded applications of the processor.

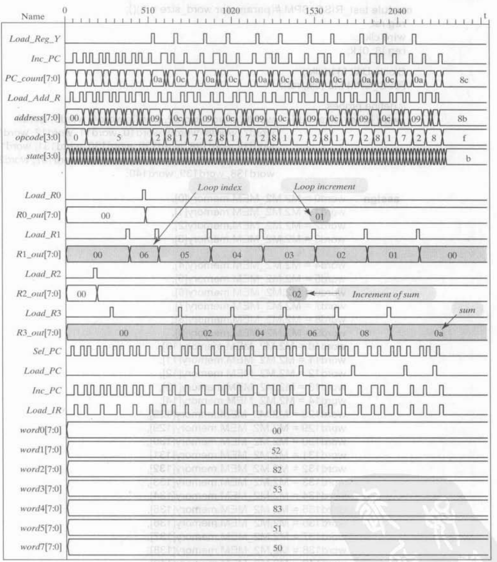


FIGURE 7-13 Simulation results produced by executing a stored program with *RISC_SPM*.

areas of memory. The program (1) reads memory and loads the data into the registers of the processor, (2) executes subtraction to decrement a loop counter, (3) adds register contents while executing the loop, and (4) branches to a halt when the loop index is 0. This sequence of instructions demonstrates the machine's ability to execute a "for" loop. The results of executing the program are displayed in Figure 7-13.

```
module test_RISC_SPM #(parameter word_size = 8)();
 reg rst;
 wire clk;
 reg [8: 0] k;

 Clock_Unit M1 (clk);
 RISC_SPM M2 (clk, rst);

 // define probes
  wire [word_size-1: 0]    word0, word1, word2, word3, word4, word5, word6,
                           word7, word8, word9, word10, word11, word12, word13,
                           word14,word128, word129, word130, word131, word132,
                           word133, word134, word135, word136, word137, word255,
                           word138, word139, word140;

  assign    word0 = M2.M2_MEM.memory[0],
            word1 = M2.M2_MEM.memory[1],
            word2 = M2.M2_MEM.memory[2],
            word3 = M2.M2_MEM.memory[3],

            word4 = M2.M2_MEM.memory[4],
            word5 = M2.M2_MEM.memory[5],
            word6 = M2.M2_MEM.memory[6],
            word7 = M2.M2_MEM.memory[7],
            word8 = M2.M2_MEM.memory[8],
            word9 = M2.M2_MEM.memory[9],
            word10 = M2.M2_MEM.memory[10],
            word11 = M2.M2_MEM.memory[11],
            word12 = M2.M2_MEM.memory[12],
            word13 = M2.M2_MEM.memory[13],
            word14 = M2.M2_MEM.memory[14],
            word128 = M2.M2_MEM.memory[128],
            word129 = M2.M2_MEM.memory[129],
            word130 = M2.M2_MEM.memory[130],
            word131 = M2.M2_MEM.memory[131],
            word132 = M2.M2_MEM.memory[132],
            word133 = M2.M2_MEM.memory[133],
            word134 = M2.M2_MEM.memory[134],
            word135 = M2.M2_MEM.memory[135],
            word136 = M2.M2_MEM.memory[136],
            word137 = M2.M2_MEM.memory[137],
            word138 = M2.M2_MEM.memory[138],
            word140 = M2.M2_MEM.memory[140],
            word255 = M2.M2_MEM.memory[255];
  initial #2800 $finish;

 // Flush Memory
  initial begin: Flush_Memory
  #2 rst = 0; for (k=0; k<=255; k=k+1) M2.M2_MEM.memory[k] = 0; #10 rst = 1;
  end
```

```
initial begin: Load_program
#5
                                                // opcode_src_dest
  M2.M2_MEM.memory[0] = 8'b0000_00_00;          // NOP
  M2.M2_MEM.memory[1] = 8'b0101_00_10;          // Read 130 to R2
  M2.M2_MEM.memory[2] = 130;
  M2.M2_MEM.memory[3] = 8'b0101_00_11;          // Read 131 to R3
  M2.M2_MEM.memory[4] = 131;
  M2.M2_MEM.memory[5] = 8'b0101_00_01;          // Read 128 to R1
  M2.M2_MEM.memory[6] = 128;
  M2.M2_MEM.memory[7] = 8'b0101_00_00;          // Read 129 to R0
  M2.M2_MEM.memory[8] = 129;
  M2.M2_MEM.memory[9] = 8'b0010_00_01;          // Sub R1-R0 to R1
  M2.M2_MEM.memory[10] = 8'b1000_00_00;         // BRZ
  M2.M2_MEM.memory[11] = 134;                   // Holds address for BRZ
  M2.M2_MEM.memory[12] = 8'b0001_10_11;         // Add R2+R3 to R3
  M2.M2_MEM.memory[13] = 8'b0111_00_11;         // BR
  M2.M2_MEM.memory[14] = 140;
  // Load data
  M2.M2_MEM.memory[128] = 6;
  M2.M2_MEM.memory[129] = 1;
  M2.M2_MEM.memory[130] = 2;
  M2.M2_MEM.memory[131] = 0;
  M2.M2_MEM.memory[134] = 139;
  M2.M2_MEM.memory[139] = 8'b1111_00_00;        // HALT
  M2.M2_MEM.memory[140] = 9;                    // Recycle
 end
endmodule

module Clock_Unit (output reg clock);
 parameter delay = 0;
 parameter half_cycle = 10;
 initial begin #delay clock = 0; forever #half_cycle clock = ~clock; end
endmodule
```

7.4 Design Example: UART

Systems that exchange information and interact remotely via serial data channels use serializer/deserializer (SerDes) interfaces for conversion between serial and parallel formats of data.[11] There are many different architectures, coding schemes, and clocking schemes for such circuits. For simplicity, we will consider here a simple modem, which acts as an interface between a host machines/device and the serial data channel, as shown in Figure 7-14. A modem allows a computer to connect to a telephone line and communicate with a receiving computer [2, 5]. The host machine stores information in a parallel word format, but transmits and receives data in a serial, single-bit, format.

[11] See, for example, reference material at National Semiconductor (www.national.com), or Google SerDes.

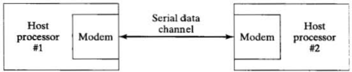


FIGURE 7-14 Processor/modem communication over a serial channel.

A modem is also called a *universal asynchronous receiver and transmitter* (UART), indicating that the device has the capability to both receive and transmit serial data, and that the sending and receiving units are not synchronized to each other. This design example will address the basic modeling and synthesis of a UART's transmitter and receiver.

For this discussion, a UART exchanges text data in an American Standard Code for Information Interchange (ASCII) format in which each alphabetical character is encoded by 7 bits and augmented by a parity bit that can be used for error detection. For transmission, the modem wraps this 8-bit sub-word with a *start-bit* in the least significant bit (LSB), and a *stop-bit* in the most significant bit (MSB), resulting in the 10-bit word format shown in Figure 7-15. The first 9 data bits of the word are transmitted in sequence, beginning with the start-bit, with each bit being asserted at the serial line for one cycle (bit-time) of the modem clock. The stop-bit may assert for more than one clock.

7.4.1 UART Operation

The UART transmitter is always part of a larger environment in which a host processor controls transmission by fetching a data word in parallel format and directing the UART to transmit it in a serial format. The receiver must detect transmission, receive the data in serial format, strip off the start- and stop-bits, and store the data word in a parallel format. The receiver's job is more complex, because the clock used to send the inbound data is not available at the remote receiver—data arrives at the receiver asynchronously. The receiver must regenerate the clock locally to synchronize the sampling of inbound data using the receiving machine's clock rather than the clock of the transmitting machine.

The simplified architecture of a UART presented in Figure 7-16 shows the signals used by a host processor to control the UART and to move data to and from a data bus in the host machine. Details of the host machine are not shown.

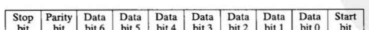


FIGURE 7-15 Data format for ASCII text transmitted by a UART.

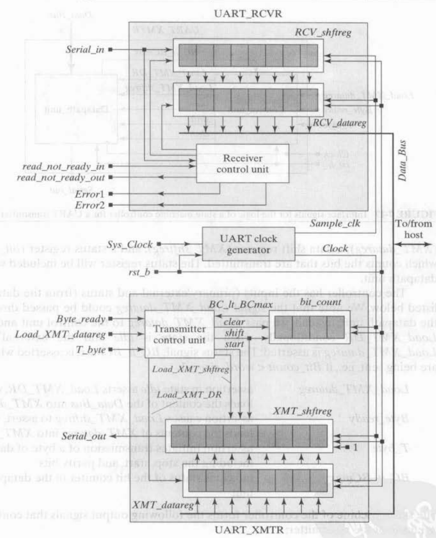


FIGURE 7-16 Block diagram of a UART (transmitter and receiver).

7.4.2 UART Transmitter

The input–output signals of the transmitter are shown in the high-level block diagram in Figure 7-17. The input signals are provided by the host processor, and the output signal is the serial data stream, a status signal (*read_not_ready_out*), and two error signals. The architecture of the transmitter consists of a control unit, a data register

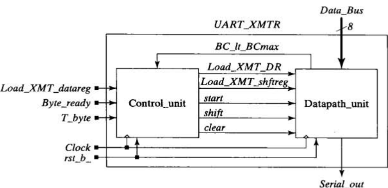


FIGURE 7-17 Interface signals for the logic of a state machine controller for a UART transmitter.

(*XMT_datareg*), a data shift register (*XMT_shftreg*), and a status register (*bit_count*), which counts the bits that are transmitted. The status register will be included with the datapath unit.

The controller has the inputs (primary/external and status (from the datapath)) listed below. We note that the signal *Load_XMT_datareg* could be passed directly to the datapath unit; instead, we pass *Load_XMT_datareg* to the control unit and assert *Load_XMT_DR* conditionally when the state is *idle* and the external signal *Load_XMT_datareg* is asserted. The status signal, *BC_lt_BCmax*, is asserted while bits are being sent, i.e., if *Bit_count* < *word_size* _ 1.

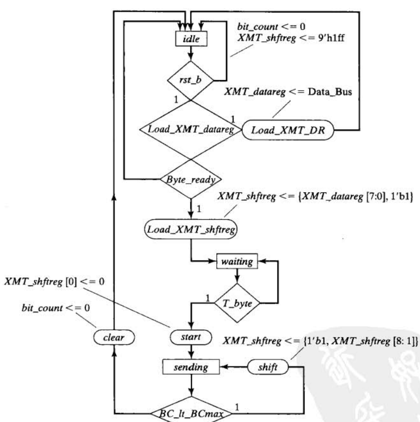
Load_XMT_datareg	assertion in state *idle* asserts *Load_XMT_DR*, which loads the content of the *Data_Bus* into *XMT_data_reg*
Byte_ready	assertion causes *Load_XMT_shftreg* to assert, which loads the contents of *XMT_datareg* into *XMT_shftreg*
T_byte	assertion initiates transmission of a byte of data, including the stop, start, and parity bits
BC_lt_BCmax	indicates status of the bit counter in the datapath unit

The state machine of the controller forms the following output signals that control the datapath of the transmitter:

Load_XMT_DR	assertion loads *Data_Bus* into *XMT_datareg*
Load_XMT_shftreg	assertion loads the contents of *XMT_data_reg* into *XMT_shftreg*
start	signals the start of transmission by dropping *XMT_shftreg*[0] to 0
shift	directs *XMT_shftreg* to shift by one bit towards the LSB and to backfill with a stop bit (1).
clear	clears *bit_count*

The ASMD chart of the state machine controlling the transmitter is shown in Figure 7.18. The machine has three states: *idle, waiting*, and *sending*. When the active-low, synchronous reset signal *rst_b* is asserted, the machine enters *idle, bit_count* is flushed, and *XMT_shftreg* is loaded with 1s. In *idle*, if an active edge of *Clock* occurs while *Load_XMT_data_reg* is asserted by the external host, the output signal *Load_XMT_DR* will load *XMT_data_reg* with the contents of *Data_Bus*. The machine remains in *idle* until *start* is asserted to drop *XMT_shftreg*[0].

When *Byte_ready* is asserted (with *rst_b* and *Load_XMT_datareg* de-asserted), *Load_XMT_shftreg* is asserted and *next_state* is driven to *waiting*. The assertion of



Note: Only the branch corresponding to a true decision is annotated at a decision diamond; signals that are de-asserted are not shown explicitly de-asserted. Conditional assertions are indicated by the name of the asserted signal.
Note: *BC_lt_BCmax* asserts if *Bit_count* < *word_size* + 1.

FIGURE 7-18 ASMD chart for the state machine controller of the UART transmitter.

Load_XMT_shftreg indicates that *XMT_datareg* now contains data that can be transferred to the internal shift register. At the next active edge of *Clock*, with *Load_XMT_shftreg* asserted, three activities occur: (1) *state* transfers from *idle* to *waiting*, (2) the contents of *XMT_datareg* are loaded into the leftmost bits of *XMT_shftreg*, a (*word_size* + 1)-bit shift register whose LSB signals the start and stop of transmission, and (3) the LSB of *XMT_shftreg* is reloaded with 1, the stop-bit. The machine remains in *waiting* until the external host processor asserts *T_byte*.

At the next active edge of *Clock*, with *T_byte* asserted, *state* enters *sending*, and the LSB of *XMT_shftreg* is set to 0 to signal the start of transmission. At the same time, *shift* is driven to 1, and *next_state* retains the state code corresponding to *sending*. At subsequent active edges of *Clock*, with *shift* asserted, *state* remains in *sending* and the contents of *XMT_shftreg* are shifted toward the LSB, which drives the external serial channel. As the data shifts occur, 1s are back-filled in *XMT_shftreg*, and *bit_count* is incremented. With *state* in *sending*, *shift* asserts while *bit_count* is less than 9, i.e., while *BC_lt_BCmax* is asserted. The machine increments *bit_count* after each movement of data, and when *bit_count* reaches 9 (*BC_lt_BCmax* is 0) *clear* asserts, indicating that all of the bits of the augmented word have been shifted to the serial output. At the next active edge of *Clock*, the machine returns to *idle*.

The control signals produced by the state machine induce state-dependent register transfers in the datapath. The activity of *Load_XMT_DR* (conditioned on the primary input *Load_XMT_datareg*), the other primary inputs (*Byte_ready*, and *T_byte*), and the other signals from the controller (*Load_XMT_shftreg, start, shift, clear*) are shown in Figure 7-19 with *bit_count*, along with the movement of data in the datapath registers. The contents of the registers are shown at successive edges of clock, with a time axis going from the top of the figure toward the bottom. Transitions of the active edge of *clock* occur between the successive rows displaying contents of *XMT_datareg*. The bits of the transmitted signal are shown in the sequence in which they are transmitted, with the rightmost cell of *XMT_shftreg* holding the bit that is transmitted at the serial interface at each step. The state of the machine is shown, and the state transitions and datapath register transitions that occur on the rising edges of *clock* are shown in the register boxes. The values of the control signals that cause the register transitions are also shown. The displayed values of the control signals are those they held immediately before the active edge of *Clock* and which cause the register transfers that are shown. The sequence of output bits is also shown, with 1s being pushed into the MSB of *XMT_shftreg* under the action of *shift*. The sequence of output bits of the transmitted signal are shown as a word at each time step, with the understanding that *the LSB of the word is the first bit that was transmitted*, and the MSB of the word is the most recent bit that was transmitted at the serial interface.

The Verilog description, *UART_XMTR*, of the partitioned machine has three cyclic behaviors—a level-sensitive behavior describing the combinational logic for next state and outputs of the controller, an edge-sensitive behavior to synchronize the

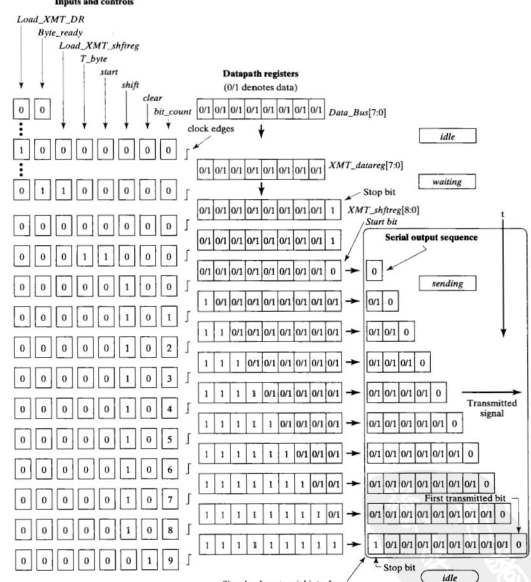


FIGURE 7-19 Control signals and dataflow in an 8-bit UART transmitter.

state transitions of the controller, and another edge-sensitive behavior to synchronize the register transfers of the datapath registers.

```
module UART_XMTR #(parameter word_size = 8)(  // Size of data, e.g., 8 bits
 output                Serial_out,            // Serial output to data channel
 input [word_size - 1 : 0] Data_Bus,          // Host data bus containing data
                                              //   word
 input                 Load_XMT_datareg,      // Used by host to load the data
                                              //   register
                       Byte_ready,            // Used by host to signal ready
                       T_byte,                // Used by host to signal start of
                                              //   transmission
                       Clock,                 // Bit clock of the transmitter
                       rst_b                  // Resets internal registers, loads
                                              //   the
);                                            // XMT_shftreg with ones,
 Control_Unit M0 (
  Load_XMT_DR, Load_XMT_shftreg, start, shift, clear, Load_XMT_datareg,
   Byte_ready, T_byte, BC_lt_BCmax, Clock, rst_b
 );
 Datapath_Unit M1 (
  Serial_out, BC_lt_BCmax, Data_Bus, Load_XMT_DR, Load_XMT_shftreg, start, shift,
  clear, Clock, rst_b
 );
 endmodule

module Control_Unit #(
  parameter  one_hot_count = 3,              // Number of one-hot states
             state_count = one_hot_count,    // Number of bits in state register
             size_bit_count = 3,             // Size of the bit counter, e.g., 4
                                             // Must count to word_size + 1
             idle = 3'b001,                  // one-hot state encoding
             waiting = 3'b010,
             sending = 3'b100,
             all_ones = 9'b1_1111_1111       // Word + 1 extra bit
 )(
  output reg       Load_XMT_DR,              // Loads Data_Bus into
                                             //   XMT_datareg
  output reg       Load_XMT_shftreg,         // Loads XMT_datareg into
                                             //   XMT_shftreg
  output reg       start,                    // Launches shifting of bits in
                                             //   XMT_shftreg
  output reg       shift,                    // Shifts bits in XMT_shftreg
  output reg       clear,                    // Cears bit_count after last bit is
                                             //   sent
  input            Load_XMT_datareg,         // Asserts Load_XMT_DR in
                                             //   state idle
  input            Byte_ready,               // Asserts Load_XMT_shftreg in
                                             //   state idle
```

```
  input               T_byte,           // asserts start signal in state
                                           waiting
  input               BC_lt_BCmax,      // Indicates status of bit counter
  input               Clock,
  input               rst_b
);
  reg [state_count -1: 0]  state, next_state;   // State machine controller
  always @ (state, Load_XMT_datareg, Byte_ready, T_byte, BC_lt_BCmax) begin:
  Output_and_next_state
    Load_XMT_DR = 0;
    Load_XMT_shftreg = 0;
    start = 0;
    shift = 0;
    clear = 0;
    next_state = idle;
    case (state)
     idle:          if (Load_XMT_datareg == 1'b1) begin
                     Load_XMT_DR = 1;
                     next_state = idle;
                    end
                    else if (Byte_ready == 1'b1) begin
                     Load_XMT_shftreg = 1;
                     next_state = waiting;
                    end
    waiting:        if (T_byte == 1) begin
                     start = 1;
                     next_state = sending;
                    end else next_state = waiting;
    sending:        if (BC_lt_BCmax) begin
                     shift = 1;
                     next_state = sending;
                    end
                    else begin
                     clear = 1;
                     next_state = idle;
                    end
    default:        next_state = idle;
   endcase
 end
 always @ (posedge Clock, negedge rst_b) begin: State_Transitions
   if (rst_b == 1'b0) state <= idle; else state <= next_state; end
 endmodule
 module Datapath_Unit #(
  parameter                word_size = 8,
                           size_bit_count = 3,
                           all_ones = {(word_size +1){1'b1}}// 9 bits of ones
 )(
  output                   Serial_out,
                           BC_lt_BCmax,
```

```
    input [word_size -1: 0]   Data_Bus,
    input                     Load_XMT_DR,
    input                     Load_XMT_shftreg,
    input                     start,
    input                     shift,
    input                     clear,
    input                     Clock,
    input                     rst_b
  );
    reg [word_size -1: 0]     XMT_datareg;        // Transmit Data Register
    reg [word_size: 0]        XMT_shftreg;        // Transmit Shift Register:
                                                     {data, start bit}
    reg [size_bit_count: 0]   bit_count;          // Counts the bits that are
                                                     transmitted

    assign Serial_out = XMT_shftreg[0];
    assign BC_lt_BCmax = (bit_count < word_size + 1);

    always @ (posedge Clock, negedge rst_b)
      if (rst_b == 0) begin
        XMT_shftreg <= all_ones;
        bit_count <= 0;
      end
      else begin: Register_Transfers
        if (Load_XMT_DR == 1'b1) XMT_datareg <= Data_Bus;   // Get the data bus

        if (Load_XMT_shftreg == 1'b1)
          XMT_shftreg <= {XMT_datareg,1'b1};                // Load shift reg,
                                                             // insert stop bit

        if (start == 1'b1) XMT_shftreg[0] <= 0;             // Signal start of
                                                                transmission

        if (clear == 1'b1) bit_count <= 0;

        if (shift == 1'b1) begin
          XMT_shftreg <= {1'b1, XMT_shftreg[word_size:1]};  // Shift right, fill with
                                                                1's

          bit_count <= bit_count + 1;
       end
    end   // Register_Transfers
  endmodule
```

Some simulation results are shown in Figures 7-20 and 7-21 for an 8-bit data word. The waveforms produced by the simulator have been annotated to indicate significant features of the transmitter's behavior. First, observe the values of the signals immediately after *rst_b* is asserted. The state is *idle*. Note that *Data_Bus* initially contains the value $a7_h$ (1010_0111_2), a value specified by the testbench used for simulation. With *Byte_ready* not yet asserted, and with *Load_XMT_datareg* asserted, the *Data_Bus* is loaded into *XMT_datareg*. The machine remains in *idle* until *Byte_ready* is asserted. When *Byte_ready* asserts, then *Load_XMT_shftreg* asserts. This causes the state to change to *waiting* at the next active edge of *Clock*. The 9-bit *XMT_shftreg* is now loaded

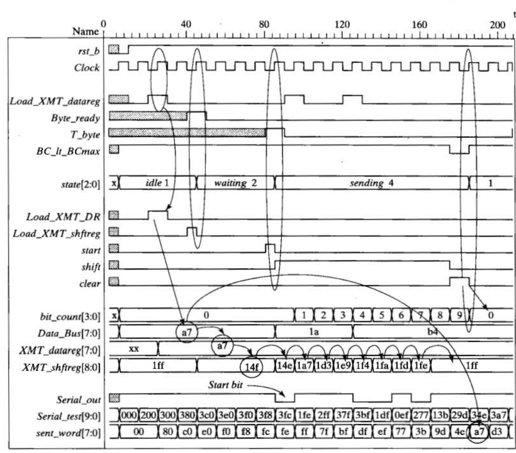


FIGURE 7-20 Annotated simulation results for the 8-bit UART transmitter.

with the value $\{a7_h, 1\} = 1_0100_1111_2 = 14f_h$. Note that the LSB of *XMT_shftreg* is loaded with a 1. The machine remains in *waiting* until *T_byte* is asserted. The assertion of *T_byte* asserts *start*. The machine enters *sending* at the active edge of *Clock* immediately after the host processor's assertion of *T_byte*, and the LSB of *XMT_shftreg* is loaded with a 0. The 9-bit word in *XMT_shftreg* becomes $1_0100_1110_2 = 14e_h$. The 0 in the LSB signals the start of transmission. Figure 7-21 has been annotated to show the movement of data through *XMT_shftreg*. Note that 1s are filled behind, as the word shifts to the right. At the active edge of the clock after *bit_count* reaches 9 (for an 8-bit word), *clear* asserts, *bit_count* is flushed, and the machine returns to *idle*.

For diagnostic purposes, the testbench includes a 10-bit shift register that receives *Serial_out* (by hierarchical dereferencing). The eight innermost bits of this register are displayed in Figure 7-20 as *sent_word*[7:0]. Note that because *sent_word* is a registered output (loaded into a shift register) it has the value $a7_h$ in the clock cycle (bit time)

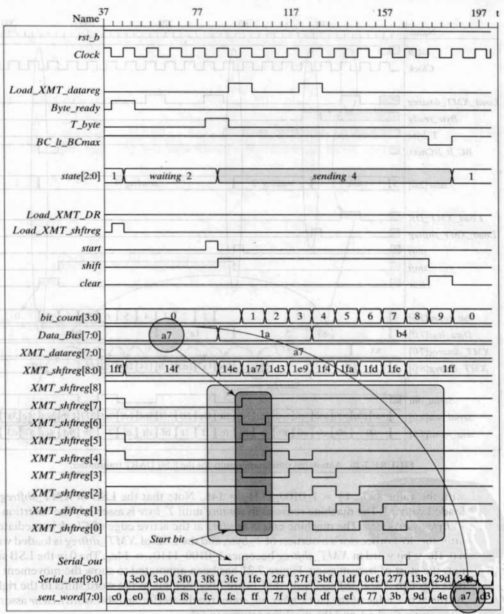


FIGURE 7-21 Data movement through *XMT_shftreg*.

after *bit_count* reaches 9. The results in Figures 7-20 and 7-21 demonstrate that (1) the content of the data bus was loaded into the XMT data register under the control of *Load_XMT_DR*, (2) the 8-bit content of the XMT data register was loaded into the most significant 8 bits of the 9-bit XMT shift register under the control of *Load_XMT_shftreg*, and (3) the start bit dropped and the contents of the XMT shift register were shifted out under the control of *start*.

The circuit synthesized from *UART_XMTR* is shown in Figures 7-22 and 7-23. The Verilog model synthesizes as a unit, but for illustration and discussion, the description was partitioned and synthesized in two parts: one for the register transfers of the datapath unit and another for the combinational logic forming the next state and the control signals that govern the register transfers. The datapath unit consists of an 8-bit register holding *XMT_datareg*, a 9-bit shift register holding *XMT_shftreg*, and a bit counter. The circuit uses *dffrgpqb_a*, a D-type flip-flop with a rising-edge clock, asynchronous active-low reset, and internal gated data of the external data or the output, and *dffspb_a*, a D-type flip-flop with rising-edge clock, and asynchronous active-low set. The shift registers have been highlighted in Figure 7-23.

7.4.3 UART Receiver

The UART receiver receives the serial bit stream of data, removes the start-bit, and transfers the data in a parallel format to a storage register connected to the host data bus. The transmitter's clock is not available to the receiver, so, although the data arrives at a standard bit rate, the data is not necessarily synchronized with the internal clock at the host of the receiver. This issue of synchronization is resolved by generating a *local* clock at a higher frequency and using it to sample the received data in a manner that preserves the integrity of the data.[12] In the scheme used here, the data, assumed to be in a 10-bit format, will be sampled at a rate determined by *Sample_clock*, which is generated at the receiver's host. The cycles of *Sample_clock* will be counted to ensure that the data are sampled in the middle of a bit time, as shown in Figure 7-24.[13] The sampling algorithm must (1) verify that a start bit has been received, (2) generate samples from 8 bits of the data, and (3) load the sampled data onto the local bus.

Although a higher sampling frequency could be used, the frequency of *Sample_clock* in this example is 8 times the (known) frequency of the bit clock that transmitted the data. This ensures that a slight misalignment between the leading edge of a cycle of *Sample_clock* and the arrival of the start-bit will not compromise the sampling scheme, because the sample will still be taken within the interval of time corresponding to a transmitted bit. The arrival of a start-bit will be determined by the detection of successive samples of value 0 after the serial input data goes low. Then three additional samples will be taken to confirm that a valid start-bit has arrived. Thereafter, 8 successive bits will be sampled at approximately the center of their bit times. Under worst-case conditions of misalignment, the sample is taken a full cycle of *Sample_clock* ahead of the actual center of the bit time, which is a tolerable skew. The

[12] Typically, a phase-lock-loop is used to implement a local clock. This circuit is outside the scope of synthesis tools.

[13] We assume that the inbound data has been synchronized to the local clock of the receiver.

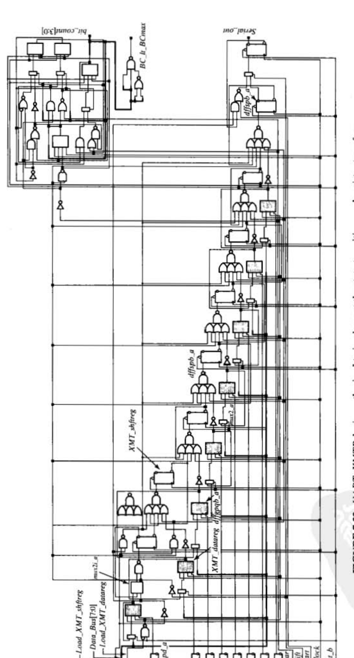


FIGURE 7-22 *UART_XMTR*: logic synthesized to implement the state transitions and register transfers.

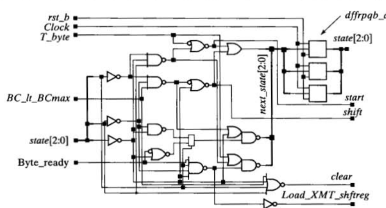


FIGURE 7-23 *UART_XMTR*: combinational logic forming the next state and control signals for the register transfer.

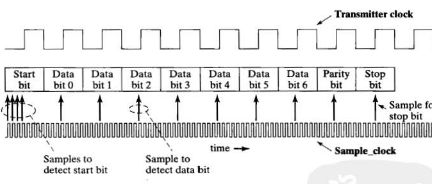


FIGURE 7-24 UART receiver sampling format for clock regeneration.

datapath unit holds the counters which implement this scheme, and its status signals are reported to the control unit.

The high-level block diagram in Figure 7-25 shows the input–output signals of a state-machine controller that will interface with the host processor and direct the receiver's sampling scheme.

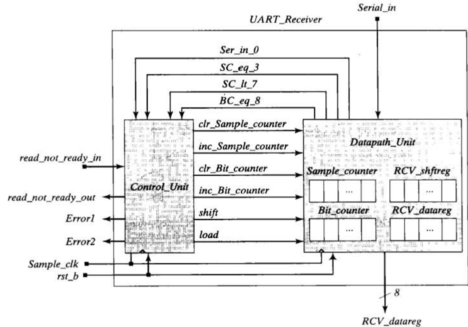


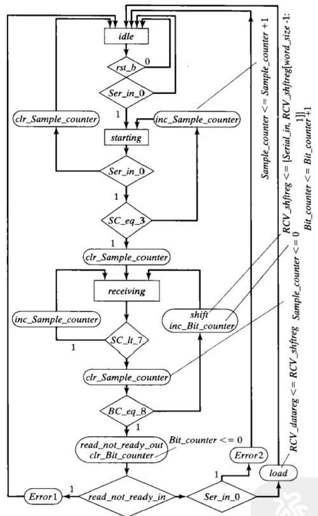
FIGURE 7-25 Block diagram of *UART_receiver*, including the interface signals between the control unit and the datapath unit.

The state machine has the following primary (external) inputs and status inputs:

read_not_ready_in	signals that the host is not ready to receive data
Ser_in_0	asserts while *Serial_in* is 0
SC_eq_3	asserts while *Sample_counter* = 3
SC_lt_7	asserts while *Sample_counter* < 7
BC_eq_8	asserts while *Bit_counter* = 8
Sample_counter	counts the samples of a bit
Bit_counter	counts the bits that have been sampled

The state machine produces the following outputs:

read_not_ready_out	signals that the receiver has received 8 bits
clr_Sample_counter	clears *Sample_counter*
inc_Sample_counter	increments *Sample_counter*
clr_Bit_counter	clears *Bit_counter*
inc_Bit_counter	increments *Bit_counter*
shift	causes *RCV_shftreg* to shift towards the LSB
load	causes *RCV_shftreg* to transfer data to *RCV_datareg*
Error1	asserts if host is not ready to receive data after last bit has been sampled
Error2	asserts if the stop-bit is missing



Note: Ser_in_0 asserts if *Serial_in is* 0
SC_eq_3 asserts if *Sample_counter* = 3
SC_lt_7 asserts while *Sample_counter* < 7
BC_eq_8 asserts if *Bit_counter* = 8

FIGURE 7-26 ASMD chart for *UART_receiver*.

The ASMD chart of a state machine controller for the receiver is shown in Figure 7-26. The machine has three states: *idle, starting*, and *receiving*. Transitions between states are synchronized by *Sample_clk*. Assertion of a synchronous active-low reset puts the machine in the *idle* state. It remains there until the status signal *Ser_in_0* is low and then makes a transition to *starting*. In *starting*, the machine

samples *Serial_in* repeatedly to determine whether the first bit is a valid start-bit (it must be 0). Depending on the sampled values, *inc_Sample_counter* and *clr_Sample_counter* may be asserted to increment or clear the counter at the next active edge of *Sample_clock*. If the next three samples of *Serial_in* are 0, the machine treats the bit as a valid start-bit and goes to the state *receiving*. *Sample_counter* is cleared on the transition to *receiving*. In this state, eight successive samples are taken (one for each bit of the byte, at each active edge of *Sample_clk*), with *inc_Sample_counter* asserted. Then *Bit_counter* is incremented. If the sampled bit is not the last (parity) bit, *inc_Bit_counter* and *shift* are asserted. The assertion of *shift* will cause the sample value to be loaded into the MSB of *RCV_shftreg*, the receiver shift register, and will shift the 7 leftmost bits of the register toward the LSB.

After the last bit has been sampled, the machine will assert *read_not_ready_out*, a handshake output signal to the processor and clear the bit counter. At this time, the machine also checks the integrity of the data and the status of the host processor. If *read_not_ready_in* is asserted, the host processor is not ready to receive the data (*Error1*). If a stop-bit is not the next bit (detected by *Ser_in_0* = 1)), there is an error in the format of the received data (*Error2*). Otherwise, *load* is asserted to cause the contents of the shift register to be transferred as a parallel word to *RCV_datareg*, a data register in the host machine, with a direct connection to *data_bus*.

The Verilog description of the 8-bit UART receiver is given below. The description follows directly from the ASMD chart in Figure 7-26. Note that the ports of the parent modules of a partition must be sized properly to accommodate vector ports in the child modules. Otherwise, the ports will be treated as default scalars in the scope of the parent module.

```
module UART_RCVR #(parameter word_size = 8, half_word = word_size /2)(
  output [word_size -1: 0]  RCV_datareg,
  output                    read_not_ready_out,
                            Error1, Error2,
  input                     Serial_in,
                            read_not_ready_in,
                            Sample_clk,
                            rst_b
  );
   Control_Unit M0 (
    read_not_ready_out,
    Error1, Error2,
    clr_Sample_counter,
    inc_Sample_counter,
    clr_Bit_counter,
    inc_Bit_counter,
    shift,
    load,
    read_not_ready_in,
    Ser_in_0,
    SC_eq_3,
    SC_lt_7,

    BC_eq_8,
    Sample_clk,
```

```
    rst_b
  );
  Datapath_Unit M1 (
    RCV_datareg,
    Ser_in_0,
    SC_eq_3,
    SC_lt_7,
    BC_eq_8,
    Serial_in,
    clr_Sample_counter,
    inc_Sample_counter,
    clr_Bit_counter,
    inc_Bit_counter,
    shift,
    load,
    Sample_clk,
    rst_b
  );
endmodule
module Control_Unit #(parameter
  word_size = 8, half_word = word_size /2, Num_state_bits = 2,
  idle = 2'b00, starting = 2'b01, receiving = 2'b10        // one-hot assignment
)(
  output reg        read_not_ready_out,
                    Error1, Error2,
                    clr_Sample_counter,
                    inc_Sample_counter,
                    clr_Bit_counter,
                    inc_Bit_counter,
                    shift,
                    load,
  input             read_not_ready_in,
                    Ser_in_0,
                    SC_eq_3,
                    SC_lt_7,
                    BC_eq_8,
                    Sample_clk,
                    rst_b
);
  reg               [word_size-1: 0]          RCV_shftreg;
  reg               [Num_state_bits -1: 0]    state, next_state;
  always @ (posedge Sample_clk)
    if (rst_b == 1'b0) state <= idle; else state <= next_state;
  always @ (state, Ser_in_0, SC_eq_3, SC_lt_7, read_not_ready_in) begin
    read_not_ready_out = 0;
    clr_Sample_counter = 0;
    clr_Bit_counter = 0;
    inc_Sample_counter = 0;
    inc_Bit_counter = 0;
    shift = 0;
```

```
Error1 = 0;
Error2 = 0;
load = 0;
next_state = idle;
case (state)
 idle: if (Ser_in_0 == 1'b1) next_state = starting;
               else next_state = idle;
 starting:     if (Ser_in_0 == 1'b0) begin
                next_state = idle;
                clr_Sample_counter = 1;
               end else
               if (SC_eq_3 == 1'b1) begin
                next_state = receiving;
                clr_Sample_counter = 1;
               end else begin inc_Sample_counter = 1; next_state = starting; end
 receiving:    if (SC_lt_7 == 1'b1) begin
                 inc_Sample_counter = 1;
                 next_state = receiving;
               end
               else begin
                clr_Sample_counter = 1;
                if (!BC_eq_8) begin
                 shift = 1;
                 inc_Bit_counter = 1;
                 next_state = receiving;
                end
                else begin
                 next_state = idle;
                 read_not_ready_out = 1;
                 clr_Bit_counter = 1;
                 if (read_not_ready_in == 1'b1) Error1 = 1;
                 else if (Ser_in_0 == 1'b1) Error2 = 1;
                 else load = 1;
                end
               end
   default:    next_state = idle;
  endcase
 end
endmodule

module Datapath_Unit #(parameter
 word_size = 8, half_word = word_size /2, Num_counter_bits = 4
)(
 output reg    [word_size-1: 0]         RCV_datareg,
 output                                 Ser_in_0,
                                        SC_eq_3,
                                        SC_lt_7,
                                        BC_eq_8,
 input                                  Serial_in,
```

```
                                        clr_Sample_counter,
                                        inc_Sample_counter,
                                        clr_Bit_counter,
                                        inc_Bit_counter,
                                        shift,
                                        load,
                                        Sample_clk,
                                        rst_b
);
  reg   [word_size-1: 0]                RCV_shftreg;
  reg   [Num_counter_bits -1: 0]        Sample_counter;
  reg   [Num_counter_bits: 0]           Bit_counter;
  assign Ser_in_0 = (Serial_in == 1'b0);
  assign BC_eq_8 = (Bit_counter == word_size);
  assign SC_lt_7 = (Sample_counter < word_size -1);
  assign SC_eq_3 = (Sample_counter == half_word -1);
  always @ (posedge Sample_clk)
  if (rst_b == 1'b0) begin              // synchronous rst_b
    Sample_counter <= 0;
    Bit_counter <= 0;
    RCV_datareg <= 0;
    RCV_shftreg <= 0;
  end
  else begin
    if (clr_Sample_counter == 1) Sample_counter <= 0;
    else if (inc_Sample_counter == 1) Sample_counter <= Sample_counter + 1;
    if (clr_Bit_counter == 1) Bit_counter <= 0;
    else if (inc_Bit_counter == 1) Bit_counter <= Bit_counter + 1;
    if (shift == 1) RCV_shftreg <= {Serial_in, RCV_shftreg[word_size-1:1]};
    if (load == 1) RCV_datareg <= RCV_shftreg;
  end
endmodule
```

The simulation results in Figure 7-27 are annotated to show functional features of the waveforms. The received data word is $b5_h = 1011_0101_2$. The reception sequence is from LSB to MSB, and the data move through the inbound shift register from MSB to LSB. The data word is preceded by a start-bit and followed by a stop-bit. With *rst_b* having a value of 0, the state is *idle* and the counters are cleared. At the first active edge of *Sample_clock* after the reset condition is de-asserted, with *Ser_in_0_in* having a value of 1, the controller's state enters *starting* to determine whether a start-bit is being received. Three more samples of *serial_in* are taken, and after a total of four samples have been found to be 0, the *Sample_counter* is cleared and the state enters *receiving*. After the eighth sample, *shift* is asserted. The sample at the next active edge of the clock is shifted into the MSB of *RCV_shftreg*. The value of *RCV_shftreg* becomes $80_h = 1000_0000_2$. The sampling cycle repeats again, and a value of 0 is sampled and loaded into *RCV_shftreg*, changing the contents of the register to $0100_0000_2 = 40_h$.

The end of the sampling cycle of the received word is shown in Figure 7-28. After the last data bit is sampled, the machine samples once more to detect the stop

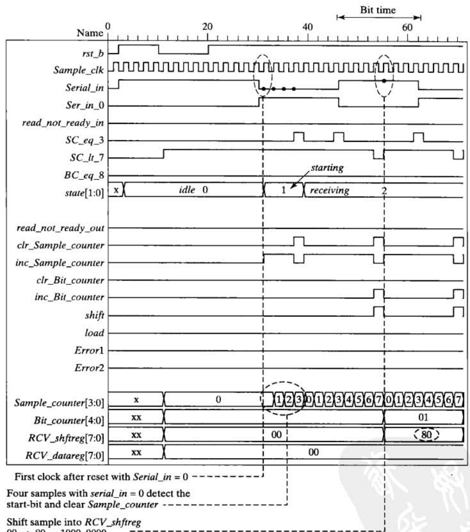
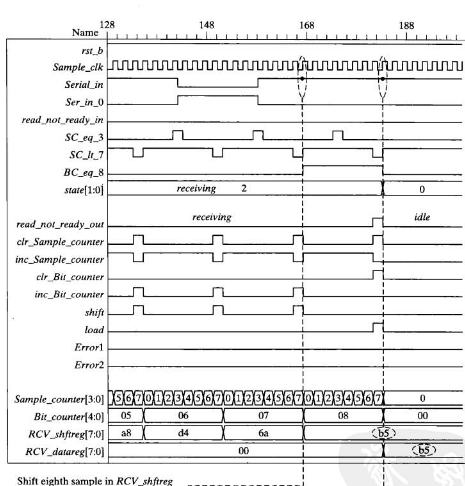


FIGURE 7-27 Annotated simulation results for *UART_receiver*.



Shift eighth sample in *RCV_shftreg*
$6a_h = 0110_1010_2 \text{->} 1011_0101_2 = b5_h$

Verify stop-bit
Load *RCV_datareg* with *RCV_shftreg*

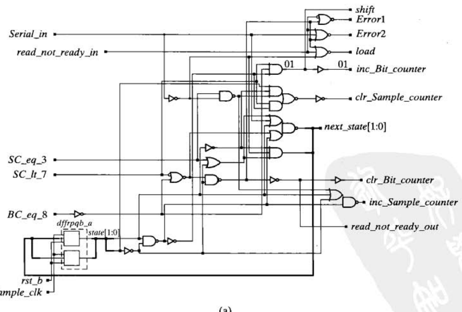
FIGURE 7-28 Transfer of data word into *RCV_datareg* at the end of sampling.

bit. In the absence of an error, the contents of *RCV_shftreg* will be loaded into *RCV_datareg*. In this example, the value $b5_h$ is finally loaded from *RCV_shftreg* into *RCV_datareg*. Other tests can be conducted to completely verify the functionality of the receiver.

The Verilog description of the partitioned receiver synthesizes into the circuit shown in Figure 7-29.

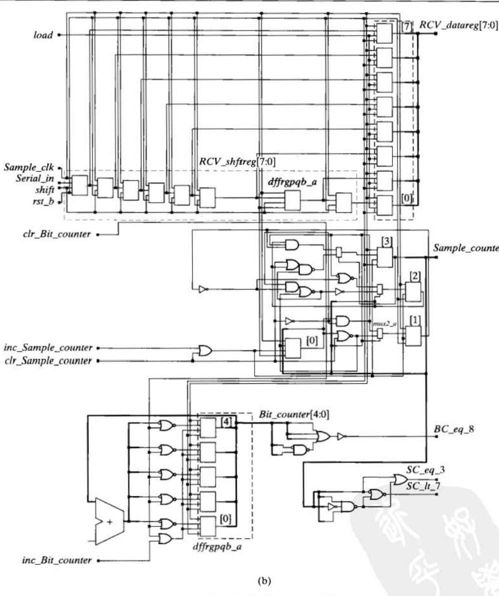
The control unit in Figure 7-29(a) has two flip-flops and combinational logic. Its output signals control the datapath unit shown in Figure 7-29(b). The datapath unit has *RCV_shftreg*, which accepts the serial input bits, *RCV_datareg*, which holds the parallel word formed by the contents of *RCV_shftreg*, the sample counter and the bit counter. A small amount of additional combinational logic forms the status signals *BC_eq_8*, *SC_eq_3*, and *SC_lt_7*.

The machines use two types of flip-flops: the four-input *dffrgpqb_a* and the three-input *dffrpqb_a*. The former is a D-type flip-flop with internal gated data between the external datapath and the output, a rising clock, and an asynchronous active-low reset; the latter is a D-type flip-flop with data, rising clock, and asynchronous active-low reset. Only the state register uses the *dffrpqb_a* flip-flop.



(a)

FIGURE 7-29 Circuits synthesized from *UART_receiver*: (a) control unit and (b) datapath unit, including logic to generate status signals *BC_eq_8*, *SC_eq_3*, *and SC_lt_7*.



(b)

FIGURE 7-29 Continued

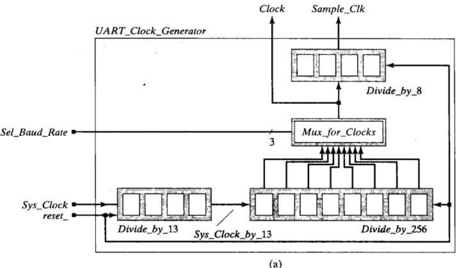
REFERENCES

1. Ernst R. "Target Architectures." *Hardware/Software Co-Design: Principles and Practice*. Boston, MA: Kluwer, 1997.
2. Gajski D, et al. "Essential Issues in Design," In: Staunstrup J, Wolf W, eds. *Hardware/Software Co-Design: Principles and Practice*. Boston, MA: Kluwer, 1997.

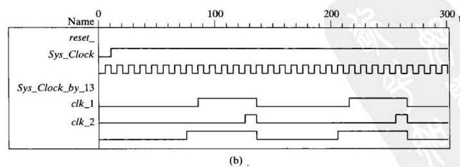
3. Hennessy JL, Patterson DA. *Computer Architecture—A Quantitative Approach*. 4th ed. San Francisco, CA: Morgan Kaufman, 2006.
4. Heuring VP, Jordan HF. *Computer Systems Design and Architecture*. Upper Saddle River, NJ: Prentice-Hall, 2004.
5. Roth CW, Jr. *Digital Systems Design Using VHDL*. Boston, MA: CL-Engineering, 1998.

PROBLEMS

1. Develop, verify, and synthesize *Johnson_Counter_ASMD*, a Verilog behavioral module of a 4-bit Johnson counter based on a direct implementation of the ASMD chart in Figure 7-6. *Hint*: Use a Verilog function to described *next_count*.
2. The functional unit *UART_Clock_Generator* in Figure P7-2 can be used to create a set of baud rate signal pairs for use in the UART in Figure 7-16. Table P7-2 shows



(a)



(b)

FIGURE P7-2

TABLE P7-2

Sel_Baud_Rate	*Clock*	*Sample_Clock*
000	307,696	38462
001	153,838	19231
010	76,920	9615
011	38,464	4808
100	18,232	2404
101	9,616	1202
110	4,808	601
111	2,404	300.5

the pairs that are generated if *Sys_Clock* is an 8-MHz signal. The code for an experimental version of *Divide_by_*13 is also given, with *temp* being a 4-bit counter that counts from 0 to 12 before recycling, at 1/13 the frequency of *Sys_Clock*. The condition that register *temp* has a 1 in its MSB generates the *signal Sys_Clock_by_*13, which is true for the last five contiguous cycles. The condition that *temp* has the value 12 generates the signal *clk_*1; *clk_2* is generated from the condition that *temp* is greater than 6 and produces a more symmetric waveform than the two others.
(a) Synthesize three versions of *Divide_by_*13, one for each of the three possibilities illustrated by the waveforms in Figure P7-2, and compare their circuits.
(b) Choose one of the methods for forming *Sys_Clock_by_13*, then develop, verify, and synthesize the complete description of *UART_Clock_Generator*.

```
module Divide_by_13 (output Sys_Clock_by_13, input Sys_Clock, reset_);
  reg    [3: 0]    temp;
assign Sys_Clock_by_13 = temp[3];
  //wire clk_1 = (temp == 12);
  //wire clk_2 = (temp > 6);
  always @ (posedge Sys_Clock,negedge reset_ )
    if (reset_ == 0) temp <= 4'b0000;
    else if (temp == 4'd12) temp <= 4'd0;
    else temp <= temp +1;
endmodule
```

3. A counter is said to enter an abnormal state if it enters a state that is not explicitly decoded in its next-state function. A self-correcting counter has the ability to recover from an abnormal state. The key is to choose default assignments that ensure recovery. Develop and verify a Verilog model of a self-correcting 4-bit Johnson counter using the next state 0001_2 if the current state is 0--0, with "-" denoting a don't-care. Demonstrate that the counter is self-correcting, and synthesize the circuit. *Hint*: Consider using Verilog's **force** ... **release** construct pair in the testbench to drive the counter into an abnormal state.
4. Modify the counter in Figure 7-3 to wait for three clock cycles after *enable* is asserted before asserting *enable_DP*.
5. The register transfers of the UART transmitter in Figure 7-18 decode the controlling signals in separate ***if*** statements. Discuss whether the signals controlling the datapath can be in conflict, and whether the controlling signals should have

been decoded by a priority decoder. Modify the datapath unit to use a priority decoder and determine whether the synthesized circuit differs from the circuit that was synthesized from *UART_Transmitter_Arch*.

6. Verify the models of *Processing_Unit*, *Control_Unit*, and *Memory_Unit* in *RISC_SPM* and then demonstrate that the machine (see Section 7.3) correctly executes its complete instruction set. After verifying the functionality of the machine, synthesize and verify standard-cell-based realizations of the individual units and then integrate the synthesized units to verify the synthesized model of *RISC_SPM*.
7. Modify *ALU_RISC* in *RISC_SPM* (see Section 7.3) to handle carries. Verify and synthesize the new machine.
8. (a) Develop and verify *Binary_Counter_Arch*, a Verilog model with the hierarchical partition, module names, and port structure (*count, enable, clock, rst*) shown in Figure 7-3, with *count* having a parameterized width. Also, develop and verify the nested modules *Control_Unit* and *Datapath_Unit_Arch*, as well as the child modules, *Mux* and *Count_Register*. (b) Synthesize *Binary_Counter_Arch* and *Binary_Counter_Behav_imp* (an implicit-state machine behavioral model) for a 4-bit wide datapath. (c) Compare the synthesized circuits. (d) Compare the results of simulating the counters, using a common testbench.
9. Develop, verify, and synthesize *Binary_Counter_STG*, using the STG in Figure 7-4 as a guide. Compare to the results of Problem 8.
10. The implicit Moore machine with a modified control unit in *Binary_Counter_Part_RTL_by_3* (see Example 7.1) increments the counter every third clock, but takes one extra cycle to recover from a reset condition (i.e., the first increment of the counter occurs at the fourth clock edge after reset is de-asserted). Using a testbench, verify this claim. Explain why this behavior occurs, then develop and verify an alternative (partitioned) machine that will recover from a reset condition after three clock edges and increment thereafter on every third edge.
11. Modify the datapath and control units in *Binary_Counter_Part_RTL* (see Example 7.1) to implement *Johnson_Counter_RTL_by_3*, a Johnson counter that increments every third edge of the external clock. Synthesize the behavioral model of the implicit state machine (control unit) or an equivalent explicit state machine, and verify the functionality of the synthesized circuit.
12. The top level architecture shown in Figure P7-12a depicts *Gap_Finder*, a sequential machine that (1) loads a 16-bit word from a data bus, (2) finds the largest gap between two successive 1s in the word, and (3) produces an output word, *Gap*, whose value is the binary equivalent of the gap size. The datapath unit (*Datapath*) is controlled by a finite state machine (*Controller*) under the direction of an external agent. When *Run* is asserted the machine loads data from its external datapath, finds the largest gap between two ones, generates a 4-bit output, *Gap*, the binary equivalent of the size of the largest gap, and asserts a handshake signal, *Done*. The signals generated by the state machine to control the datapath are shown in Figure P7-12a, together with status signals that are fed back from the datapath to the controller.

    When loaded by the signal *load_Data*, the datapath register *Data_int* holds the 16-bit word provided by the external bus. The word may contain 0,1 or multiple gaps in its data pattern. Register *tmp* is a 4-bit word that holds the size of a gap as the data is processed by a sequential algorithm. Register *Gap* holds the size of the largest gap that has been found, dynamically, as the process evolves. At the end of processing, it holds the output of the machine. Register *k* is a 4-bit counter that addresses the bits of *Data_int*.

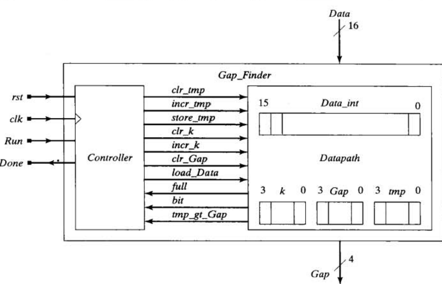


FIGURE P7-12a Block diagram for *Gap_Finder.*

The partially completed ASMD chart in Figure P7-12b shows the datapath register operations that occur concurrently with state transitions of the controller, but, for simplicity, does not contain conditional output boxes that identify the (Mealy) output signals that are generated by the controller to direct the activity of the datapath. Instead, an alternative annotation is used in which the control signal and associated action are denoted by the control signal–register action pair, e.g., *clr_tmp*: *tmp* <= 0. The status signals generated by the datapath unit are identified in the decision boxes of the chart.

The machine (1) enters *S_0* synchronously from any state under the action of *rst*, (2) initializes registers *k*, *Gap*, and *tmp*, and (3) idles in state *S_0* while *rst* is asserted. With *Run* asserted, the machine transitions to *S_1* at the first active edge after *rst* is de-asserted. The external datapath is loaded into an internal register, *Data_int*, during the transition to *S_1*, thereby freeing the external data bus for other activity.

In *S_1*, the datapath unit's bit counter, *k*, sequences through the bits of *Data_int* and skips any leading 0's in the word until finding the first 1. If no bit is found to be 1 (*full* is asserted), the machine transitions to *S_4*, where the Moore output *Done* is asserted and the machine waits until *Run* is asserted again. Upon finding the first 1 in *Data_int* the machine transitions to *S_2*, where a similar process skips over contiguous 1s. Otherwise, the next string of 0s is skipped until a 1 is found. At this point, the algorithm compares the current value of *tmp* to *Gap*. If *tmp* > *Gap* then *tmp* is loaded into *Gap* and the machine returns to *S_2* to continue searching for additional Gaps until the last bit of *Data_int* has been processed. Note that the Moore-type output *Done* remains asserted in *S_4*, and does not assert in *S_0*.

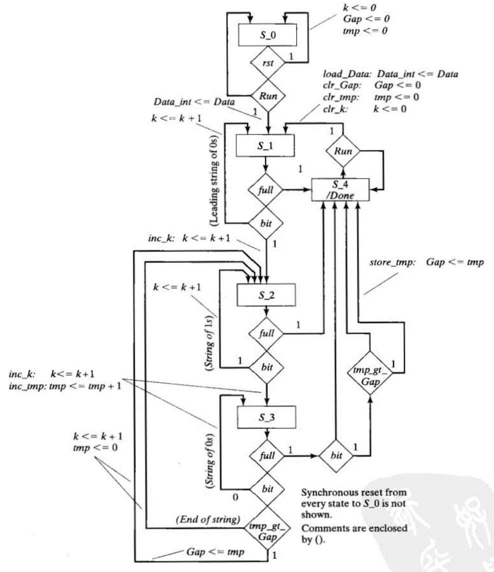


FIGURE P7-12b ASMD chart for *Gap_Finder*.

The output signals of the controller have the following functionality:

Done;	// asserts when algorithm is done (i.e., state is *S_5*).	
clr_tmp;	// tmp <= 0	clears register *tmp*
incr_tmp;	// tmp <= tmp + 1	increments register *tmp*

store_tmp;	// Gap <= tmp	stores the contents of register tmp in register *Gap*
clr_k;	// k <= 0	clears register *k*
incr_k;	// k <= k + 1	increments bit counter register *k*
clr_Gap;	// Gap <= 0	clears register *Gap*
load_Data;	// Data_int <= Data	loads *Data_int* with external data

The inputs to *Controller* are defined below:

tmp_gt_Gap;	// asserted in the datapath unit when *tmp* > *Gap*
full;	// asserted in datapath unit when index *k* = 4'b1111
bit;	// value of *Data_int* at index *k*
Run;	// initiates sequential algorithm
clk;	// clock for synchronous activity
rst;	// active-high rst

(a) Using Figure P7-12b and the partially completed code for the modules *Gap_Finder*, *Datapath*, and *Controller* given below, develop a Verilog model of *Gap_Finder*. The code provided includes all signals that will be needed and includes partially declared cyclic behaviors that (1) synchronize the state transitions, (2) generate the next state and outputs, and (3) control the datapath registers.
(b) Using the test bench below (also available at the web site), verify the design. Organize your graphical output to list the signals in the order shown in Figure P7-15c. (*Note*: data is shown in hexadecimal format; *Gap, tmp* ,and *k are* shown in decimal format.)
The testbench, *t_Gap_Finder*, considers:

(1) Power-on rst
(2) A single gap shrinking from a maximum size (14) to 0, from left to right
(3) A single gap shrinking from a maximum size (14) to 0, from right to left
(4) A single gap growing from 0 to a maximum size (14), from left to right
(5) A single gap growing from 0 maximum size (14), from right to left
(6) Recovery from *rst* on-the-fly
(7) Recovery from cycling *Run* on-the-fly
(8) Miscellaneous patterns having multiple gaps (growing and shrinking)
(9) All 1s
(10) All 0s
(11) Leading 0s and no gap
(12) Leading 1s and no gap

Note: *Load_data* anticipates the clock that loads the data, i.e., *Load_data* is asserted in the previous cycle. The stimulus generator generates *Data* when triggered by the rising edge of *load_Data*. *Data* is available for the datapath to consume it at the clock corresponding to the falling edge of *load_Data*. The expected value of *Gap* is generated on the falling edge of *load_Data*. *Gap* and *expected_Gap* are compared at the rising edge of *Done*. The difference is reported as the signal *error*.

It is critical that a testbench take into account latency in datapaths in order to maintain coherency between expected and actual values of signals. Since *load_Data* and *Done* may assert at the same time (i.e., in *S_4*), we pipeline *Data* to create *old_Data*, for use in eliminating a race between *data* and *Done* in the standard output listing of the results. This is necessary because *Data* is

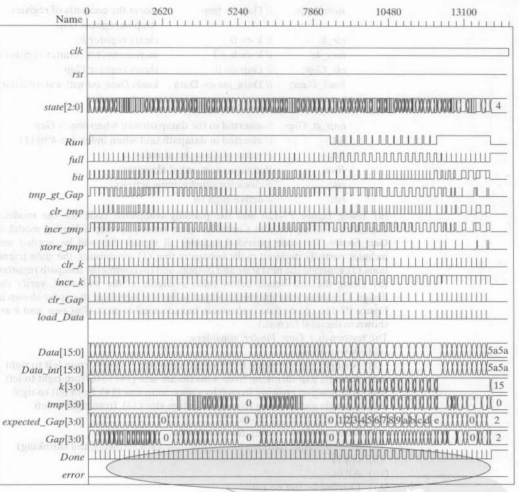


FIGURE P7-12c Simulation results demonstrating error-free operation.

updated on the rising edge of *load_Data*, and *Gap* is updated at the same time. *old_Data* corresponds to the *Gap* that is listed, and the comparison is coherent.

Sample simulation results are presented in Figures P7-12c, d, and e. Note that the signals *error* and *Done* in Figure P7-12c indicate error-free operation, serving as a quick check for behavior, even though the resolution of the displayed data suppresses other details of operation.

(c) Confirm that your model is error free for the entire set of patterns provided and matches the output shown in Figure P7-12c, and the samples shown in Figures P7-12d, e. Produce simulation results for *Data* = 16'h0000,

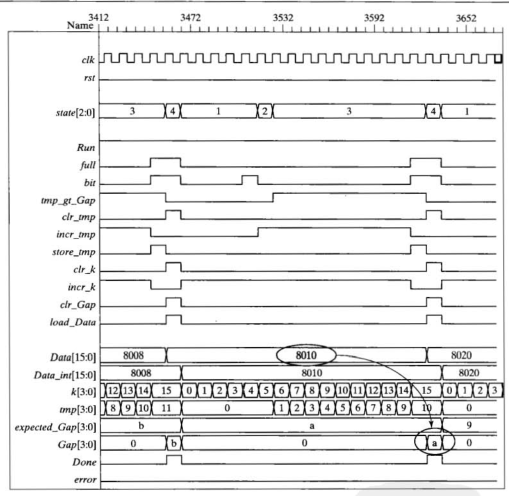


FIGURE P7-12d Simulation results for *Data* = 8010_{16} = $1000_0000_0001_0000_2$; Gap = 10_{10} = a_{16}.

16'hffff, 16'h8295, 16'ha441, and verify by observation that *Gap* and *expected_Gap* are correct and match.
(d) Remove the comments on the signal *rst* from the testbench and verify that your model recovers correctly from asserting *rst* on-the-fly. Replace the comments that were removed above and remove the comments on the signal *Run* from the testbench and verify that your model recovers correctly from cycling *Run* on-the-fly. Verify that the standard output produced by your machine is correct. (A sample is listed after the testbench.)
(e) Synthesize your model of *Gap_Finder*. Indicate whether the synthesized circuit contains latches. Determine whether the synthesized circuit duplicates the simulation results of your behavioral model.

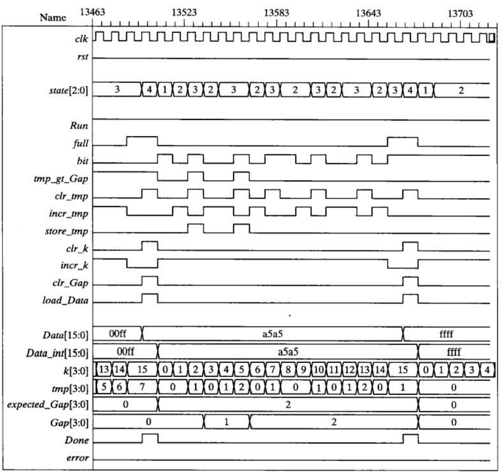


FIGURE P7-12e Simulation results for $Data = a5a5_{16} = 1010_0101_1010_0101_2$; $Gap = 2_{10} = 2_{16}$.

```
module Gap_Finder (Gap, Done, Data, Run, clk, rst);
// Declarations go here

  Datapath M1
   (Gap, tmp_gt_Gap, full, bit, Data, clr_tmp, incr_tmp, store_tmp, clr_k, incr_k,
     clr_Gap, load_Data, clk, rst);

  Controller M2
   (Done, clr_tmp, incr_tmp, store_tmp, clr_k, incr_k, clr_Gap, load_Data,
     tmp_gt_Gap, full, bit, Run, clk, rst);
endmodule
```

```
module Datapath (/* Declare ports here */);
 // Declarations go here

 assign          bit = Data_int[k];
 assign          full = (k == 15);
 assign          tmp_gt_Gap = (tmp > Gap);

 always @ (posedge clk)
 // Your model of the Datapath operations goes here
endmodule

module Controller (Done, clr_tmp, incr_tmp, store_tmp, clr_k, incr_k, clr_Gap,
  load_Data, tmp_gt_Gap, full, bit, Run, clk, rst);
 // Declarations go here
 assign   Done = (state == S_4);

 always @ (posedge clk)
  if (rst == 1) state <= S_0;
  else state <= next_state;

 always @ (/* Your sensitivity list goes here */)
 // Your logic for state transitions and outputs goes here

endmodule

module annotate_t_Gap_Finder ();
 defparam t_Gap_Finder.M2.LATENCY = 10;
 defparam t_Gap_Finder.M2.OFFSET = 5;
 defparam t_Gap_Finder.M2.PULSE_WIDTH = 5;
endmodule

module Clock_Prog (clock);
 output clock;
 reg clock;
 parameter LATENCY = 100;
 parameter PULSE_WIDTH = 50;
 parameter OFFSET = 50;
 initial begin
  #0 clock = 0;
  #LATENCY forever
   begin #OFFSET clock = ~clock; #PULSE_WIDTH clock = ~clock; end
 end
endmodule

module t_ Gap_Finder  ();
 wire       eexpected_Gapeeexpected_Gap &and; Gap)) && (!rst);

 Gap_Finder M1 (Gap, Done, Data, Run, clk, rst);        // UUT
 Clock_Prog M2 (clk);                                   // Programmable clock unit
```

```
initial fork
 #1 $display ("");
 #1 $display ("Start of simulation");
 #20000 $display ("");
 #20000 $display ("End of simulation");
 #20001 $stop;
join

initial fork
 #5 rst = 1;        // initial rst
 #25 rst = 0;
join

initial fork        // Test for Run action
 #5 Run = 0;
 #60 Run = 1;
 #14000 Run = 0;
join
always @(negedge M1.load_Data) old_Data = Data;

always @(posedge Done)
  #1 $display ($time,,"Data = %b expected_Gap = %d Gap = %d error = %d",
  old_Data, expected_Gap, Gap, error);

initial begin
 Data = 16'h0001;
 pointer = 18'h8000;
 expected_Gap = 15;

//////////////////////////////////////////////////////////////////////
  #1$display ("");
  $display ("Test Patterns: shrinking Gap from left to right");
  repeat (16) begin
   @(posedge M1.load_Data)
   Data = 16'h0001 | pointer;
   @ (negedge M1.load_Data)
   expected_Gap = (pointer > 2)? expected_Gap -1: 0;
   pointer = pointer >> 1;
  end

  pointer = 16'h0001;
  @ (posedge M1.load_Data)
  Data = 16'h8000 | pointer;
  @ (negedge M1.load_Data)
  expected_Gap = 14;
  pointer = pointer << 1;

//////////////////////////////////////////////////////////////////////
  $display ("");
```

```
    $display ("Test Patterns: shrinking Gap from right to left");
    repeat (15) begin
     @(posedge M1.load_Data)
     Data = 16'h8000 | pointer;
     @ (negedge M1.load_Data)
     expected_Gap = (pointer < 32768)? expected_Gap -1: 0;
     pointer = pointer << 1;
    end

// Crossover pattern between sets
    pointer = 16'h8000;
    @ (posedge M1.load_Data)
    Data = 16'h8000 | pointer;
    @ (negedge M1.load_Data)
    expected_Gap = 0;
    pointer = pointer >> 1;

//////////////////////////////////////////////////////////////////////
    $display ("");
    $display ("Test Patterns: expanding Gap from left to right");
    repeat (15) begin
     @(posedge M1.load_Data)
     Data = 16'h8000 | pointer;
     @ (negedge M1.load_Data)
     expected_Gap = (pointer < 16384)? expected_Gap + 1: 0;
     pointer = pointer >> 1;
   end

// Crossover pattern between sets
    pointer = 16'h0001;
    @ (posedge M1.load_Data)
    Data = 16'h0001 | pointer;
    @ (negedge M1.load_Data)
    expected_Gap = 0;
    pointer = pointer << 1;

//////////////////////////////////////////////////////////////////////
    $display ("");
    $display ("Test Patterns: expanding Gap from right to left");
    repeat (15) begin
     @(posedge M1.load_Data)
     Data = 16'h0001 | pointer;
    // Remove comments to include test
    //$display ("Test for recovery from rst on-the-fly");
    // Remove comments to include test
    //#40 rst = 1;       // machine should park in state S_0
    //#40 rst = 0;

    //$display ("");
    //$display ("Test for cycling of Run on-the-fly");
```

```
// Remove comments to include test
// #40 Run = 0;     // machine should park in state S_4
// #200 Run = 1;
  @ (negedge M1.load_Data)
  expected_Gap = (pointer > 2)? expected_Gap + 1: 0;
  pointer = pointer << 1;
end

@ (posedge M1.load_Data) ;
@ (negedge M1.load_Data) ;

$display ("");
$display ("Miscellaneous patterns");
@ (posedge M1.load_Data) Data = 16'b0001_0000_0000_1000;
@ (negedge M1.load_Data) expected_Gap = 8;

@ (posedge M1.load_Data) Data = 16'h0ff0;
@ (negedge M1.load_Data) expected_Gap = 0;

@ (posedge M1.load_Data) Data = 16'b0001_0000_1000_0000;
@ (negedge M1.load_Data) expected_Gap = 4;

@ (posedge M1.load_Data) Data = 16'h0000;
@ (negedge M1.load_Data) expected_Gap = 0;

@ (posedge M1.load_Data) Data = 16'haaaa;
@ (negedge M1.load_Data) expected_Gap = 1;

@ (posedge M1.load_Data) Data = 16'hff00;
@ (negedge M1.load_Data) expected_Gap = 0;

@ (posedge M1.load_Data) Data = 16'h00ff;
@ (negedge M1.load_Data) expected_Gap = 0;

@ (posedge M1.load_Data) Data = 16'ha5a5;
@ (negedge M1.load_Data) expected_Gap = 2;

@ (posedge M1.load_Data) Data = 16'hffff;
@ (negedge M1.load_Data) expected_Gap = 0;

@ (posedge M1.load_Data) Data = 16'h5a5a;
@ (negedge M1.load_Data) expected_Gap = 2;

@ (posedge M1.load_Data) Data = 16'h5a5a;
@ (negedge M1.load_Data) expected_Gap = 2;

@ (posedge M1.load_Data) Data = 16'hc225;
@ (negedge M1.load_Data) expected_Gap = 4;
```

```
    @ (posedge M1.load_Data) Data = 16'ha443;
    @ (negedge M1.load_Data) expected_Gap = 4;

    @ (posedge M1.load_Data) Data = 16'h8121;
    @ (negedge M1.load_Data) expected_Gap = 6;

  end
endmodule
```

A sample of the standard text output produced by the testbench is listed below.

```
   Test Patterns: expanding Gap from left to right
5666 Data = 1000000000000000 expected_Gap = 0 Gap = 0 error = 0
5836 Data = 1100000000000000 expected_Gap = 0 Gap = 0 error = 0
6006 Data = 1010000000000000 expected_Gap = 1 Gap = 1 error = 0
6176 Data = 1001000000000000 expected_Gap = 2 Gap = 2 error = 0
6346 Data = 1000100000000000 expected_Gap = 3 Gap = 3 error = 0
6516 Data = 1000010000000000 expected_Gap = 4 Gap = 4 error = 0
6686 Data = 1000001000000000 expected_Gap = 5 Gap = 5 error = 0
6856 Data = 1000000100000000 expected_Gap = 6 Gap = 6 error = 0
7026 Data = 1000000010000000 expected_Gap = 7 Gap = 7 error = 0
7196 Data = 1000000001000000 expected_Gap = 8 Gap = 8 error = 0
7366 Data = 1000000000100000 expected_Gap = 9 Gap = 9 error = 0
7536 Data = 1000000000010000 expected_Gap = 10 Gap = 10 error = 0
7706 Data = 1000000000001000 expected_Gap = 11 Gap = 11 error = 0
7876 Data = 1000000000000100 expected_Gap = 12 Gap = 12 error = 0
8046 Data = 1000000000000010 expected_Gap = 13 Gap = 13 error = 0
8216 Data = 1000000000000001 expected_Gap = 14 Gap = 14 error = 0
```

13. Develop, verify, and synthesize a frequency divider with a programmable divisor for the base frequency, and a programmable duty cycle.
14. Develop, verify, and synthesize a Verilog model of a decoder that will decode a 16-bit address to determine in which of eight 8-k segments of a 64-k memory the word resides.
15. Describe the differences between the circuits that will be synthesized from the following Verilog cyclic behaviors:

```
always @ (a,b,c,d) y = a + b + c + d;
always @ (a,b,c,d) y = (a + b) + (c + d);
```

16. The instruction set of *RISC_SPM* is limited and might not serve a particular application very well. Develop *RISC_SPM_e* and an enhanced version of *RISC_SPM* with additional instructions that would be useful if it is to be an embedded processor within a vending machine that is to accept currency, make change, and dispense coffee in response to selections made by the customer. The allowed selections are identified in Figure P7-16. The machine is to assert signals that (1) control dispensing units that blend the coffee according to the customer's choices, (2) accept currency and dispense change, and (3) send messages to a display panel.

Size	Venti	Grande	Tall
Coffee	Normal	Decaff	Espresso
Flavor	Hazelnut	Vanilla	Raspberry
Creamer Type	Half-n-Half	Whole	Skim
Creamer Amount	Heavy	Medium	Tall
Sweetener Type	Sugar	Artificial	Light
Sweetener Amount	Heavy	Medium	Equal

FIGURE P7-16

17. Write, verify, and synthesize a partitioned Verilog model (separate controller and datapath) of the two-state pipeline described by Figure 5-24.
18. The block diagram below shows the datapath and controller for a machine that (a) transfers two 16-bit signed numbers in 2's complement representation into registers *AR*, and *BR*, (b) divides the number in *AR* by 2 and transfers the result to register *CR*, if the number in *AR* is negative, (c) multiplies the number in *BR* by 2 and transfers the result to register *CR*, if the number in *AR* is positive but nonzero, and (d) if the number in *AR* is 0, clears register *CR* to 0. Develop an ASMD chart for the machine, write, and verify a partitioned Verilog model of the circuit. *Hint*: Consider using the arithmetic right shift operator. The reset action of the machine is active-low.

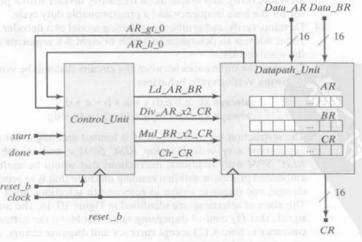


FIGURE P7-18

CHAPTER 8

Programmable Logic and Storage Devices

As technology advances, the density, complexity, and size of field-programmable gate arrays (FPGAs) provide an attractive, cost-efficient, and increasingly important alternative to semi-custom application-specific integrated circuits (ASICs). Mask charges for cell-based ASICs can cost several $100k, eliminating these devices from the low-volume end of the market. The opportunity to realize large circuits in FPGAs has created pressure for a change in the method by which circuits are designed for FPGA-based applications. Historically, designers with sufficient experience could be productive and efficient using schematic entry tools when designs are small, but the clear trend today is toward complex and larger designs targeted for FPGAs. The language-based design methodology that has served the ASIC design flow is also essential to FPGA-based design flows, because it is the key to meeting ever-shrinking windows of opportunity for new products. As a result, FPGA vendors have greatly improved their support of language-entry tools for FPGAs, and designers have shifted from schematic entry tools to language-entry tools. This chapter will emphasize a design flow for FPGAs that is entirely Verilog-based.

The technologies available for implementing digital circuits range from the discrete gates and standard integrated circuits (ICs) used in low-density/low-performance applications, to cell-based and full-custom ICs for high-density/high-performance circuits. Standard ICs can be manufactured cheaply, but they implement very limited, basic functionality at low levels of integration. Production of customized logic, having a small market, creates an inventory risk because the quantities that could be sold do not warrant the expense of their development and production, and IC manufacturers cannot afford to stock multiple variants of specialized functional units, as they do with

standard parts. Inevitable progress in physical technology alone would render their inventory worthless before their investment could be recovered. This chapter will consider programmable logic devices (PLDs), which lie between the two extremes of density and performance that characterize standard parts and semi- and full-custom circuits.

PLDs were born out of a necessity created by two conflicting realities: large, dense, high-performance circuits cannot be built practically, reliably or economically from discrete devices, and dedicated ICs cannot be produced economically and sold profitably to satisfy a diversity of low-volume applications. The resolution of these forces lies in PLDs.

Although read-only memories (ROMs), programmable logic arrays (PLAs), programmable array logic (PAL), complex PLDs (CPLDs), FPGAs, and mask-programmable gate arrays (MPGAs) are all programmable, we will use the term *PLD* to indicate the low-density structures that were introduced to implement two-level combinational logic: PLAs, PALs, and similar vendor-named devices. PLDs are distinguished by their having a regular structure of identical basic functional units with fixed architecture. MPGAs are formed from a regular array of transistors. Unprogrammed MPGAs have an identical structure. They are programmed by adding layers of metal interconnects to compose and connect macros with a desired functionality. On a local basis, the interconnect might establish, for example, the connectivity that forms a NAND gate, and on a global basis establish the functionality of an adder. The architecture of the basic functional units remains unchanged, but the interconnection fabric (i.e., the layers of metallization) is unique to the application. In contrast, standard cell layouts do not have a fixed, basic architecture of functional units. There is regularity in the structure of the layout channels, but the functional units themselves are not uniform and do not have an architecturally determined placement. One cell may implement an inverter, another a flip-flop. The use and location of a given cell are application dependent. Nor is a cell itself programmed (i.e., transistors are not connected to establish a basic functionality). The overall architecture of a cell-based design is completely flexible within the constraints imposed by layout routing channels and a library of cells. No cell pattern need be replicated in a cell-based layout. For these reasons, *we make a distinction between programming that overlays an interconnection fabric on a given fixed architecture of devices, and programming that establishes an architecture from fixed, pre-designed and characterized functional units, as is the case with standard cells.* We mean the former case when we use the term *PLD* and do not include standard cells in that category.

Storage devices, such as ROMs, are considered to be PLDs because they can implement combinational logic by storing the values of a function at memory locations that are addressed by the inputs of the function. These implementations, of necessity, implement the full truth table of the function. Memory-implemented combinational logic may be inefficient, because minimization techniques are not used to implement a full truth table of a function, and device resources might not be fully utilized.

8.1 Programmable Logic Devices

PLDs have a fixed architecture, but their functionality is programmed for a specific application, either by the manufacturer or by the end user. PLDs whose architecture is programmed by the manufacturer are referred to as *mask-programmable logic devices* (MPLDs); those that are programmed by the end-user are referred to as *field-programmable logic devices* (FPLDs). The architecture of the basic functional unit of a PLD is fixed, and is not customized by the user. Consequently, the development and production costs of PLDs can be amortized over a larger base of customers, and the range of applications for the devices can be very broad. This reduces unit costs for the consumer and production and inventory risks for the manufacturer, while at the same time allowing advances in processing technology to be incorporated into an evolving product line. The design cycle of a system that uses a PLD can be very short because pre-customized PLDs can be manufactured, tested, and placed in inventory in advance of their being chosen as a technology for an application. They are suitable for rapid prototyping of a design.

Three basic characteristics distinguish PLDs from each other: (1) an architecture of identical basic functional units, (2) a programmable interconnection fabric, and (3) a programming technology. ROMs, PLAs, and PALs have the *AND-OR* plane structure shown in Figure 8-1. It implements Boolean expressions in sum-of-products (SOP) form: The *AND* plane forms product terms selectively from the inputs, and the *OR* plane forms outputs from sums of selected product terms. A programmable interconnect fabric joins the two planes, so that the outputs implement SOP expressions of the inputs. Whether and how a plane can be programmed determines the particular type of PLD that is implemented by the overall structure.

8.2 Storage Devices

The architecture used to implement PLDs can implement read-only or random-access storage devices, depending on whether the contents of a memory cell can be written during normal operation of the device. The contents of a read-only memory (ROM)

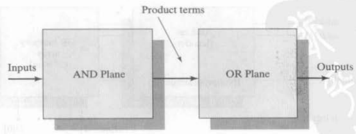


FIGURE 8-1 *AND-OR* plane structure of a programmable logic device

remain unchanged during operation and after power is removed from the device. In contrast, the contents of a random-access memory (RAM) can be changed during operation, and they vanish when power is removed. There is another major distinction between ROMs and RAMs: The circuit of a ROM is structurally modified to program the device prior to its use, but the circuit of a RAM is not programmed—it is fixed. Only the contents of a RAM are programmed, dynamically, during normal read and write operations of the circuit.

8.2.1 Read-Only Memory (ROM)

A $2^n \times m$ ROM consists of an addressable array of semiconductor memory cells organized as 2^n words of m bits each. A ROM has n inputs, called "address lines," and m outputs, called "bit lines." The *AND*-plane of the structure shown in Figure 8-2 serves as an address decoder and is nonprogrammable. The address decoder implements a full decode of the n inputs, and each pattern of input bits addresses a unique decoded output, called a "word line." Each input address word selects one of the 2^n memory words to assert a word line, and each cell of a word stores 1 bit of information. Consequently, each word line corresponds to a minterm of a Boolean expression.

ROMs can be manufactured in a variety of technologies: bipolar, complementary metal-oxide semiconductor (CMOS), n-channel MOS (nMOS), and p-channel MOS (pMOS). A mask-programmed ROM implemented in nMOS technology has the circuit structure shown in Figure 8-3. The bit lines form the output word, and n-channel link transistors connect the word lines to the bit lines. A bit line is normally pulled up to V_{dd}, but when a word line is pulled high by the address decoder, the n-channel transistors that are attached to it will be turned on. This action pulls the corresponding bit lines down. The set of masks that program the device holds the pattern of link transistors attached to a given word, and determines the pattern of 1s and 0s that appear on the bit lines for the applied input address word. Given the three-state output inverters,

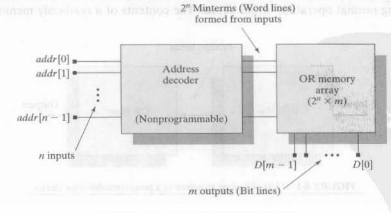


FIGURE 8-2 *AND-OR* planes for a ROM.

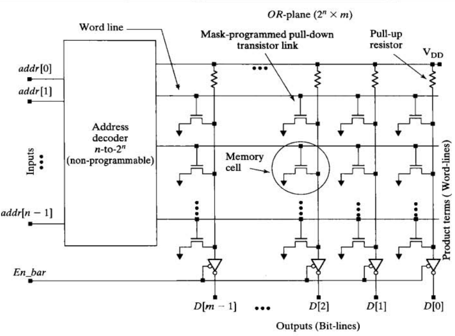


FIGURE 8-3 Circuit structure of a mask-programmed nMOS ROM.

the presence of a link transistor corresponds to a stored 1 at the location of the memory cell. The information stored in a ROM can be read under normal operation of a host circuit, but not written. The outputs of a ROM are normally three-stated, so that the device can be connected to a shared bus serving multiple devices. In commercial ROMs, an additional chip-select input allows multiple devices to be connected to a common bus, each selectable by its unique address. If a ROM has been selected, a pattern of 0s and 1s at its address inputs causes one and only one word line to be asserted.

A $2^n \times m$ ROM can store m different functions of n variables (i.e., truth table storage). Figure 8-4 illustrates a 16×8 ROM having a 4-bit address word and a total

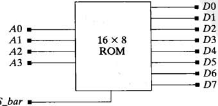


FIGURE 8-4 Schematic symbol for a 16×8 ROM.

TABLE 8-1 Organization and density of commercial ROMs.

Organization	Density
32K × 8	256K bit
64K × 8	512K bit
128K × 8	1M bit
256K × 8	2M bit
512K × 8	4M bit
1024K × 8	8M bit
64K × 16	1M bit
...	
256K × 16	4M bit
512K × 16	8M bit

of 16 memory words of 8 bits each. Commercial ROMs are available in a range of organization and densities, as shown in Table 8-1.

A ROM is a nonvolatile memory because the information remains stored when power is removed from the device. Mask-programmable ROMs are manufactured with a fixed, nonerasable memory pattern, usually for high-volume applications. Their nonrecurring engineering (NRE) cost is relatively high compared to a field-programmable ROM because the mask set that programs the chip is customized to a particular end user's application. The mask set can be produced in about a 4-week cycle. Mask-programmed ROMs are used in applications in which a system needs stored data and has no need to alter the data in ordinary use. For example, they are used as data tables that hold the codes for characters that are to be displayed on the display of a handheld device, and hold the bootstrap program that executes immediately when the device is powered on. ROMs are widely used in electronic point-of-sale terminals in retail stores, instrumentation, domestic appliances, industrial equipment, video games, and a host of handheld devices (e.g., digital cameras).

8.2.2 Programmable ROM (PROM)

A field-programmable ROM (PROM) can be programmed (once) by an end user with a special apparatus called a PROM programmer. PROMs are nonvolatile and nonerasable. Usually manufactured in a bipolar technology, a PROM initially has a pull-up device in the OR-plane at every crosspoint between a word line and the internal bit line. The pull-up device (diode or transistor) is also connected to a metal fusible link, as shown in Figure 8-5. A PROM programmer selectively applies a voltage (10–30 V) to cause current sufficient to vaporize the link, thereby disconnecting the pull-down device from the word line, and permanently causing a 1 to appear in that cell when it is decoded by a word line. The output of the bit line is the inverted content of the memory cell.

The bit line outputs of a PROM are driven by three-state inverters, and each inverter input is connected to ground by a pull-down resistor and is also connected to the internal bit line. In the absence of a signal on a word line, the bit line will be at

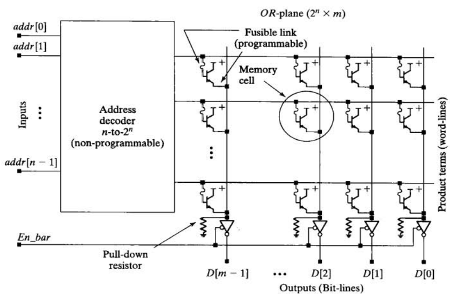


FIGURE 8-5 Circuit structure for a fusible-link bipolar PROM.

ground potential and the output will be high. The enable line in the circuit of Figure 8-5 is active-low, and a high minterm line pulls a bit line output low. Cells having a blown link are not affected by their minterm line, and their output remains at 1, due to the action of the pull-down resistor. Note that a bit-line can be pulled down by one or more cells. In this scheme, the presence of a link transistor implies a 0 in the output word when the word line is decoded. The programming is permanent (i.e., there is no way to restore the blown links and create a different program). A program can be modified however, if the modification affects only links that have not yet been blown.

8.2.3 Erasable ROMs

The architecture of an erasable PROM is similar to that of a PROM, but it uses a floating-gate nMOS transistor as the link device between a word line and the bit lines (Figure 8-6). A floating-gate transistor has an additional gate inserted between the operational gate and the channel. This gate is insulated from the operational gate by a high-impedance dielectric material. When special circuitry (not shown) applies a sufficiently high voltage (e.g., 21 V) to the operational gate the insulator breaks down and a negative charge is pulled from the channel and becomes trapped on the floating gate when the programming voltage is removed. The effect of the trapped charge is to turn off the transistor by depleting the channel of carriers, which effectively raises the threshold voltage of the transistor and breaks the link between the word line and the

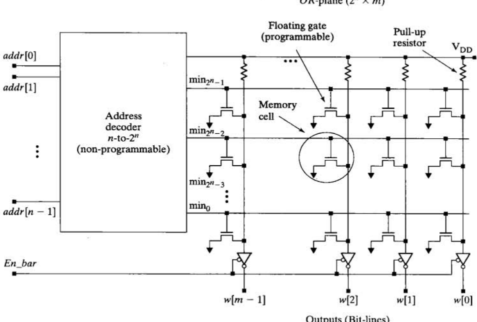


FIGURE 8-6 Circuit structure for a floating-gate EEPROM.

bit line, allowing the bit line to float high and remain high independently of the affected word line. Subsequent reads of the cell are 1. In this scheme, the presence of a programmed link transistor (i.e., one that has trapped charge) implies a 0 in the output word when the word line is decoded.

There are two types of erasable ROMs: those that are erasable by ultraviolet (UV) light and those that are erasable by electricity. The former, called an EPROM or UVEPROM, have a quartz opening and rely on a mechanism by which the UV light at a specific wavelength causes a temporary breakdown of the insulation of the floating gate and allows photocurrents to remove the trapped charge and effectively erase the stored information. The latter type, called an EEPROM (electrically erasable PROM), use electrical pulses to break down the insulated floating gate and erase the stored pattern. Application of a high negative voltage to a minterm line will remove the trapped charge from the floating gate. The UV-erase mechanism of an EPROM is nonselective (also referred to as "bulk erase"); all of the memory contents are reprogrammed to 1. A EEPROM, however, has additional circuitry providing a selective erase capability, allowing individual words to be selectively erased and reprogrammed.

EPROMs are commonly used in the debug phase of firmware development for microprocessor-based systems. They require 5 to 20 minutes of exposure to UV light to

accomplish an erase. ROMs are substituted for production after the program is correct because their packages omit the quartz window and are cheaper. EEPROMs are attractive because they can be programmed in-circuit, can be erased with low current, and do not require the additional hardware and expense of a PROM programmer or a UV source.

Volatility and fatigue are two important considerations in applications of ROM technology. In the absence of UV light, an EPROM is guaranteed to hold 70% of its charge for at least 10 years [1]. The insulating material in an EEPROM is thinner than that for an EPROM, and can deteriorate, so EEPROMs have a limited number of write/erase cycles, typically 10^2 to 10^5. EEPROMs that have exceeded their fatigue limit may fail to hold a charge on the floating gate or may trap charge on the gate. Because they can be erased electrically, they erase much faster than an ordinary EPROM, making them suitable for prototype code development. They are also used in system applications that do not require a high number of write/erase cycles over the useful life of a product, such as storage of default configuration data in a personal computer [1]. EEPROMs are also available with low in-circuit programming voltages (e.g., Atmel AT49LV1024).[1]

8.2.4 ROM-Based Implementation of Combinational Logic

ROMs are commonly used in applications that require a truth table for combinational logic. They are an attractive technology because a ROM can be programmed to implement any of 2^{2^n} different functions of n inputs, and a single ROM can implement any of those functions at any of its bit lines (standard logic would require a new circuit structure for each different function). A ROM-based design can be modified by simply replacing the ROM, without altering the external circuitry. The complexity of the logic being implemented does not have an impact on the effort to program the device, as it would in the case of discrete or building-block logic. ROMs are usually faster than multiple large- and medium-scale integrated (LSI/MSI) devices and other PLDs in moderately sized circuit applications, and often they are faster than an FPGA or custom LSI chip in a comparable technology. On the other hand, for moderately complex functions, a ROM-based circuit is usually more expensive, consumes more power, and may run more slowly than a circuit that uses multiple LSI/MSI devices and PLDs or a small FPGA [1]. Their full address decoding circuitry ultimately limits ROMS to applications that have no more than 20 inputs. Like other semiconductor devices, ROMs benefit from advances in technology that are leading to cheaper and denser devices.

8.2.5 Verilog System Tasks for ROMs

Verilog has two file input–output (I/O) system tasks that can be used to load memory data from a text file, reducing the effort required to initialize a large memory, as an alternative to writing the individual words within the ROM model. A single ROM

[1]See www.atmel.com.

model can serve a variety of applications by substituting text files. The tasks ***$readmemb*** and ***$readmemh*** load to specified locations in a memory the contents of a text file formatted as binary or hexadecimal words, respectively (see "Selected System Tasks and Functions" at the companion Web site: www.pearsonhighered.com/ciletti).

Example 8.1

The truth table of the 2-bit comparator presented in Example 4.4 is shown in Figure 8-7 with a symbolic diagram of the fuse links required to program a PROM-based implementation of the circuit with active-low enabled, three-stated outputs. Note that the output column *D0* is unused, and that the pattern of links accounts for the inverted outputs of the device.

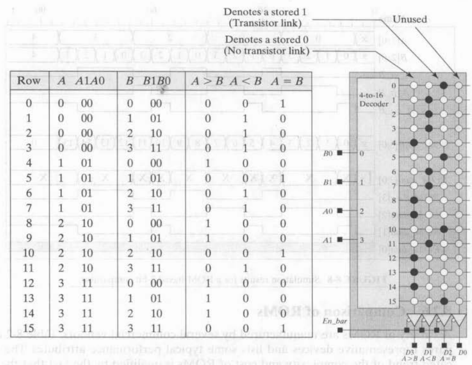
The Verilog model *ROM*_16_x_4 illustrates how to declare a memory of 16 words, each having a width of 4 bits, and how to load the memory from a text file of data.

```
module ROM_16_x_4 (output [3:0] ROM_data, input [3:0] ROM_addr);
 reg          [3:0] ROM [15:0];
 assign ROM_data = ROM [ROM_addr];
 initial $readmemb ("ROM_Data_2bit_Comparator.txt", ROM, 0, 15);
endmodule
```

The contents of a binary-formatted text file[2] for the 2-bit comparator would be listed from address 0 to address 15 as:

```
001x
010x
010x
010x
100x
001x
010x
010x
100x
100x
001x
010x
100x
100x
100x
001x
```

[2] A useful tip: simulation tools expect the text file to be read by ***$readmemb*** or ***$readmemh*** to be located in the same directory (folder) in which the project is located, unless a pathname is specified to a different location.



Row	A	A1A0	B	B1B0	$A > B$	$A < B$	$A = B$
0	0	00	0	00	0	0	1
1	0	00	1	01	0	1	0
2	0	00	2	10	0	1	0
3	0	00	3	11	0	1	0
4	1	01	0	00	1	0	0
5	1	01	1	01	0	0	1
6	1	01	2	10	0	1	0
7	1	01	3	11	0	1	0
8	2	10	0	00	1	0	0
9	2	10	1	01	1	0	0
10	2	10	2	10	0	0	1
11	2	10	3	11	0	1	0
12	3	11	0	00	1	0	0
13	3	11	1	01	1	0	0
14	3	11	2	10	1	0	0
15	3	11	3	11	0	0	1

FIGURE 8-7 Truth table and PROM fuse map for a 2-bit comparator.

The simulation results in Figure 8-8 display the contents of the ROM as a word and as individual bits, illustrating that the unused bit is displayed in Verilog's 4-valued logic as an unknown (ambiguous) logic value (denoted by x). The actual fuse map would have to specify a 1 or 0 for the unused bits.

End of Example 8.1

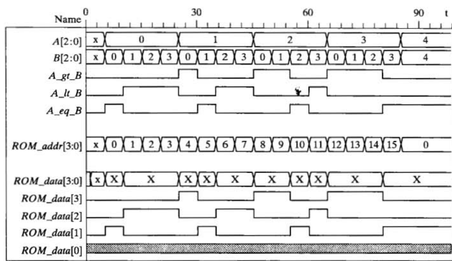


FIGURE 8-8 Simulation results for a ROM-based 2-bit comparator.

8.2.6 Comparison of ROMs

A variety of ROMs are manufactured by several commercial vendors. Table 8-2 compares representative devices and lists some typical performance attributes. The indicated trend of the complexity and cost of ROMs is qualified by the fact that the unit cost of mask-programmed ROMs can be quite low, depending on the volume of parts that are produced. The performance characteristics are a moving target, linked to advances in process technology.

8.2.7 ROM-Based State Machines

ROMs provide a convenient implementation of a state machine and can be an economical implementation if the attributes of the device match the application. The ROM-based state machine shown in Figure 8-9 uses a $2^n \times m$. ROM to store the next-state and output functions of a state machine. The state of the machine is stored in a set of D-type flip-flops, because they typically require fewer outputs from the ROM than would a J-K flip-flop.

The method for designing a ROM-based state machine is simplified because the truth table is implemented directly, without minimization. The size of the array depends on the number of inputs, not on the complexity of the implemented logic. We form a ROM table in which the row address represents the present state of the machine, and the contents associated with that address hold the output and the next state.

PDG

TABLE 8-2 Comparison of (a) ROM types and (b) performance attributes.

Device	Programming Mode	Erase Mode	Complexity and Cost	Example	Access Time
EEPROM	In-circuit Byte-by-byte	In-circuit Byte-by-byte	↑	Intel 2864 8K × 8 nMOS	
FLASH	In-circuit	In-circuit Bulk or sector		AT49LV1024 64K × 16 nMOS	70 ns*
EPROM	Out-of-circuit	Out-of-circuit Bulk, UV Light		Intel 2732 4K × 8 nMOS	45 ns
PROM	Custom by user (OTP**)	None		TMS47C256 32K × 8 CMOS AT27BV400 256K × 16 or 512K × 8	150 ns
ROM***	Mask	None			

*Programming time: 500 ms.
**One-time programmable.
***Requires high volume to offset NRE.

(a)

Type	Technology	Read cycle	Write cycle
ROM	NMOS, CMOS	10–200 ns	4 weeks
ROM	Bipolar	< 100 ns	4 weeks
PROM	Bipolar	< 100 ns	10–50 μs/byte
EPROM	NMOS, CMOS	25–200 ns	10–50 ms/byte
EEPROM	NMOS	50–200 ns	10–50 μs/byte

Adapted from Wakerly JF. *Digital Design—Principles and Practice*, Upper Saddle River, NJ: Prentice-Hall, 2006.

(b)

Example 8.2

A Mealy-type state machine describing a binary-coded decimal (BCD)-to-Excess_3 code converter was developed by manual methods in Example 3.2. Verilog models of a ROM memory and of the machine are listed below. The ROM model is external to the state machine model. In simulation, its contents are written immediately by the ***initial*** (single-pass) behavior that executes when a simulation begins. The listing identifies the

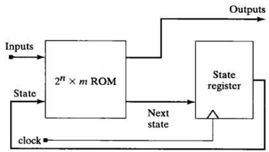


FIGURE 8-9 Block diagram for a ROM-based finite-state machine.

contents stored at each ROM address. The comments identify the state associated with each address and hold the output and next state. A continuous assignment updates the address of the ROM (*ROM_addr*) whenever the state or the input of the machine changes, ensuring that the machine is of the Mealy type. The testbench specifies a simple input sequence for the purpose of illustrating the machine's behavior, shown in Figure 8-10, which matches the behavior of the manually designed gate-level machine

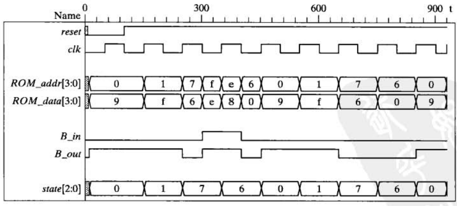


FIGURE 8-10 Simulation results for a ROM-based Verilog model of a BCD-to-Excess_3 code converter.

shown in Figure 3-23. In this example, the contents of the ROM are listed within the ROM,[3] rather than in an external file.

```
module ROM_BCD_to_Excess_3 (output [3: 0] ROM_data, input [3: 0] ROM_addr);
 reg [3: 0] ROM [15: 0];
 assign ROM_data = ROM[ROM_addr];

                                   // input state output next_state
 initial begin
  ROM[0] = 4'b1001;                // S_00 000 1 001
  ROM[1] = 4'b1111;                // S_10 001 1 111
  ROM[2] = 4'b1000;                // S_60 010 1 000
  ROM[3] = 4'b1110;                // S_40 011 1 110
  ROM[4] = 4'bxxxx;                // not used
  ROM[5] = 4'b0011;                // S_20 101 0 011
  ROM[6] = 4'b0000;                // S_50 110 0 000
  ROM[7] = 4'b0110;                // S_30 111 0 110
  ROM[8] = 4'b0101;                // S_01 000 0 101
  ROM[9] = 4'b0011;                // S_11 001 0 011
  ROM[10] = 4'b0000;               // S_61 010 0 000
  ROM[11] = 4'b0010;               // S_41 011 0 010
  ROM[12] = 4'bxxxx;               // not used
  ROM[13] = 4'b1011;               // S_21 101 1 011
  ROM[14] = 4'b1000;               // S_51 110 1 000
  ROM[15] = 4'b1110;               // S_31 111 1 110
 end
endmodule

module BCD_to_Excess_3_ROM (output [3: 0] ROM_addr, output B_out,
 input [3: 0] ROM_data, input B_in, clk, reset
);
 reg [2: 0] state;
 wire [2: 0] next_state;

 assign next_state = ROM_data [2: 0];
 assign B_out = ROM_data[3];
 assign ROM_addr = {B_in, state};

 always @ (posedge clk, negedge reset)
  if (reset == 0) state <= 0; else state <= next_state;
endmodule

module test_BCD_to_Excess_3b_Converter ();
 wire B_out, clk;
 wire [3: 0] ROM_addr, ROM_data;
 reg B_in, reset;
```

[3]This approach would be impractical for a large ROM.

```
    BCD_to_Excess_3_ROM M1 (ROM_addr, B_out, ROM_data, B_in, clk, reset);
    ROM_BCD_to_Excess_3 M2 (ROM_data, ROM_addr);
    clock_gen M3 (clk);

    initial begin #1000 $finish; end
    initial begin
     #10 reset = 0;
     #90 reset = 1;
    end

    initial begin
     #0 B_in = 0;
     #100 B_in = 0;
     #100 B_in = 0;
     #100 B_in = 1;
     #100 B_in = 0;
    end
  endmodule
```

End of Example 8.2

8.2.8 Flash Memory

Flash memory devices are similar to EEPROMs, but have additional built-in circuitry to selectively program and erase the device in-circuit, without the need for a special programmer. They have widespread application in modern technology for cell phones, digital cameras, set-top boxes, digital TV, telecommunications, nonvolatile data storage, and microcontrollers. Flash memory is cost-competitive with a magnetic disk for capacities under 5 MB. Its low consumption of power makes it an attractive storage medium for laptop and notebook computers. Flash memories incorporate additional circuitry too, allowing simultaneous erasing of blocks of memory. Like EEPROMs, flash memories are subject to fatigue, typically having about 10^5 block erase cycles.

8.2.9 Static Random Access Memory (SRAM)

Read-only memories are limited to applications that require retrieval, but not storage, of information during ordinary operation. Computers and other digital systems perform many operations that retrieve, manipulate, transform, and store data, and therefore need read/write memories. For example, an application program must be retrieved from a relatively slow storage medium, such as a floppy disk or a CD-ROM, and moved on demand to a location where it can be accessed quickly by the processor. ROMs are not used to store large application programs, and they cannot dynamically store the data generated by a program's execution. Storage registers and register files support fast, random storage, but cannot be used for mass storage because they are implemented with flip-flops and occupy too much physical area in silicon to support applications that

generate and store vast amounts of data. Small register files may be integrated in an ASIC or an FPGA to avoid having to access an external (slower) memory device.

RAM is faster and occupies less area than a register file, and it serves the function of providing fast storage and retrieval of large amounts of data during the operation of a computer (e.g., a video frame buffer). The name *random* indicates that RAMs allow words of data to be written to or read from any storage location in any order.[4] Most RAMs are volatile—the information they contain vanishes after power is removed from the device. A newer and emerging technology, nonvolatile RAM, will be discussed later in this chapter.

There are two basic types of RAMs: static and dynamic. Static RAMs (SRAMs) are implemented with a transistor-capacitor storage cell structure that does not require refresh; dynamic RAMs (DRAMs) are slower, use fewer transistors, and occupy less physical area, but they require refresh circuitry to retain stored data. They provide the densest storage devices, but their contents must be refreshed every few milliseconds; therefore DRAMs require additional supporting circuitry. SRAMs are used as fast-cache memory in a computer.

The circuit in Figure 8-11 shows the basic structure of an SRAM cell. A pair of inverters are connected in a closed loop and their outputs are tied to pass transistors attached to *Bit_line* and its complement, *Bit_line_bar*. SRAMs commonly use the 6-transistor circuit[5] shown in Figure 8-12. The gate of each pass transistor is connected to the word line of the circuit. Suppose that *Word_enable* is de-asserted, that the stored content of the cell has *cell* = 1 and *cell_bar* = 0, and that the inputs are changed to *Bit_line* = 0 and *Bit_line_bar* = 1. When *Word_enable* is asserted, *cell* is driven to 0 and *cell_bar* is driven to 1. The feedback structure forces the output of one inverter to be the complement of the output of the other inverter.

The values of *Bit_line* and *Bit_line_bar* control the read and write operations. An array of such storage cells is configured with sense amplifiers that are used to read

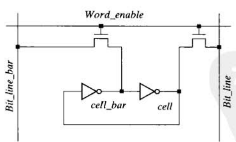


FIGURE 8-11 SRAM circuit structure.

[4]In contrast, note that data are read serially from a tape storage media.

[5]Other schemes use as few as four transistors by replacing the p-channel pull-up transistors with depletion-load devices that function as resistors and compensate for leakage current.

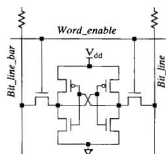


FIGURE 8-12 Transistor-level SRAM cell.

the contents of a cell. Data are written to the cell by precharging *Bit_line* and *Bit_line_bar* to complementary values, and then strobing *Word_enable*. This forces the inverters to have the values imposed by the bit lines. To understand how reading is done, suppose that *Bit_line* and *Bit_line_bar* are both precharged to a 1. When *Word_enable* is strobed, the internal node at which a 0 is held by an inverter will provide an n-channel pull-down path for the bitline to which it is attached by a pass transistor. The differential voltage between *Bit_line* and *Bit_line_bar* can be detected by a sense amplifier and used to determine the configuration of the stored data [2]. The read operation is nondestructive, because the internal state of the stored data is not affected by the circuit activity during a read cycle.

In the following examples, we will progressively develop a series of Verilog functional models of SRAMs, beginning with a model of a simple SRAM cell, and proceeding to larger memory blocks with unidirectional and bidirectional data ports.

The basic RAM cell represented by the block diagram symbol in Figure 8-13 has active-low inputs for chip select (*CS_b*) and write enable (*WE_b*). The chip select signal is generated by a decoder that selects among multiple chips in the same system. Note the absence of a clock signal. Storage registers and register files are implemented by flip-flops, but the storage devices of RAMs are implemented as transparent latches, which support asynchronous storage and retrieval of data and minimize the time that a RAM requires service from a shared bus.

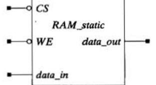


FIGURE 8-13 SRAM cell: block diagram symbol.

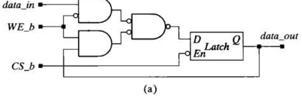
Example 8.3

The level-sensitive Verilog description, *RAM_static* models a simple RAM cell, without accounting for propagation delays. With active-low signals denoted by the suffix *_b*, level-sensitive behavior is modeled here by a single continuous assignment declaration with nested conditional operators decoding the status of *CS_b* and *WE_b*. If *CS_b* is not asserted the output is in the three-state mode (has the value Verilog logic value *z*). If *CS_b* and *WE_b* are asserted (low), the cell is in transparent mode, and *data_out* follows *data_in*; if *CS_b* is asserted and *WE_b* is de-asserted, the cell is latched. The contents of the cell can always be read, but a host processor would access *data_out* only when *WE_b* is de-asserted. The functional schematic in Figure 8-14(a) forms *RAM-static*; the simulation results presented in Figure 8-14(b), demonstrates the cell's behavior.

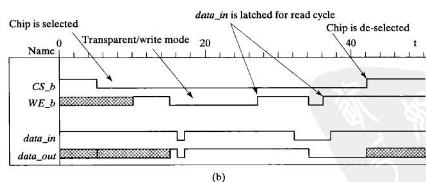
```
module RAM_static (output data_out, input data_in, CS_b, WE_b);
// Note: chip select and write are active-low

  assign data_out = (CS_b == 0) ? (WE_b == 0) ? data_in : data_out : 1'bz;
endmodule
```

The Verilog description is synthesizable, and is implemented by a single four-input lookup table (LUT) in a Xilinx FPGA.



(a)



(b)

FIGURE 8-14 SRAM cell: (a) Xilinx-generated functional schematic and (b) simulation results illustrating chip select, write, and read behavior of *RAM_static*.

End of Example 8.3

Example 8.4

The Verilog model of an SRAM cell can be modified to incorporate a single bidirectional port for use in a bus-based architecture. An additional active-low signal, *OE_b* (output enable) is added to the block diagram symbol (see Figure 8-15) and controls the datapaths through the three-state I/O buffers. The datapath is reduced from two signal ports to one port, which renders a great savings of package pins and total area if the data port is a wide vector. The structure of the model is shown in Figure 8-16, where the latch is implemented by a mux with feedback. The data paths for a write operation are shown. Output enable (*OE_b*) is asserted (low) during a read operation, and write enable (*WE_b*) is asserted (low) during a write operation. If *WE_b* is asserted and *OE_b* is not, the value at *data* is transparent through *latch_out* and is held when *WE_b* is de-asserted (i.e., written to the cell). Conversely, when *WE_b* is not asserted and *OE_b* is asserted, the content of the cell can be read through *data*, as shown by the data paths in Figure 8-17.

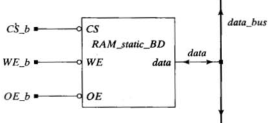


FIGURE 8-15 SRAM cell: block diagram symbol with a bidirectional data port interface to a shared bus.

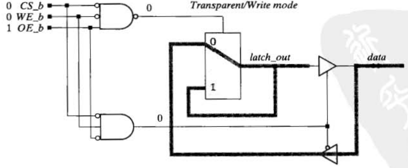


FIGURE 8-16 SRAM cell with bidirectional data port: configured to write external data through a bidirectional data port to the internal cell ($CS_b = 0, WE_b = 0, OE_b = 1$).

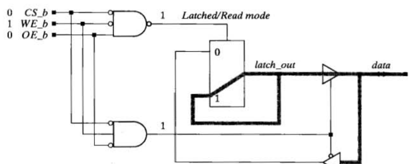


FIGURE 8-17 SRAM cell with bidirectional data port: configured to read the cell contents via the bidirectional data port ($CD_b = 0, WE_b = 1, OE_b = 0$).

The Verilog model[6] of the RAM cell with bidirectional data port, *RAM_static_BD*, is given below.

```
module RAM_static_BD (inout data, input CS_b, OE_b, WE_b);
 // Note: the data port is bi-directional
 // Note: chip select, output enable, and write enable are active-low

 wire latch_out = ((CS_b == 0) && (WE_b == 0) && (OE_b == 1)) ? data: latch_out;

 assign data = ((CS_b == 0) && (WE_b == 1) && (OE_b == 0)) ? latch_out : 1'bz;
endmodule
```

Two additional modes are possible. With *CS_b* asserted (low), the control lines could be $WE_b = 0, OE_b = 0$, and $WE_b = 1, OE_b = 1$. The configurations that result are shown in Figure 8-18. The cell is latched in both cases; its contents are not affected by the external data path, and *latch_out* does not affect *data*. The contents of the cell are not available at *data*.

The functional schematic of *RAM_static_BD*, created by the Xilinx ISE synthesis tool,[7] is shown in Figure 8-19. The schematic consists of a latch with additional logic to steer the I/O datapaths through a bidirectional data port. The synthesized and implemented circuit has *data* mapped to an I/O block (IOB) configured for bidirectional operation.

The interface between *RAM_static_BD* and a bidirectional shared bus is illustrated in Figure 8-20, and the structure of the testbench for verifying *RAM_static_BD* is shown in Figure 8-21. A separately declared register variable, *bus_driver*, drives the bidirectional bus and sends data to *RAM_static_BD*. The Verilog testbench, *test_RAM_static_BD*, uses a continuous assignment to assign the value of *bus_driver* to *data_bus* if *OE_b* is asserted during a write operation, and to disconnect *bus_driver*

[6]Continuous assignments are used here to illustrate another style for modeling level-sensitive behavior. The default type of the target of the assignment is a ***wire***. (Some tools might require an explicit declaration of type.)
[7]ISE is the Xilinx "Integrated Synthesis Environment," a tool for HDL-driven design entry and synthesis.

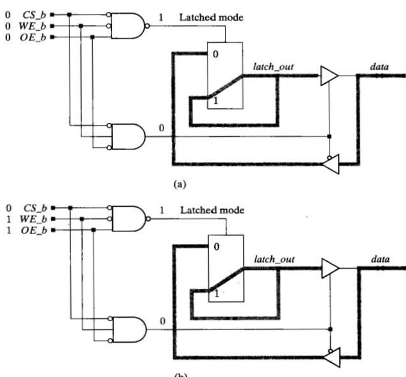


FIGURE 8-18 SRAM cell with bidirectional data port: configured for latched data and not reading or writing, (a) with ($CS_b = 0, WE_b = 0, OE_b = 0$) and (b) with ($CS_b = 0, WE_b = 1, OE_b = 1$).

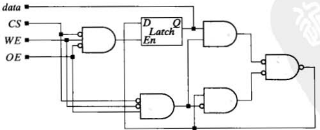


FIGURE 8-19 SRAM cell with bidirectional data port: preoptimization functional schematic created by Xilinx ISE tools.

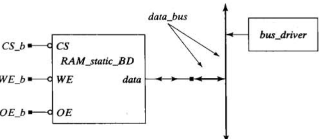


FIGURE 8-20 SRAM interface to a bidirectional data port.

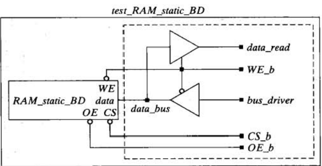


FIGURE 8-21 Testbench structure for SRAM cell with bidirectional data port.

otherwise. Note that *data_bus* has two drivers, *bus_driver* from the testbench, and *data*, the value driven through the bidirectional port of *RAM_static_BD*. The assignments from *bus_driver* to *data_bus* must be synchronized by *WE_b* and *OE_b* to avoid bus contention (i.e., so that the bus has only one driver at a time). The bidirectional nature of the testbench is illustrated in Figure 8-21, which shows *RAM_static_BD* instantiated within the testbench, *test_RAM_static_BD*. The signals *OE_b*, *WE_b*, and *CS_b* are declared as register variables in the testbench.

```
module test_RAM_static_BD ();
 // Demonstrate write / read capability.
  reg bus_driver;
  reg CS_b, WE_b, OE_b;

  wire data_bus = ((WE_b == 0) && (OE_b == 1)) ? bus_driver : 1'bz;

  RAM_static_BD M1 (data_bus, CS_b, OE_b, WE_b);
```

```
  initial #4500 $finish;
  initial begin
    CS_b = 1; bus_driver = 1; OE_b = 1;
    #500 CS_b = 0;
    #500 WE_b = 0;
    #100 bus_driver = 0;
    #100 bus_driver = 1;
    #300 WE_b = 1; #200 bus_driver = 0;
    #300 OE_b = 0; #200 OE_b = 1;
    #200 OE_b = 0; #300 OE_b = 1; WE_b = 0;
    #200 WE_b = 1; #200 OE_b = 0; #200 OE_b = 1;
    #500 CS_b = 1;
    #500 bus_driver = 0;
  end

  initial begin
    #3600 WE_b = 1; OE_b = 1;
    #200 WE_b = 0; OE_b = 0;
  end
endmodule
```

The simulation results in Figure 8-22 show a sequence of values for *CS_b*, *WE_b*, and *OE_b* to demonstrate the modes of operation of *RAM_static_BD*. In the transparent mode, with *WE_b* = 0 and *OE_b* = 1, the value of data is determined by *bus_driver* and *latch_out* is the same as *data*; when *WE_b* de-asserts, the value of data is latched (i.e., data are written to the cell).[8] When *OE_b* is asserted, with *WE_b*

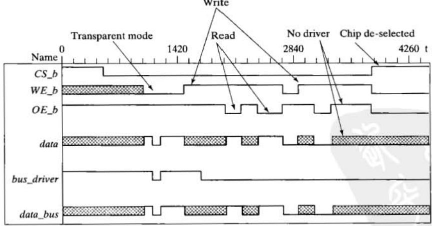


FIGURE 8-22 SRAM cell with bidirectional data port: simulation results.

[8]The output of the cell can also be latched by de-asserting *CS*, but this is not the ordinary way to end a write cycle.

de-asserted, the value of *latch_out* appears at *data* and at *data_bus*. When *OE_b* and *WE_b* are simultaneously asserted or de-asserted, the bus is not driven and could be used by another client.

End of Example 8.4

Large SRAMs cannot be implemented practically as a simple array structure for two important reasons. Large SRAMs require wide input decoders, and the footprints of long rectangular arrays might not be as convenient as square arrays for physical layout in silicon. As an alternative, large SRAMs arrays are reorganized into nearly rectangular block structures using two levels of decoding.

Example 8.5

A 32K $\times$ 8 SRAM can be organized in the structure shown in Figure 8-23, where the array has been partitioned into 8 blocks of size 512 $\times$ 64. A 32K memory requires a 15-bit address. The lower 6 bits of the address are passed to a bank of 8 muxes, each having a 64-bit wide datapath. The 6-bit address selects 1 bit from each datapath to form an 8-bit output word, *Data_Out*. These same 6 bits steer *Data_In* to 1 of 64 input lines connected to each of the 8 memory blocks. The upper 9 bits of the address are decoded by combinational logic to select one 64-bit word in each of the 8 blocks. In this reorganized structure, the address decoders have a practical size because they decode fewer outcomes, and the overall structure is nearly square, having a height of 512 cells and a width of 512 cells.

End of Example 8.5

Example 8.6

The alternative architecture shown in the block diagram in Figure 8-24 for a large SRAM has a bidirectional data port, where the column decoder, row decoder, and column I/O circuitry are represented by functional blocks adjacent to a 128 $\times$ 128 array of memory cells holding 2048 eight-bit words. The upper 7 bits of the address word decode the 128 rows of the array, and the lower 4 bits of the address decode the 16 columns of words. The three-state devices that gate the bidirectional datapaths are not shown, but are contained in the column I/O circuitry. The Verilog model of the SRAM, *RAM*_2048_8 is based on the organization of the data cells shown in Figure 8-25, where the address organization leads naturally to a row-by-row

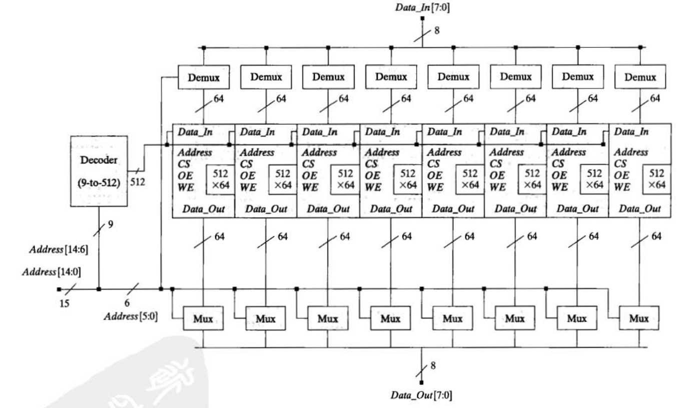


FIGURE 8-23 A 32K SRAM organized partitioned into 512 × 64 blocks with two levels of decoding.

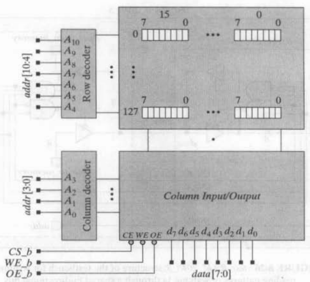


FIGURE 8-24 A 16K SRAM partitioned into 128 × 128 cells.

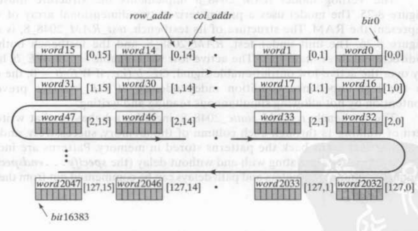


FIGURE 8-25 Organization of data words in a 16K SRAM.

sequential access, beginning at the upper rightmost cell and proceeding ultimately to the lower leftmost cell. The model will include timing parameters, describing the propagation delays of the device, and timing checks to detect violations of operational constraints during simulation.

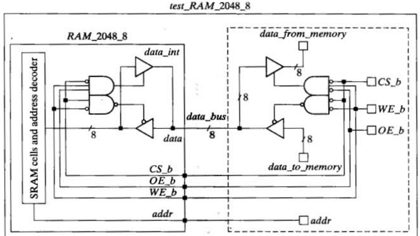


FIGURE 8-26 *test_RAM_2048_8*: structure of the testbench for writing and reading patterns of walking 1s through a shared bi-directional bus.

The Verilog model *RAM*_2048_8 implements the structure illustrated in Figure 8-25. The model uses a parameterized two-dimensional array of words to represent the RAM. The structure of its testbench, *test_RAM*_2048_8, is shown in Figure 8-26. The unit under test, *RAM*_2048_8, and the testbench both include bidirectional three-state I/O.[9] The active-low write-enable signal, *WE_b*, has priority over the active-low output-enable signal, *OE_b* (i.e., if $WE_b = 0$, the output is in the high-impedance condition independently of *OE_b*). This prevents bus contention by not allowing simultaneous reading and writing.

The testbench, *t_RAM_static*_2048_8, includes a behavior that writes a pattern of walking 1s through each column of the memory, successively, and another behavior that reads back the patterns stored in memory. Patterns are included in the testbench for simulating with and without delay (the ***specify . . . endspecify*** block containing timing parameters and path delays can be commented out from the code).[10]

[9]Note that we have simplified the schematic by showing a single three-state buffer instead of an actual configuration having a buffer on each bit line of each bus.

[10]The delay values used are for illustration and do not represent the fastest devices that are available with the most advanced technology. They should be replaced by the parameters of a specific part.

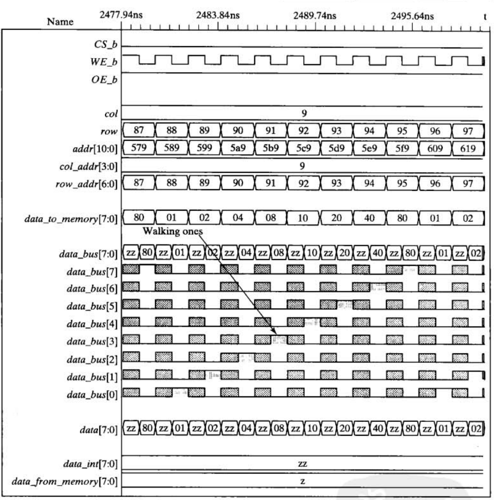


FIGURE 8-27 *RAM_2048_8*: simulation results for writing a walking 1s pattern to memory with zero delay.

The simulation results shown in Figure 8-27 show the patterns written in column 9, starting at row 88, for zero-delay simulation. The three-state action of the bus and the bidirectional datapath causes *data_bus* and *data* to have the value zz_H in the displayed waveforms when *WE_b* is 1. The results in Figure 8-28 show the patterns read

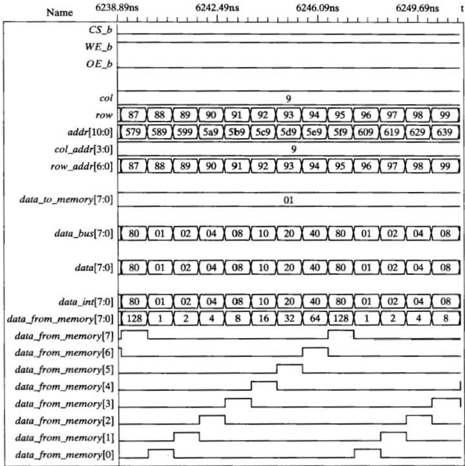


FIGURE 8-28 *RAM_2048_8*: simulation results for reading back a walking 1s pattern from memory with zero delay.

back (later) from the same locations. When nonzero propagation delays are included in the model, the simulation results in Figure 8-29 are obtained for writing to memory. When the inputs generated by the testbench have *WE_b* and *OE_b* simultaneously high, the data_bus is in its three-state mode. The waveforms in Figure 8-30 are obtained by reading data from memory. The testbench includes an

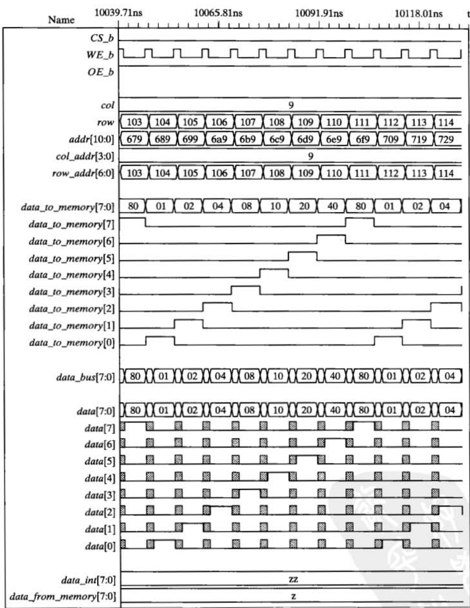


FIGURE 8-29 *RAM*_2048_8: simulation results for writing to memory a walking 1s pattern with nonzero propagation delay.

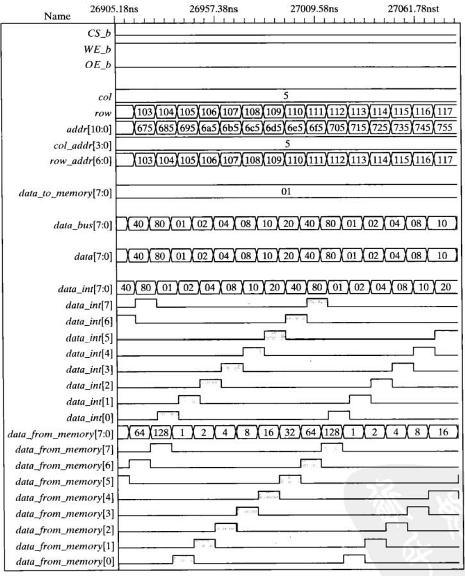


FIGURE 8-30 *RAM_2048_8*: simulation results for reading data from memory.

optional signal, *write_probe*, which reports the value that is stored in memory at the rising edge of *WE_b* and is used to verify the latching activity of the model.

```
'timescale 1ns / 10ps
module RAM_2048_8 #(parameter
```

```
   word_size = 8,
   addr_size = 11,
   mem_depth = 128,
   mem_width = 16,
   col_addr_size = 4,
   row_addr_size = 7,
   Hi_Z_pattern = {word_size{1'bz}}
)(
   inout [word_size -1: 0] data,
   input [addr_size -1: 0] addr,
   input CS_b, WE_b, OE_b
);
   reg [word_size -1 : 0] data_int;
   reg [word_size -1 : 0] RAM [0: mem_depth -1] [0: mem_width -1]      // 128 rows,
                                                                          16 columns
   wire [col_addr_size -1: 0] col_addr = addr[col_addr_size -1: 0];
   wire [row_addr_size -1: 0] row_addr = addr[addr_size -1: col_addr_size];

   assign data = ((CS_b == 0) && (WE_b == 1) && (OE_b == 0))
    ? data_int: Hi_Z_pattern;

   always @ (data, col_addr, row_addr, CS_b, WE_b, OE_b)
    begin
     data_int = Hi_Z_pattern;
     if ((CS_b == 0) && (WE_b == 0)) RAM [row_addr][col_addr] = data;

     else if ((CS_b == 0) && (WE_b == 1) && (OE_b == 0))  // Read from memory
       data_int = RAM [row_addr][col_addr];
    end

/* Comment out the model for a zero delay functional test.
// Also adjust stop time in test bench

   specify
   // Parameters for the read cycle
    specparam t_RC = 10;  // Read cycle time
    specparam t_AA = 8;   // Address access time
    specparam t_ACS = 8;  // Chip select access time
    specparam t_CLZ = 2;  // Chip select to output in low-z
    specparam t_OE = 4;   // Output enable to output valid
    specparam t_OLZ = 0;  // Output enable to output in low-z
    specparam t_CHZ = 4;  // Chip de-select to output in hi-z
    specparam t_OHZ = 3.5;// Output disable to output in hi-z
    specparam t_OH = 2;   // Output hold from address change
   // Parameters for the write cycle
    specparam t_WC = 7;   // Write cycle time
    specparam t_CW = 5;   // Chip select to end of write
    specparam t_AW = 5;
    specparam t_AS = 0;   // Address setup time
    specparam t_WP = 5;   // Write pulse width
    specparam t_WR = 0;   // Write recovery time
    specparam t_WHZ = 3;  // Write enable to output in hi-z
```

```
  specparam t_DW = 3.5;// Data set up time
  specparam t_DH = 0;   // Data hold time
  specparam t_OW = 10; // Output active from end of write

 //Module path timing specifications
  (addr *> data) = t_AA;
  (CS_b *> data) = (t_ACS, t_ACS, t_CHZ);
  (OE_b *> data) = (t_OE, t_OE, t_OHZ);

 // Timing checks (Note use of conditioned events for the address setup,
 // depending on whether the write is controlled by the WE_b or by CS_b.
 //Width of write/read cycle
  $width (negedge addr, t_WC);
 //Address valid to end of write
  $setup (addr, posedge WE_b &&& CS_b == 0, t_AW);
  $setup (addr, posedge CS_b &&& WE_b == 0, t_AW);
 //Address setup before write enabled
  $setup (addr, negedge WE_b &&& CS_b == 0, t_AS);
  $setup (addr, negedge CS_b &&& WE_b == 0, t_AS);
 //Width of write pulse
  $width (negedge WE_b, t_WP);
 //Data valid to end of write
  $setup (data, posedge WE_b &&& CS_b == 0, t_DW);
  $setup (data, posedge CS_b &&& WE_b == 0, t_DW);
 //Data hold from end of write
  $hold (data, posedge WE_b &&& CS_b == 0, t_DH);
  $hold (data, posedge CS_b &&& WE_b == 0, t_DH);
 //Chip sel to end of write
  $setup (CS_b, posedge WE_b &&& CS_b == 0, t_CW);
  $width (negedge CS_b &&& WE_b == 0, t_CW);
 endspecify
*/
endmodule

module test_RAM_2048_8 ();
 parameter word_size = 8;
 parameter addr_size = 11;
 parameter mem_depth = 128;
 parameter num_col = 16;
 parameter col_addr_size = 4;
 parameter row_addr_size = 7;
 parameter initial_pattern = 8'b0000_0001;
 parameter Hi_Z_pattern = {word_size{1'bz}};

 reg [word_size -1 : 0] data_to_memory;
 reg CS_b, WE_b, OE_b;

 integer col, row;
 wire [col_addr_size -1:0] col_addr = col;
 wire [row_addr_size -1:0] row_addr = row;
 wire [addr_size -1:0] addr = {row_addr, col_addr};
```

```
  parameter t_WPC = 8;
  parameter t_RPC = 12;
  parameter latency_Zero_Delay = 5000;
  parameter latency_Non_Zero_Delay = 18000;
  parameter stop_time = 7200;   // For zero-delay simulation
//parameter stop_time = 45000; // For non-zero delay simulation

// Three-state, bi-directional I/O bus
  wire [word_size -1: 0] data_bus = ((CS_b == 0) && (WE_b == 0) && (OE_b == 1))
    ? data_to_memory: Hi_Z_pattern;

  wire [word_size -1: 0] data_from_memory = ((CS_b == 0) && (WE_b == 1) &&
    (OE_b == 0))
    ? data_bus: Hi_Z_pattern;

  RAM_2048_8 M1 (data_bus, addr, CS_b, WE_b, OE_b);       // UUT

  initial #stop_time $finish;
///*
// Zero delay test: Write walking ones to memory
  initial begin
    CS_b = 0;
    WE_b = 0;
    OE_b = 1;
    for (col= 0; col <= num_col-1; col = col +1) begin
      data_to_memory = initial_pattern;
      for (row = 0; row <= mem_depth-1; row = row + 1) begin
        #1 WE_b = 0;
        #1 WE_b = 1;

        data_to_memory ={data_to_memory[word_size-2:0], data_to_memory
          [word_size -1]};
      end
    end
  end
//*/

///* // Zero delay test: Read back walking ones from memory
  initial begin
    #latency_Zero_Delay;
    CS_b = 0;
    WE_b = 1;
    OE_b = 0;
    for (col= 0; col <= num_col-1; col = col +1) begin
      for (row = 0; row <= mem_depth-1; row = row + 1) begin
        #1;
      end
    end
  end
//*/
// Non-Zero delay test: Write walking ones to memory
```

```
// Writing controlled by WE_b
/*
 initial begin
   CS_b = 0;
   WE_b = 1;
   OE_b = 1;
   for (col= 0; col <= num_col-1; col = col +1) begin
     data_to_memory = initial_pattern;
     for (row = 0; row <= mem_depth-1; row = row + 1) begin
       #(t_WPC/8) WE_b = 0;
       #(t_WPC/4);
       #(t_WPC/2) WE_b = 1;
       data_to_memory ={data_to_memory[word_size-2: 0], data_to_memory
         [word_size -1]};
       #(t_WPC/8);
     end
   end
 end

// Non-Zero delay test: Read back walking ones from memory
 initial begin
   #latency_Non_Zero_Delay;
   CS_b = 0;
   WE_b = 1;
   OE_b = 0;
   for (col= 0; col <= num_col-1; col = col +1) begin
     for (row = 0; row <= mem_depth-1; row = row + 1) begin
       #t_RPC;
     end
   end
 end
*/
// Testbench probe to monitor write activity
 reg [word_size -1: 0] write_probe;

 always @ (posedge M1.WE_b) write_probe = M1.RAM[row_addr][col_addr];
endmodule
```

End of Example 8.6

The timing parameters incorporated in the model for *RAM*_2048_8 govern the transitions of the output waveforms in response to changes of the input waveforms and establish operational constraints that must be satisfied for correct operation of the device. For example, if the address is not stable when *CS_b* and *WE_b* are low, multiple memory cells can be affected while the device is in the transparent/write mode. The address access time is a key parameter that dictates the rate at which the memory can be read. Table 8-3 lists parameters describing the write cycle of a static RAM, and Table 8-4 describes the read cycle.

TABLE 8-3 Parameters for the write cycle of a static RAM.

	SRAM Write-Cycle Parameters
t_WC	**Write cycle time**: Specifies the minimum period for successive writing of data to memory.
t_CW	**Chip select to end of write**: Specifies the minimum interval between the falling edge of *CS_b* and the rising edge of *WE_b*.
t_AW	**Address valid to end of write**: Specifies the minimum interval between a change in the address and the end of write (the rising edge of *WE_b*).
t_AS	**Address setup time before write**: Specifies the width of the interval over which the address must be stable prior to the falling edge of *WE_b*.
t_WP	**Write pulse width**: Specifies the minimum width of the write pulse.
t_WR	**Write recovery time**: Specifies the minimum interval between the rising edge of *WE_b* and the end of the write cycle.
t_WHZ	**Write enable to output in high-z**: Specifies the minimum interval between the falling edge of *WE_b* and the output entering the high-impedance state.
t_DW	**Data setup time**: Specifies the minimum width of the interval over which the data must be stable prior to the rising edge of *WE_b*.
t_DH	**Data hold time after end of write**: Specifies the minimum interval that the data must be stable after the rising edge of *WE_b*.
t_OW	**Output active from end of write**: Specifies the earliest time that the output is available after the rising edge of *WE_b*.

The timing parameters of a write cycle are illustrated in Figure 8-31. Two cases must be considered: (1) the operation controlled by *WE_b* with $CS_b = 0$ (the device is selected) and $OE_b = 1$ (the read cycle is not active) and (2) the operation controlled by *CS_b* with $WE_b = 0$ (write is enabled) and $CS_b = 0$.

In the former case (shown in Figure 8-31(a)), the address must be stable and the chip must be selected before the falling edge of *WE_b*. The write cycle occurs over an interval of width t_{WC}, which includes the times at which the address lines may be changed. The address setup time, t_{AS}, establishes the minimum time between the stable address and the falling edge of *WE_b*. This constraint ensures that the address-decoding circuitry is stable before the write is attempted. The enable input of a transparent latch must satisfy a minimum pulsewidth constraint (t_{WP}); similarly, the time from chip select to the end of the write cycle (t_{CW}) must also satisfy a pulsewidth constraint (t_{CW}). While *WE_b* is low the device is in the transparent mode and the three-state device driving *data_int* is in the high-impedance state. The device enters this state, with a delay specified by t_{WHZ}, when *WE_b* is asserted. The data to be written to the SRAM must satisfy a setup time constraint[11] (t_{DW}) and a hold time constraint (t_{DH}) relative to the rising edge of *WE_b*. *Note*: Figure 8-31(a) is drawn to illustrate the rising edge of *CS_b* occurring after the rising edge of *WE_b*.[12] The address must be stable for the write recovery time interval (t_{WR}), after

[11]We will consider timing constraints in more detail in Chapter 11.

[12]If the rising edge of *CS* occurs before the rising edge of *WE* the timing constraints must be applied relative to the rising edge of *CS*.

TABLE 8-4 Parameters for the read cycle of a static RAM.

	SRAM Read-Cycle Parameters
t_RC	**Read-cycle time**: Specifies the minimum period for successive reading of data from memory.
t_AA	**Address access time**: A key performance parameter specifying the minimum interval between a change in the address and the availability of valid data retrieved from memory.
t_ACS	**Chip select access time**: Specifies the minimum interval between assertion of chip select and the availability of valid data from memory, assuming that $OE_b = 0$ and $WE_b = 1$ before $CS_b = 0$.
t_CLZ	**Chip select low z**: Specifies the minimum interval between assertion of chip select and the output leaving the high-impedance state.
t_OE	**Output enable to output valid**: Specifies the minimum interval between the falling edge of *OE_b* and the availability of valid data from memory.
t_OLZ	**Output enable to output in low Z**: Specifies the minimum interval between the falling edge of *OE_b* and the output leaving the high-impedance state.
t_CHZ	**Chip deselect to output in high Z**: Specifies the minimum interval between the rising edge of *OE_b* and the output entering the high-impedance state.
t_OHZ	**Output disable to output in high Z**: Specifies the minimum interval between the rising edge of *OE_b* and the output entering the high-impedance state.
t_OH	**Output hold from address change: Specifies the minimum interval that the output remains valid after a change in the address.**

the rising edge of *WE_b*, and the bus becomes available after an interval (t_{OW}) expires from the rising edge of *WE_b*. The interval from the onset of a stable address to the end of the write cycle is represented by the parameter t_{AW}.

When *WE_b* is low before the falling edge of *CS_b*, and rises after the rising edge of *CS_b*, the SRAM is controlled by *CS_b* and is characterized by the waveforms in Figure 8-31(b). In this case, the setup and hold time constraints for the data on the bus are relative to the rising (latching) edge of *CS_b*.

The two modes of the read cycle are illustrated in Figure 8-32. In Figure 8-32(a), the data is determined by the address (with $CS_b = 0$ and $WE_b = 1$, and is valid t_{AA} time units after the address is stable. In Figure 8-32(b), the data becomes valid after t_{ACS} time units from the falling edge of *CS_b*.

8.2.10 Ferroelectric Nonvolatile Memory

Ferroelectric materials are so named because their electrical characteristics resemble those of ferromagnetic materials. Despite the suggestion implied by their name, ferroelectric materials have nothing to do with ferromagnetics. Their similarity is primarily in the fact that certain ferroelectric materials can exhibit a significant hysteresis effect, but it is not associated with magnetic properties. Instead, the hysteresis effect in a ferroelectric is due to the so-called spontaneous electrical polarization of a ferroelectric

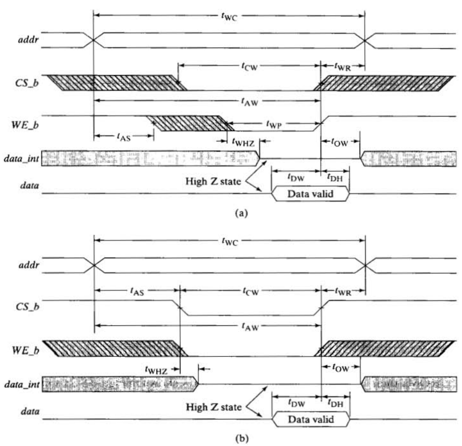


FIGURE 8-31 SRAM timing: (a) write cycle controlled by WE_b, with $CS_b = 0$ and $OE_b = 1$, and (b) write cycle controlled by CS_b, with $WE_b = 0$ and $OE_b = 1$.

material under the influence of an applied voltage. When power is removed the residual polarization behaves like a bistable memory device. Ferroelectric memories hold the promise of replacing other nonvolatile memories, such as EEPROMs in applications that require short programming time, low power consumption, and low fatigue. Contactless smart cards, digital cameras, and utility meters are considered to be appropriate applications for this technology. EEPROMs and flash memories are also nonvolatile and have lower power to read data than ferroelectrics. Ferroelectric memories can also be embedded with other devices. This technology is expected to mature to have competitive circuit densities compared to other alternatives. See Sheilholeslami and Gulak [3] for a survey of circuits exploiting ferroelectric technology.

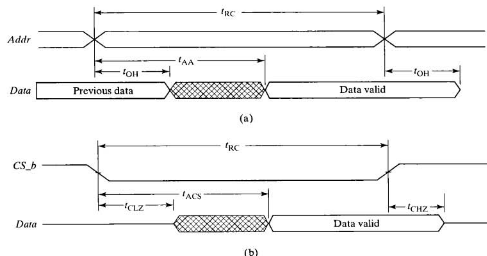


FIGURE 8-32 SRAM timing: (a) read cycle controlled by address changes, with $CS_b = 0$, $OE_b = 0$, and $WE_b = 1$, and (b) read cycle controlled by changes in *CS_b*, with $OE_b = 0$, and $WE_b = 1$.

8.3 Programmable Logic Array (PLA)

PLAs were developed for integrating large two-level combinational logic circuits. Like ROMs, their architecture consists of two arrays, shown in Figure 8-33. One array implements the *AND* operation that forms a product term (i.e., a Boolean cube, possibly a minterm) and another array implements the *OR* operation that forms a SOP terms. A PLA implements a two-level Boolean function in SOP form.

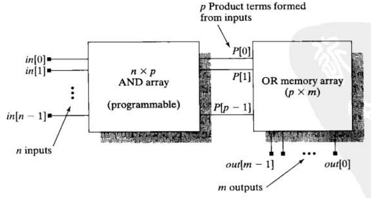


FIGURE 8-33 *AND-OR* plane structure of a PLA.

Unlike ROMs, *both arrays of a PLA are programmable* (mask-programmable or one-time field-programmable). However, *the AND plane does not implement a full decoder*, but instead forms a limited number of product terms. The programmable *OR*-plane forms expressions by *OR*-ing together product terms (cubes). An $n \times p \times m$ PLA has n inputs, p product terms (outputs of the *AND* plane), and m output expressions (from the *OR* plane). A $16 \times 48 \times 8$ PLA has 48 product terms. A 16-input ROM would have $2^{16} = 65{,}536$ input patterns decoded as minterms and available to form the outputs. A PLA would have 8 outputs formed from the 48 product terms (not necessarily minterms).

A PLA implements general product terms, not just minterms or maxterms. Because it has limited *AND*-plane resources, minimal SOP forms must be found so that device resources might accommodate an application's requirements for product terms. PLA minimization algorithms led to development of widely used synthesis algorithms having general application to ASICs [4].

The circuit structure of a PLA implemented in nMOS technology is shown in Figure 8-34. The *AND-OR* plane structure shown implements *NOR-NOR* logic, which reverts to equivalent *AND-OR* logic with inverted inputs and three-stated inverters at

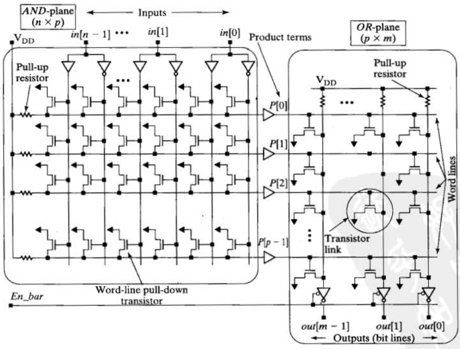


FIGURE 8-34 Circuit structure of a PLA.

the outputs. Each input is available as a literal in complemented and uncomplemented forms. A programmable link in the *AND* plane determines whether the associated input literal (or its complement) is connected to a buffered word line.

Programming determines whether inputs have a link to word lines and whether word lines connect to the output lines. A word line may be linked to an input or its complement, but not both. Each word line is connected to a pull-up resistor (active device). The aggregate of linked input literals and complements of input literals forms a Boolean cube at the word line to which the links are attached. Unconnected inputs have no effect on a word line. In the absence of an asserted and connected input literal (or its complement), a word line is pulled up. In the absence of an asserted (high) level on its linked word lines, a column line is pulled high. An asserted input turns on a connected link transistor in the *AND* plane and pulls the word line to ground by overriding its pull-up resistor. A column line is asserted (high) if all of its connected word lines are de-asserted (low). An asserted word line turns on a connected link transistor in the *OR* plane, causing its connected word line to be pulled down. A column line is low if any of its connected word lines is asserted (high). If any word line is asserted (high), a connected column line is pulled down. A column line is asserted (high) only if all of its connected word lines are de-asserted (low).

To see that the circuit shown in Figure 8-35 exhibits wired-*AND* logic at its word lines, note that

$$W1 = A'B'$$

$$W1' = (A + B)'$$

$$W2 = C'D'$$

$$W2' = (C + D)'$$

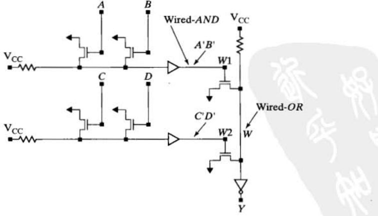


FIGURE 8-35 Wired-*OR* logic of a PLA.

Similarly, the column lines exhibit wired-*OR* behavior, where W is low if $W1$ or $W2$ is high; otherwise W is high (pull-up):

$$W' = W1 + W2$$
$$W = (W1 + W2)'$$
$$Y = W' = W1 + W2 = A'B' + C'D'$$
$$Y = (A + B)' + (C + D)'$$

The overall structure is that of *NOR-NOR* logic, with

$$Y' = \lfloor (A + B)' + (C + D)' \rfloor'$$

The equivalent circuit is shown in Figure 8-36(a) and an equivalent *OR-AND* structure is shown in Figure 8-36(b).

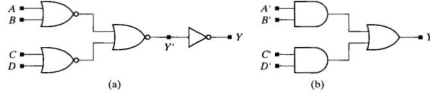


FIGURE 8-36 Equivalent circuit structures for PLA logic: (a) *NOR-NOR* logic and (b) *OR-AND* logic with inverted inputs.

8.3.1 PLA Minimization

The area of a PLA depends primarily on the number of word lines (i.e., distinct product terms), so it is advantageous to find ways to reduce the number of product terms by sharing logic as much as possible. One approach would be to use Karnaugh maps or other minimization methods to reduce each Boolean expression. However, independent minimization of individual Boolean functions does not necessarily produce an optimal PLA implementation. Minimization of a set of Boolean functions, as an aggregate, can exploit don't-cares and opportunities to share logic because a product term that is generated to form one output expression can be used in another output expression that uses the same term. Alternatively, a common factor in a product of sums form can be shared by multiple functions that have the same factor.

Example 8.7

Consider the three Boolean functions shown below, with their K-maps shown in Figure 8-37. Before minimization, the implementation would require 13 product terms (word lines) to support the cubes of the three functions.

$$f_1(a, b, c, d) = \sum m(1, 6, 7, 9, 13, 14, 15)$$
$$f_2(a, b, c, d) = \sum m(6, 7, 8, 9, 13, 14, 15)$$
$$f_3(a, b, c, d) = \sum m(1, 2, 3, 9, 10, 11, 12, 13, 14, 15)$$

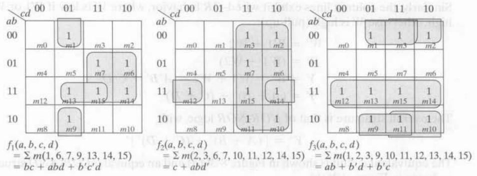


FIGURE 8-37 Individual Karnaugh map minimization of three Boolean functions.

After each function is individually minimized, the total number of cubes is 8, a savings of nearly 40%. To minimize the functions as an aggregate, a task easily done by modern synthesis tools, we re-cover the functions and identify common cubes by considering pairwise and threewise intersections, as shown in Figure 8-38. The final result needs only five word lines, having eliminated an additional four-word lines.

End of Example 8.7

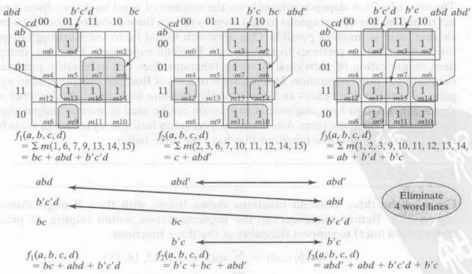


FIGURE 8-38 Karnaugh map minimization of a set of three Boolean functions.

Systematic manual minimization of multiple output functions is feasible for up to three functions, with a maximum of four inputs [5]. Otherwise, a computer-based approach is needed (e.g., espresso [4] and MIS-II [6–8]).

The tabular format shown in Figure 8-39 can be used to specify the functionality of a PLA. Table rows correspond to PLA rows (word lines). Table columns list inputs and functions indicating whether an input is in a cube, and whether a cube is in a function. Inputs are coded as 1 (care-on), 0 (care-off), and (don't-care). Outputs are coded as 1 (contains the word line) or 0 (does not contain the word line).

	a	*b*	*c*	*d*	f_1	f_2	f_3
abd	1	1	–	1	1	0	1
b'c'd	–	0	0	1	1	0	1
b'c	–	0	1	–	0	1	1
bc	–	1	1	–	1	1	0
abd'	1	1	–	0	0	1	1

FIGURE 8-39 Tabular format for specifying the structure of a PLA.

ROMs require canonical data (i.e., a complete truth table), but PLAs require only minimum SOP Boolean forms. The cubes in a minimized PLA table may cover multiple minterms, and a given input vector may assert multiple output functions. For a given input vector, the cubes are formed by *AND*-ing the complemented and uncomplemented literals in a row; the outputs are determined by column-wise *OR*-ing the cube (word line) entries having a 1. A simplified representation of a PLA is shown in Figure 8-40. The filled circles indicate whether a literal or its complement is used in a cube and whether a cube is used in an expression.

8.3.2 PLA Modeling

An application for a PLA must be compatible with the limited number of product terms (word lines) that can fit within the device. PLAs are used to implement the next-state and output-forming logic of large state machines that control more complex sequential machines, such as computers. PLAs are a more attractive implementation than ROMs for large state machines because the area of a PLA can be minimized and tailored to an application.

Verilog includes a set of system tasks for modeling multiinput, multioutput PLAs. PLAs implement two-level combinational logic by an array structure of *AND, NAND, OR*, and *NOR* logic array planes. The "personality" file, or matrix, of the PLA specifies the physical connections of transistors forming the product of input terms (cubes) and the sums of those products to form the Boolean expressions of the outputs. See "Selected System Tasks and Functions" at the companion Web site. Built-in system tasks describe synchronous and asynchronous arrays. The outputs of the asynchronous arrays are updated whenever an input signal changes value or whenever the personality matrix of the PLA

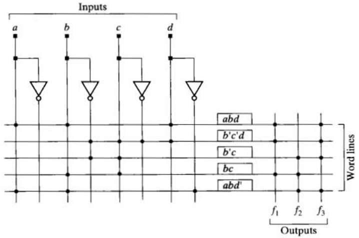


FIGURE 8-40 Simplified representation of a PLA.

changes during simulation. The synchronous types are updated when evaluated in a synchronous behavior. Both forms update their outputs with zero delay.

The personality matrix of a PLA specifies the cubes that form the inputs to the PLA and the expressions forming the outputs of the PLA. The data describing the personality is stored in a memory whose width accommodates the inputs and outputs of the PLA and whose depth accommodates the number of outputs.

There are two ways to load data into the personality matrix: (1) using the ***$readmemb*** task, read the data from a file and (2) load the data directly with procedural assignment statements. Both methods can be used at any time during a simulation to reconfigure the PLA dynamically.

Two formats may be used to describe the contents of an array: *array* and *plane*. The array format stores either a 1 or 0 in memory to indicate whether a given input is in a cube, and whether a given cube is in an output.

Example 8.8

The statements below illustrate calls to Verilog's built-in PLA system tasks describing synchronous and asynchronous arrays and planes:

```
$async$and$array   (PLA_mem, {in0, in1, in2, in3, in4, in5, in6, in7}, {out0,
                   out1, out2});
$sync$or$plane     (PLA_mem, {in0, in1, in2, in3, in4, in5, in6, in7}, {out0,
                   out1, out2});
$async$and$array   (PLA_mem, {in0, in1, in2, in3, in4, in5, in6, in7}, {out0,
                   out1, out2});
$async$and$array   (PLA_mem, {in0, in1, in2, in3, in4, in5, in6, in7}, {out0,
                   out1, out2});
```

For example, the array format shown below indicates that the cube *in*1 & *in*2 & *in*3 is formed and used in *out*1, but not in *out*2. The cube *in*1 & *in*3 is used in *out*2.

*in*1	*in*2	*in*3	*out*1	*out*2
1	1	1	1	0
1	0	1	0	1

End of Example 8.8

Example 8.9

Suppose we want to implement the logic of the following Boolean equations with a PLA:

*out*0 = *in*0 & *in*1 & *in*2 & *in*3 + *in*4 & *in*5 & *in*6 & *in*7 + *in*1 & *in*3 & *in*5 & *in*7

*out*1 = *in*1 & *in*3 & *in*5 & *in*7 + *in*4 & *in*5 & *in*6 & *in*7

*out*2 = *in*0 & *in*2 & *in*4 & *in*6 + *in*4 & *in*5 & *in*6 & *in*7 + *in*1 & *in*3 & *in*5 & *in*7

The expressions use four distinct cubes:

*in*0 *in*1 *in*2 *in*3
*in*0 *in*2 *in*4 *in*6
*in*4 *in*5 *in*6 *in*7
*in*1 *in*3 *in*5 *in*7

The personality data of the PLA is shown below and is placed in a text file, *PLA_data.txt*. The data indicate the presence of a literal by a 1, and the absence of a literal by 0, listed in ascending order of the inputs. There is one row for each cube, a column for each input, and the last three columns indicate whether a row cube is present in each of the three output functions.

```
11110000 100
10101010 011
00001111 111
01010101 101
```

The Verilog model *PLA_array* describes a PLA that forms three output functions of eight Boolean input variables. The personality of the array is stored in the array of words *PLA_mem*, whose width corresponds to the width of the personality matrix and whose depth is determined by the number of Boolean expressions that

will be formed as outputs. Hence, the array of words has a width of 11 bits and a depth of three words.

```
module PLA_array (output reg out1, out2, out3, input in1, in2, in3, in4, in5, in6, in7);
 reg [0: 10] PLA_mem [0: 2];        // 3 functions of 8 variables
 initial begin
  $readmemb ("PLA_data.txt", PLA_mem);
  $async$and$array (PLA_mem, in0, in1, in2, in3, in4, in5, in6, in7}, {out0, out1, out2});
 end
endmodule
```

End of Example 8.9

The PLA in Example 8.8 is configured by the ***initial*** behavior at the beginning of a simulation. The simulator reads the file *PLA_data.txt* and loads the data into the declared memory, *PLA_mem*. Note that the inputs and outputs are declared in ascending order. When an input to the module changes value the array is evaluated to form updated values of *out0*, *out1*, and *out2*.

The array format requires that the complement of a literal be provided separately as an input if it is needed to form a cube. On the other hand, the plane format encodes the personality matrix, according to the format in Table 8-5, which was adopted from the Espresso format developed at the University of California at Berkeley [4].

Example 8.10

Suppose the logic to be implemented in a PLA is described by the following statements:

$$out0 = in0 \ \& \sim in2;$$
$$out1 = in0 \ \& \ in1 \ \& \sim in3;$$
$$out2 = \sim in0 \ \& \sim in3;$$

In the plane (Espresso) format, the personality of the PLA is described by:

4'*b*1?0?
4'*b*11?0
4'*b*0??0

TABLE 8-5 Personality matrix symbols for PLA plane format.

Table Entry	Interpretation
0	The complemented literal is used in the cube.
1	The literal is used in the cube.
x	The worst case of the input is used.
z	Don't care; the input has no significance.
?	**Same as *z*.**

PDG

The rows correspond to the outputs and are listed in descending order. A row defines the conditions of the inputs that assert that output. For example, the inputs 1000 and 1101 will both assert the first output. A Verilog description of the PLA is given below.

```
module PLA_plane (input in0, in1, in2, in3, in4, in5, in6, in7, output reg out0,
  out1, out2);
  reg                 [0: 3] PLA_mem [0: 2];
  reg                 [0: 4] a;
  reg                 [0: 3] b;
  initial begin
     $async$and$array
      (PLA_mem, {in0, in1, in2, in3, in4, in5, in6, in7}, {out0, out1, out2});
     PLA_mem [0] = 4'b1?0?;
     PLA_mem [1] = 4'b11?0;
     PLA_mem [2] = 4'b0??0;
  end
endmodule
```

End of Example 8.10

8.4 Programmable Array Logic (PAL)

PAL technology[13] emerged after PLAs, and simplified the dual-array structure by fixing the *OR* plane and allowing only the *AND* plane to be programmed. Each output is formed from a specified number of word lines, and each word line is formed from a small number of product terms. One of the more popular devices, the PAL16L8, has the structure shown in Figure 8-41. The device has 16 inputs and 8 outputs; its package has 20 pins, including power and ground. Each input is available in true or complemented form. There are eight 7-input *OR* gates connected to word lines from the *AND* plane. Each word line can be connected to any input or its complement. An eighth word line in each group controls a three-state inverter that is driven by the group's *OR* gate. Each output implements a sum of products expression from at most seven terms. The device has only 20 pins, so six of the pins are bidirectional. The *AND* gate (not shown) that is associated with each word line is permanently connected to an *OR* gate and cannot be shared with any other *OR* gate, but six of the outputs are connected to three-state inverters and can be fed back to the *AND* array to be shared with other *AND* gates, which accommodates expressions having more than seven product terms. A bidirectional pin also makes it possible for the device to implement a transparent latch by combinational feedback. PLD-based latches have application as address decoder/latches in microprocessor systems [1]. Modern PAL devices are manufactured with registered outputs and selectable output polarity.

[13]*Note*: PAL is a trademark of Applied Micro Devices (AMD).

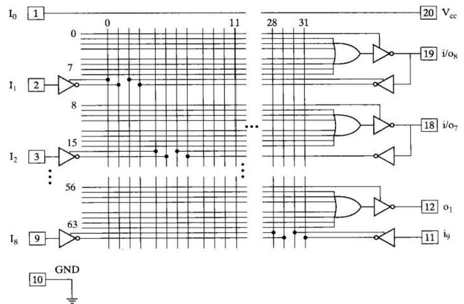


FIGURE 8-41 Circuit structure of the PAL16L8 programmable array logic device.

Early PAL devices were implemented in bipolar technology; like ROMs, they were programmed by vaporizing metal links. Contemporary devices are implemented in CMOS technology with floating-gate link transistors.

8.5 Programmability of PLDs

ROMs, PLAs, and PLDs are implemented in similar array structures. Table 8-6 compares the options that are presented for programming the devices. PLAs provide the greatest flexibility and are used for large, complex, combinational logic circuits.

TABLE 8-6 Programmability options for various PLDs.

	Programmable Block	
	***AND* Plane**	***OR* Plane**
ROM	NA	P
PLA	P	P
PAL	P	NP

NA = not applicable, P = programmable, NP = not programmable.

8.6 Complex PLDs (CPLDs)

As technology has evolved, more dense and complex devices have been developed to implement large structures (e.g., more than 1024 functions) of field-programmable combinational and sequential logic, and are referred to as complex PLDs, or CPLDs. The high-level architecture of a typical CPLD (shown in Figure 8-42) is formed as a structured array of PLD blocks that have a programmable on-chip interconnection fabric. Aside from increased performance, these architectures overcome the limitation of conventional PLDs, which have a relatively small number of inputs. CPLDs have wide inputs, but not at the expense of a dramatic (i.e., exponential) increase in area. The device area of a conventional PLD will be scaled by a factor of 2^n if its input dimension is scaled by a factor of n. An array of identical interconnected PLDs will accommodate the increased dimension of the input space too, but the cell area increases by a factor of only n, in addition to the area required by the interconnection fabric. Thus, CPLDs are distinguished by having wide fan-in *AND* gates. Large CPLDs do not connect the output of every macrocell to an output pin, but they typically have 100% connectivity between macrocells.

Each PLD block of a CPLD has a PAL-like internal structure that forms combinational logic functions of its inputs. The outputs of the macrocells in the PLDs can be programmed to route to the inputs of other logic blocks to form more complex, multilevel logic beyond the limitations of a single logic block. Some CPLDs are electrically erasable and reprogrammable (EPLD). CPLDs are suited for wide fan-in AND-OR logic structures and exploit a variety of programming technologies: SRAM/transmission gates, EPROM (floating-gate transistors), and antifuses.[14]

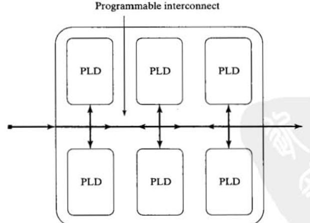


FIGURE 8-42 High-level architecture of a CPLD.

[14] Antifuses are programmable low-resistance electrical links.

8.7 Field-Programmable Gate Arrays

CPLDs are characterized by an array of PAL-like blocks of combinational logic implementing wide-input SOP expressions. They have predictable timing and a crossbar type of interconnection fabric, and are suited for low- and medium-density applications. FPGAs have a more complex and register-rich, tiled architecture of functional units, and a flexible channel-based interconnection fabric. Featuring flash-based reconfigurability, CPLDs can be reprogrammed a limited number of times, but FPGAs have no practical limit on their reconfigurability. FPGAs are suitable for medium- and high-density applications. They differ in two significant ways from CPLDs because (1) their performance is dependent on the routing that is implemented in the device for a particular application and (2) their functionality is implemented by LUTs rather than by PAL-like wide-input *AND* gates.

Mask-programmable gate arrays are fabricated in a foundry, where final layers of metal customize the wafer to the specifications of the end user. Field-programmable gated arrays are sold as fully fabricated and tested generic products. Their functionality is determined by programming done in the field by the customer and/or end user. FPGAs allow designers to turn a design into working silicon in a matter of minutes, making rapid prototyping of stand-alone and embedded systems a reality.

FPGAs are distinguished on the basis of several features: architecture, number of gates, mechanism for programming, program volatility, the granularity and robustness of a functional/logical unit, physical size (footprint), pinout, time-to-prototype, speed, power, I/O, and the availability of internal resources for connectivity and clock management [9, 10]. We will focus on the dominant technology: SRAM-based FPGAs, which lose their programming when power is removed from the part.

SRAM-based FPGAs have a fixed architecture that is programmed in the field for a particular application. A typical, basic architecture, as shown in Figure 8-43, consists of (1) an array of programmable functional units (FUs) for implementing combinational and sequential logic, (2) a fixed, but programmable, interconnection fabric, which establishes the routing of signals, (3) a configuration memory, which programs the functionality of the device, and (4) I/O resources, which provide an interface between the device and its environment. The performance and density of FPGAs have advanced with improvements in process technology. Today's leading-edge devices include block memory as well as distributed memory, robust interconnection fabrics, global signals for high-speed synchronous operation, and programmable I/O resources matching a variety of interface standards.

Volatile FPGAs are configured by a program that can be downloaded and stored in static CMOS memory, called a configuration memory. The contents of the static RAM are applied to the control lines of static CMOS transmission gates and other devices to (1) program the functionality of the functional units, (2) customize configurable features, such as slew rate, (3) establish connectivity between functional units, and (4) configure I/O/bidirectional ports of the device. The configuration program is downloaded to the FPGA from either a host machine or from an on-board PROM. When power is removed from a volatile FPGA the program stored in memory is lost, and the device must be reprogrammed before it can be used again.

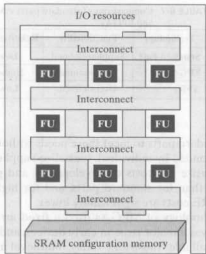


FIGURE 8-43 FPGA architecture.

The volatility of a stored-program FPGA is a double-edged sword—the FPGA must be reprogrammed in the event that power is disrupted, but the same generic part can be reprogrammed to serve a boundless variety of applications and it can be reconfigured on the same circuit board under the control of a processor. One of the programs that can be executed by an FPGA can even test the host system in which it is embedded. The ease of reprogramming a stored-program FPGA supports rapid prototyping, enabling design teams to compete effectively in an environment characterized by narrow and ever-shrinking windows of opportunity. Time-to-market is critical in many designs, and FPGAs provide a path to early entry. FPGAs can be reconfigured remotely, via the Internet, allowing designers to repair, enhance, upgrade, or completely reconfigure a device in the field.

8.7.1 The Role of FPGAs in the ASIC Market

The architectural resources of FPGAs match the general need for computational engines with memory, datapaths, and processors. The flexibility of an FPGA adds a dimension beyond what is available in masked-programmed devices, because mask-programmed devices cannot be reprogrammed. The same FPGA can be programmed to implement a variety of processors. Expensive, high-risk mask sets for ASICs have made flexibility an important consideration. On the other hand, FPGAs cost more per unit gate, and consume more power than a mask-programmed equivalent part.

Table 8-7 summarizes key distinctions between FPGA technology, cell-based and mask-programmed ASIC technologies, and standard parts. The myriad applications for ASICs and FPGAs require customization to address the needs of the market. The diversity, rewards, evolving technology, and short life span of the market preclude producing

TABLE 8-7 Comparisons of standard parts, ASICs, and FPGAs.

Technology	Functionality	Relative Cost
Standard Part	Supplier-defined	Low
FPGA	User-defined	Higher
ASIC	User-defined	Low

and stockpiling standard parts to meet these needs without unacceptably high risk. The lower volumes demanded by individual, specialized applications provide a smaller base over which to amortize the costs of development and production, so units costs for FPGAs are higher than for standard parts, and for high-volume, mask-programmed ASICs, but their NRE costs are significantly lower.

The early technology of MPGAs used a fixed array of transistors and routing channels. Routing was a major issue in early devices, and frequently led to incomplete utilization of the available transistors. Today, multilevel metal routing (e.g., five and six layers) in a sea-of-gates technology is commonplace, with high utilization of resources. MPGAs are preprocessed to the point of customizing the final metal layers to a particular application. The customization/metallization steps connect individual transistors to form gates and interconnect gates to implement logic. This technology provides a much quicker turnaround than cell-based and full-custom technologies, because only the final metallization step is customized, but not as fast as that for an FPGA. Depending on the foundry, an MPGA can be turned around in a few days to several weeks. On the other hand, designs can be implemented, programmed, and reprogrammed virtually instantaneously in an SRAM-based FPGA while the part is mounted in an emulator or in its target host application. But FPGAs will always be slower and less dense than a comparable MPGA because of the additional circuitry and delays introduced by their programmable interconnect.

FPGAs are fully tested by the manufacturer before they are shipped, so the designer's attention is focused on the creativity of the design, not on testing for manufacturing defects. Designers can quickly correct design flaws and reconfigure the part to a different functionality in the field. FPGAs address a market than cannot be met by mask-programmed technologies, which are one-time write. Mask-programmed technologies do not support reconfiguration and corrections are costly. The risk of an MPGA-based design is significantly higher than for an FPGA because a design flaw requires retooling of the final masks, with attendant costs and lost time to reenter the fabrication process queue.

MPGAs have a broad customer base for amortizing the NRE of most of the processing steps compared to cell-based and full-custom solutions. Gate arrays are widely used to implement designs that have a high content of random logic, such as state-machine controllers.

MPGAs require the direct support of a foundry, and the completion of a design can depend on the schedule and priority of the foundry's other customers. FPGAs are fully manufactured and tested in anticipation of being shipped immediately to a buyer.

The software interface between the designer and FPGA technology is simple, and it is now readily and cheaply available (if not free) on PCs and workstations that support schematic and HDL entry. Programmable logic technology continues to grow in density at exponential rates as compared with other technologies, such as dynamic random access memories (DRAMs). The speed of parts is growing at a linear rate and is now at a level that supports system-level integration.

Standard cell-based technology uses a library of predesigned and precharacterized cells that implement gates. The design of the individual cells in a library is labor-intensive, as efforts are made to achieve a dense, area-efficient layout. Consequently, a cell library has a high NRE cost, which a foundry must amortize over a large customer base during the lifetime of the underlying process technology. The mask set of a standard cell library is fully characterized and verified to be correct. Place and route tools select, place, and interconnect cells in rows on a chip to implement functionality. The structure is semiregular because the cell heights are fixed, while the width of cells may vary, depending on the functionality being implemented. Placement and routing are customized for each application. Place and route are done automatically to achieve dense configurations that meet speed and area constraints. Cell-based technology requires a fully customized mask set for each application. Consequently, volume must be sufficient to offset high production and development costs and ultimately drive an economically low unit cost.

8.7.2 FPGA Technologies

State-of-the-art FPGAs can now implement the functionality of over 1 million (two-input equivalent) gates on a single chip, and high-end parts (e.g. Xilinx Virtex 5) have over 200,000 flip-flops. Three basic types of FPGAs are available: antifuse, EPROM, and SRAM-based. The capacity and speed of these parts continues to evolve with process improvements that shrink minimum feature sizes of the underlying transistors.

Antifuse devices[15] are programmed in the field by applying a relatively high voltage between two nodes to break down a dielectric material. This eliminates the need for a memory to hold a program, but the one-time write configuration is permanent. When an antifuse is formed a low-resistance path is irreversibly created between the terminals of the device. The antifuse itself is relative small, about the size of a via, and over 1 million devices can be distributed over a single FPGA. The significant advantage of this technology is that the on-resistance and parasitic capacitance of an antifuse are much smaller than for transmission gates and pass transistors. This supports higher switching speeds and predictable timing delays along routed paths.

EPROM and EEPROM-based technology uses a charged floating gate, programmed by a high voltage. Devices based on these technologies are reprogrammable and nonvolatile and can be programmed offline while imbedded in the target system.

SRAM-based FPGA technology uses CMOS transmission gates to establish interconnect. The status of the gates is determined by the contents of the SRAM configuration memory. There are multiple vendors of SRAM-based FPGA products (e.g.,

[15]See www.actel.com for more information about antifuse devices.

Xilinx, Altera, ATMEL, and Lucent). The architecture of these FPGAs is similar to that of an MPGA, with block structures of logic and routing channels. Bidirectional and multiply driven wires are included. Devices are advertised on the basis of gate counts, but the actual use of the gates on a device depends on the router's ability to exploit the resources to support a given design.

The complexity of logic cells in an FPGA functional unit is based on competing factors. If the complexity of a cell is low (fine-grained, such as the Actel Act-1 part), the time and resources required for routing may be high. On the other hand, if the complexity is high, there will be wasted cell area and logic. An example of a fine-grained architecture would be one that is based on two-input NAND gates or muxes, as opposed to a large-grain architecture using four-input NAND gates or muxes. The former uses considerably more routing resources.

Given the rate at which process and device technology have been evolving, this text will limit its discussion of FPGAs to a representative device in the Xilinx family of parts. Readers are encouraged to consider readily available Web resources from manufacturers (e.g., altera.com, atmel.com, and xililnx.com).

8.7.3 XILINX Virtex FPGAs

The Virtex©-5 device series is the leading edge of Xilinx technology, based on a 65 nm process. The Virtex line addresses four key factors that influence the solution to complex system-level and system-on-chip (SoC) designs: (1) the level of integration, (2) the amount of embedded memory, (3) performance (timing), and (4) subsystem interfaces. The process rules allow over 330,000 logic cells and over 200,000 flipflops to be packed into a single die, providing several million system gates, and the capacity to support embedded processors and memory-intensive system-level applications requiring high density and high performance.

The Virtex family incorporates physical (electrical) and protocol support for a variety of I/O standards, including LVDS and LVPECL, with individually programmable pins. Digital clock managers provide support for frequency synthesis and phase shifting in synchronous applications that require multiple clock domains and high frequency I/O. The Virtex architecture is shown in Figure 8-44.

8.8 Embeddable and Programmable IP Cores for a System-on-a-Chip (SoC)

ASIC cores consist of intellectual property (IP) that has been designed, verified, and marketed by a vendor for re-use by other parties. Cores may be soft (software models) or hard (mask sets). The use of pre-implemented and verified embedded cores in an ASIC can shorten the time-to-market of a new product by reducing the amount of circuitry that must be developed. Whether this economy is realized depends on the reliability and documentation of the embedded logic and whether system-level tools exist for integrating and testing the embedded part.

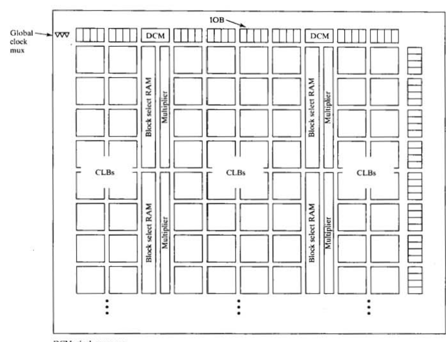


FIGURE 8-44 Xilinx Virtex-II overall architecture.

FPGA vendors have expanded their efforts beyond devices and design tools, and also provide an assortment of cores that can be embedded in a device to simplify the designer's task. For example, Xilinx offers, either directly or through partnerships with third parties, cores for basic elements (e.g., accumulators and shift registers), math functions (e.g., multipliers, multiply-and-accumulate [MAC] units, and dividers), memories (e.g., synchronous FIFO), standard bus interfaces, (e.g., PCI), processor peripherals (e.g., interrupt controller), universal asynchronous receiver and transmitters (UARTs), PCs (e.g., IBM PowerPC), and a variety of networking and communication products (e.g., protocol cores). Special design kits may be required to exploit these available resources. Vendors also provide reference designs, illustrating how to exploit embeddable cores.

ASIC designs are characterized by high performance, high NRE cost, and high risk. The risks are high because the cost of a mask set is high (e.g., $500k). A mask error and consequent re-spins of a design are prohibitive from the standpoint of cost and lost opportunity to capture market share. Embeddable programmable cores[16] offer flexibility (the design can be modified), lower NRE cost, and lower risk. These hybrid

[16]See the Web links for embeddable programmable cores.

devices are targeted at applications for which standards are evolving or for which NRE costs might have to be amortized over multiple variants of a product. For example, the control logic of a multiprocessor computer in a wireless network for image processing applications could be implemented in a programmable core, allowing the design to be modified to meet a dynamic marketplace. Multiple designs can be produced from a single die. Two aspects are involved here: the compatibility of the processes for manufacturing ASICs and FPGAs, and the IP that is ultimately configuring the FPGA for a specific application. The former will develop and set the stage for proliferation of the latter.[17] There are two variations on the theme: embeddable programmable cores for placement in an ASIC (Actel and Adaptive Silicon) and embeddable complex ASIC cores for placement within an FPGA (Triscend, Xilinx, Lucent, Altera, Atmel, QuickLogic).

Compared to FPGAs, ASICs are relatively expensive to design and manufacture. They gain performance at the expense of flexibility. An emerging technology is that of embedding an FPGA within an ASIC to gain flexibility, reduce the risk of a design,[18] and extend the life of a design to a wider range of applications.[19] Other programmable architectures are emerging as well. For example, Adaptive Silicon has developed a basic building block, called a Hex block, consisting of sixty-four 4-bit ALUs. Hex blocks can be tiled in rectangular patterns within a fabric of local and global interconnect. A 4×4 array of hex blocks supporting arithmetic functions will achieve a density of approximately 25,000 ASIC gates.

8.9 Verilog-Based Design Flows for FPGAs

The design flow for an FPGA-based target technology is shown in Figure 8-45. It relies heavily on bundled software to accomplish the synthesis, implementation, and downloading of the design into a part. The place-and-route step that plays such a dominant role in ASICs is not shown in the design flow because it is transparent to the user. Likewise, the extraction of parasitics is not shown because the fixed architecture of the devices allows their timing to be precharacterized to serve a database within the implementation tool. The simplified flow allows a designer to create design iterations and derivative designs rapidly, ultimately producing a hardware prototype.

The objective of rapid prototyping is to create a working prototype as quickly as possible to meet market conditions and to support broader testing in the host environment. Initially, the tools supporting FPGAs relied on schematic entry, but many vendors are now placing greater emphasis on supporting hardware description languages (HDLs). For example, the Xilinx ISE (integrated synthesis environment) tools are tailored for HDL-based entry, and support floor planning, simulation, automatic block placement and routing of interconnects, timing verification, downloading of

[17]The Virtual Socket Interface Alliance (VSIA) is an industry group that promotes technical standards for mixing and matching IP from multiple sources.
[18]Portions of the design that are risky and might require future change can be placed in the FPGA.
[19]LSI Logic and Adaptive Silicon have been working to embed an SRAM-based FPGA in an LSI ASIC.

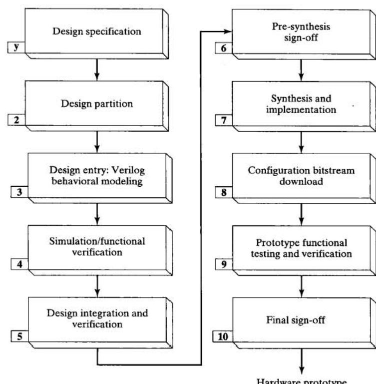


FIGURE 8-45 Design flow for FPGA-based designs with HDL entry.

configuration data, and readback of the configuration bit stream. The tools ultimately produce the bit-stream file that can be downloaded to the part to configure it on the host board.

8.10 Synthesis with FPGAs

In Chapter 6, we discussed the importance of adopting synthesis-friendly descriptive styles. A model has restricted utility if it cannot be synthesized. In addition, the models must include features that allow them to exploit the unique features of the target architecture. For example, FPGA tools must optimize the partition of memory between distributed and block memory resources. It is especially important in DSP applications that the synthesis tool minimize the use of off-chip memory in order to maximize performance.

FPGA vendors provide libraries of macros that implement specific functionality. The designer has a choice between a packaged macro and the circuit that a synthesis engine infers from a behavioral description. If technology-specific cores are used, they should be isolated within the design's hierarchy.

FPGAs are rich in registers, so it is generally advisable to employ one-hot coding of the state of a finite-state machine. There is usually little or no gain in trying to reduce the number of configurable logic blocks (CLBs) by employing a sequential binary code, because additional cells will be required to form the more complicated combinational logic that results from such a scheme.

The state decoding of a FSM must cover all possible codes of the state. Otherwise, latches will be introduced into the design.[20] This practice also protects against the machine entering a state from which it cannot recover. It is recommended that the designer assign the default state explicitly rather than use a tool option to do it automatically, if for no other reason than to encourage more thoughtful consideration about the design. (Use the Verilog keyword ***default*** as the item decoded in a ***case*** statement.) Also, as discussed in Chapter 6, it is recommended that all of the register variables that are assigned value within a level-sensitive cyclic behavior be initialized at the beginning of the listed code and then assigned value within the behavior by exception—as a way to help prevent synthesis of unwanted latches in the design.

The registers in a CLB do not power up to a specific state, so it is essential that a reset or set signal port be included at the top-level module of the design and be used to drive the machine into a known state. Synchronous resets are used to minimize the possibility of a reset signal causing a metastable condition.

Decoding logic should be implemented with ***case*** statements rather than ***if. . . then . . . else*** statements, unless a priority structure is intended. The former produces parallel logic (faster); the latter tends to produce logic that is nested (e.g., priority decoder) and will be cascaded in a multilevel series structure of LUTs, resulting in a slower circuit.

A net that fans out from a flip-flop to several points in a circuit can be slow and difficult to route; it might ultimately be the source of a timing constraint violation. This problem may be solved by duplicating the flip-flop so that the fan-out can be shared. The result will be that the routing step takes less time and is more likely to complete successfully, and the overall performance will be improved. The trade-off is that the solution occupies a larger area of the chip (more CLBs are required). Candidates for this treatment are the address and control lines of large memory arrays, clock enable lines, output enable lines, and synchronous reset signals. If the driver of a high–fan-out net is asynchronous, synchronize the signal before duplicating it.[21]

---

[20]Synthesis tools will produce reports describing device use, including a report on the number of latches and registers in the implementation. It is a good practice to review these reports to detect unwanted latches.

[21]Be aware that the Xilinx tools automatically map into the same CLB signals that end with the same numeric suffix, (e.g., *sig_1*, *sig_2*). Naming duplicated signals in this manner contradicts the effort to distribute the duplicated signals to different regions of the chip. Instead, use alphabetical labels (e.g., *_a*, *_b*) to compose the suffix of duplicated signals.

The throughput of a design can be improved by partitioning combinational logic systematically and inserting registers at the interface between the partitions. For example, a 16-bit adder can be partitioned into two 8-bit adders and pipelined to reduce the delay of the carry chain by a factor of 2. Pipelining shortens the path that a given signal must travel during a clock cycle. Consequently, the clock can run faster and timing violations can be eliminated. The trade-off, which could be unacceptable, is that the pipelined datapath has latency, because the data will take one or more additional clock cycles to propagate through the circuit, depending on the number of pipeline stages that have been added.[22] The second trade-off is that the pipeline registers occupy CLBs. Thus, the physical part that implements the design must be large enough to supply the additional registers for the pipeline. Reports produced by the software indicate the use of CLBs, so they should be consulted before attempting to increase the clock speed or eliminate a timing violation by pipelining.

If the place-and-route engine within the tool is allowed complete freedom, it will generate an optimal assignment of pins. This freedom is curtailed when the part must fit into the socket of a previously configured board. Ideally, the board is not configured until the FPGA has been fully designed. Constraining the pinout constrains the optimization process and may sacrifice performance. If feasible, careful pin assignment can lead to improved routing of the design. For example, the horizontal long lines in Xilinx architectures have three-state buffers, which makes them suitable for data busses. On the other hand, vertical long lines for clock enables and vertical carry chains lead naturally to a vertical orientation of the cells of registers and counters. These architectural features suggest that datapaths should be applied to the left and right sides of the part, and control lines should be applied to the top and bottom of the part when manual routing and pin assignment are necessary. The tool has maximum flexibility when no pins are pre-assigned, but environmental constraints may require that some pins be pre-assigned, before routing. However, it is recommended that the unconstrained design be routed first to verify that it can meet timing specifications. If it does not, the constrained design will also be too slow.

Keeping a design synchronous, with a single external clock source, allows timing-driven routing tools to work more efficiently. The parts have clock enables, so there is no need for special measures to gate clocks within a design.

FPGAs are register-rich. Therefore, it is advantageous to employ one-hot encoding in state machines. This leads to simpler next-state and output logic. This form of encoding is sometimes referred to as state-per-bit encoding, because a unique single flip-flop is asserted for each state. Coding style has an impact on the results of targeting a description into an FPGA. One notable example is in the description of a sequencer. If the count sequence does not have to be binary, linear feedback shift registers may be a more attractive alternative because they require less space and route more efficiently than binary counters. Designers should be aware that flip-flops in FPGAs tend to initialize to a cleared output during power-up. A state machine would have to anticipate this condition because it is not one of the explicit one-hot codes.

---

[22]Pipelining will be considered in more detail in Chapter 9.

REFERENCES

1. Wakerly KK. *Digital Design—Principles and Practices*. Upper Saddle River, NJ: Prentice-Hall, 2006.
2. Weste N, Eshraghian K. *Principles of CMOS VLSI Design*. Reading, MA: Addison-Wesley, 1993.
3. Sheilholeslami A, Gulak PG. "A Survey of Circuit Innovations in Ferroelectric Random-Access Memories," *Proceedings of the IEEE, 88*, 667–689.
4. Brayton RK, et al. *Logic Minimization Algorithms for VLSI Synthesis*. Boston, MA: Kluwer, 1984.
5. Tinder RF. *Engineering Digital Design*. 2nd ed. San Diego, CA: Academic Press, 2000.
6. Bartlett K, et al. "Synthesis of Multilevel Logic under Timing Constraints," *IEEE Transactions on Computer Aided Design of Integrated Circuits, CAD-7*, 582–596, 1986.
7. Bartlett K, et al. "Multilevel Logic Minimization using Implicit Don't-Cares," *IEEE Transactions on Computer Aided Design of Integrated Circuits, CAD-5*, 723–740, 1986.
8. Brayton RK, et al. "MIS: A multiple-level interactive logic optimization system." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, CAD-6*, 1062–1081.
9. Chan PK, Mourad S. *Digital Design Using Field Programmable Gate Arrays*. Upper Saddle River, NJ: Prentice-Hall, 1995.

RELATED WEB SITES

www.accellera.org	Accellera
www.actel.com	Actel Corp.
www.altera.com	Altera, Inc.
www.atmel.com	Atmel Corp.
www.cadence.com	Cadence Design Systems, Inc.
www.mentorg.com	Mentor Graphics Corp.
www.opencores.org	Opencores
www.synopsys.com	Synopsys, Inc.
www.synplicity.com	Synplicity, Inc.
www.xilinx.com	Xilinx, Inc.

PROBLEMS AND FPGA-BASED DESIGN EXERCISES

Note: The FPGA Design Exercises are suitable for a companion lab, independent study, or end-of-semester project. They are open-ended and progressively more challenging.

1. Using the ROM model given in Example 8.1, develop and verify *comp_2_ROM*, a Verilog model of a 2-bit comparator.
2. The 2-bit comparator presented in Example 8.1 has three outputs. Develop a new model that encodes the outputs in a 2-bit word. Build a testbench that will accept and decode the output of the model and assert one of three outputs corresponding to the outputs of the original model.

3. Estimate the number of memory cells that would be required to implement a 16-bit adder in a ROM.
4. Write a Verilog model, *ROM_256_x_8*, of a 256 × 8 ROM that stores the product of two 4-bit unsigned binary words, as shown in Figure P8-4. Use the multiplier (*mplr*) and multiplicand (*mcnd*) bits to form the address of the ROM.

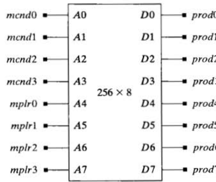


FIGURE P8-4

5. Write a testbench and verify the Verilog model of the static RAM cell, *RAM_static*, given in Example 8.3.
6. Develop an alternative model for *RAM_static* that uses a level-sensitive cyclic (***always***) behavior instead of a continuous assignment (see Example 8.3).

7. FPGA-Based Design Exercise[23]: A simple ALU

The top-level module of a sequential machine, *ALU_machine_4_bit*, is depicted in Figure P8-7a, with the input and output ports that interface the module to its environment. The machine is to operate synchronously as follows: *Led_idle* will indicate that the machine is in its "reset" state. When *Go* is asserted, an internal register is to be loaded with the content of *Data*[3:0] and is to assert *Led_wait* until *Go* is de-asserted. After *Go* is de-asserted, *Led_rdy* is to assert. While *Led_rdy* is asserted the slide switches (on a prototyping board) may be used to set a new value for *Data*[3: 0] and/or *Opcode*[2: 0]. As the slide switches are changed, the effect should be apparent at *Alu_out*. The cycle is to repeat if *Go* is re-asserted while *Led_rdy* is asserted (i.e., the storage register is to be reloaded). The machine is to be synchronized by the rising edge of a clock, and have synchronous active-high reset.

Design—Partition

An architectural partition of *ALU_machine_4_bit* is shown in Figure P8-7b. The architecture has three functional subunits: an ALU, a storage register, and a state-machine controller. The ALU implements an instruction set (described

[23]The FPGA design exercises are to be accomplished with an introductory-level FPGA prototyping board (see www.digilentinc.com) and the synthesis tool provided by an FPGA manufacturer (e.g., www.xilinx.com).

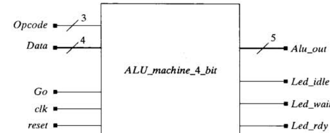


FIGURE P8-7a I/O ports for *ALU_machine_4_bit*.

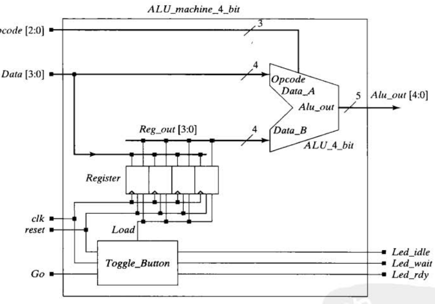


FIGURE P8-7b Architecture for *ALU_machine_4_bit*.

below) and the controller directs the operations of the machine. One input datapath of *ALU_machine_4_bit* is connected to the internal storage register and the other is connected to one data port of the ALU. The output of the register is connected to the other data port of the ALU. The datapath is to be controlled from *Toggle_Button*, a state machine that will be described below. The opcode and the input datapath of *ALU_machine_4_bit* will be controlled by the manual slide switches on the prototyping board. Board LEDs will be derived from the internal of the machine *Toggle_Button* to indicate the internal status of the system (i.e., *Led_idle*, *Led_wait*, and *Led_ready*).

The design of *ALU_machine_4_bit* will be progressive. The models of two functional units, *ALU_4_bit* and *Register*, will be developed and separately verified first. The state machine controller, *Toggle_Button*, will be designed later, along with a programmable clock generator. Then we will integrate the individually verified functional units, verify that the integrated design has the correct functionality, and achieve final pre-synthesis sign-off. The last step will be to synthesize the design into a working prototype on the FPGA board.

Design—ALU

Using the ***module*... *endmodule*** encapsulation and port declarations given below, write a Verilog model of *ALU_4_bit*, a 4-bit ALU shown in Figure P8-7c and specified in Table P8-7a.

```
module ALU_4_bit (output reg [4: 0] Alu_out, input [3: 0] Data_A, Data_B,
  input [2: 0] Opcode);
...
endmodule
```

Write a test plan that specifies the functional features that are to be tested and how they will be tested. Using the test plan, write a testbench, *t_ALU_4_bit*, that verifies the functionality of *ALU_4_bit*.

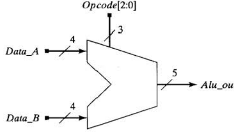


FIGURE P8-7c 4-bit ALU.

TABLE 8-7a Functional specification for a 4-bit ALU.

Code	Opcode	ALU Operation
000	*Add*	*Data_A + Data_B*
001	*Sub*	*Data_A − Data_B*
010	*Not_A*	*~Data_A*
011	*Not_B*	*~Data_B*
100	*A_and_B*	*Data_A & Data_B*
101	*A_or_B*	*Data_A | Data_B*
110	*Ror_A*	*|Data_A*
111	*Rand_B*	*&Data_B*

PDG

Verify your design for a suitable number of patterns that cover the data and opcodes of the ALU. As an example, complete Table P8-7b by specifying *Alu_out* for the indicated patterns. Complete the second table with patterns that you choose. Include these patterns in your testbench, along with others that you choose.

Note: Management appreciates your efforts to arrange the waveforms in the graphical display to enhance the utility of the information, minimizing the amount of interpretation and translation that must be done. (*Hint*: Set the radix of displayed waveforms to decimal or hex.) Consider annotating the waveforms by hand or by a graphical editor tool to label opcodes etc., or define and display text parameters to indicate mnemonics for opcodes.

Next, write a Verilog model of a 4-bit storage register that has parallel load capability. The register is to be synchronized by the rising edge of a clock, and have active-high synchronous reset. Use the module encapsulation and ports given below.[24]

```
module Register (output reg [3: 0] Reg_out, input [3: 0] Data, input
  Load, clk, reset);
...
endmodule
```

TABLE 8-7b Sample data calculations for test patterns to be applied to *ALU_4_bit*.

Data_A	1010	1111	0101	0101
Data_B	0101	0101	1010	1111
Opcode	*Alu_out*	*Alu_out*	*Alu_out*	*Alu_out*
Add Sub	0 1111 0 0101			
Not_A *Not_B*				
A_and_B *A_or_B*				
Ror_A *Rand_B*				

Data_A				
Data_B				
Opcode	*Alu_out*	*Alu_out*	*Alu_out*	*Alu_out*
Add Sub				
Not_A *Not_B*				
A_and_B *A_or_B*				
Ror_A *Rand_B*				

[24]Modify as needed to accommodate your model.

Write a test plan that specifies the functional features that are to be tested and how they will be tested. Using the test plan, write a testbench, *t_Register*, that verifies the functionality of *Register*.

Design—Programmable Clock

Using the programmable clock described in Problem 33 in Chapter 5, include the following "annotation module" in your project. The role of this module is to override the default parameters in the clock generator with those that are to be used in a given application. The annotation module uses hierarchical dereferencing, where *M1* is the instance name of the unit under test (UUT) in *t_Clock_Prog*. Your testbench must demonstrate that this works. The example below replaces the default values of *Latency, Offset*, and *Pulsewidth* by 10, 5, and 5, respectively.

```
module annotate_Clock_Prog ();
 defparam t_Clock_Prog.M1.Latency = 10;
 defparam t_Clock_Prog.M1.Offset = 5;
 defparam t_Clock_Prog.M1.Pulse_Width = 5;
endmodule
```

Design—User Interface

Values of the machine's input data will be set by slide switches on the prototyping board. The limited switch resources of the prototyping board create a need for a toggle button machine to control the loading of data into the machine's internal data register driving *Data_B*. The ALU will be driven by the data stored in the register and the data at the input bus driving *Data_A*.

The state machine described by the ASM chart in Figure P8-7d has synchronous reset and resides in *S_idle* with *Led_idle* asserted, until *Go* is asserted. Then it moves to *S_1*, where it asserts *Load* for one clock cycle and then enters *S_2*, where it asserts *Led_wait* and remains until *Go* is de-asserted. When *Go* is de-asserted the machine enters *S_3* and asserts *Led_rdy*. The machine remains in *S_3* until *Go* is again asserted. Then it returns to *S_1*. This sequence of state transitions lets us load data into a register, wait in *S_2* until the *Go* button is de-asserted, and then pause in *S_3*, where other actions can be taken to operate on a datapath. For example, data can be placed on the input port and the output port can be examined under the action of the opcodes. The outputs *Led_idle, Led_wait*, and *Led_rdy* indicate the status of the machine, and can be used to control LEDs on a prototyping board.

Write a Verilog model of the sequential machine *Toggle_Button*, using the module encapsulation and ports declared below:

```
module Toggle_Button (output Load, Led_idle, Led_wait, Led_rdy, input
   Go, clk, reset);
reg  [1: 0]      state, next_state;
...
endmodule
```

Write a test plan that specifies the functional features that are to be tested and how they will be tested. Using the test plan, write a testbench, *t_Toggle_Button*, that verifies the functionality of *Toggle_Button*.

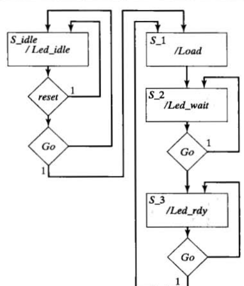


FIGURE P8-7d ASM chart for *Toggle_Button*.

Design—Integration and Verification

Now we will integrate the functional units of the architecture for *ALU_machine_4_bit* and verify that the functionality of the integration is correct, leading to pre-synthesis sign-off. Instantiate *ALU_4_bit, Register*, and *Toggle_Button* into *ALU_machine_4_bit* and create a Verilog model of the internal structure represented by the architecture shown in Figure P8-7b. Use the module header and declarations shown below.

```
module ALU_machine_4_bit (
  output [4: 0]    Alu_out,
  output           Led_idle, Led_wait, Led_rdy,
  input  [3: 0]    Data,
  input  [2: 0]    Opcode,
  input            Go,
  input            clk, reset
);
  wire   [3: 0]    Reg_out;          // Note: Must size the bus connecting
                                        M1 and M2

  ALU_4_bit        M1 (Alu_out, Data, Reg_out, Opcode);
  Register         M2 (Reg_out, Data, Load, clk, reset);
  Toggle_Button    M3 (Load, Led_idle, Led_wait, Led_rdy, Go, clk, reset);
endmodule
```

Write a test plan that specifies the functional features that are to be tested and how they will be tested. Using your test plan, and the headers and instantiations shown below, (1) write a carefully documented testbench, *t_ALU_machine_4_bit*, that implements the machine, and (2) verify the functionality of *ALU_machine_4_bit*. Note that the testbench contains the UUT (*ALU_machine_4_bit*), and the programmable clock generator (*Clock_Prog*), plus the code you write to execute the tests.

```
module annotate_ALU_machine_4_bit ();
 defparam t_ALU_machine_4_bit.M2.Latency = 10;
 defparam t_ALU_machine_4_bit.M2.Offset = 5;
 defparam t_ALU_machine_4_bit.M2.Pulse_Width = 5;
endmodule

module t_ALU_machine_4_bit ();
  wire    [4: 0]    Alu_out;
  wire              Led_idle, Led_wait, Led_rdy;
  wire              Load;
  reg               Go, reset;
  reg     [3: 0]    Data;
  reg     [2: 0]    Opcode;

  ALU_machine_4_bit M1 (                        // Instantiate UUT
   Alu_out,
   Led_idle, Led_wait, Led_rdy,
   Data,
   Opcode,
   Go, Load,
   clk, reset);
  Clock_Prog         M2 (clk);
 ...                 // Your code goes here
endmodule
```

Figure P8-7(e) shows the results of a simple test of *ALU_machine_4_bit*. Note the organization of the display.

Design—Prototype Synthesis and Implementation

The final steps of the exercise are to synthesize *ALU_machine_4_bit* into the particular FPGA of the prototyping board, download the design to the FPGA on the board, and demonstrate that the prototype functions correctly, concluding with final sign-off. The ports of the module, the pads of the FPGA, and the I/O resources of the board must be integrated.[25] The first step in this process is to decide which board resources will be mapped to the ports of the design. The second step is to map the ports to the I/Os of the FPGA. The pin configuration of the FPGA is described in the manufacturer's data sheets for the part. The board's I/O resources are described in the documentation provided by the

[25]Consult the data sheets for your particular prototyping board to identify the mapping between board resources and the I/O pads of the FPGA.

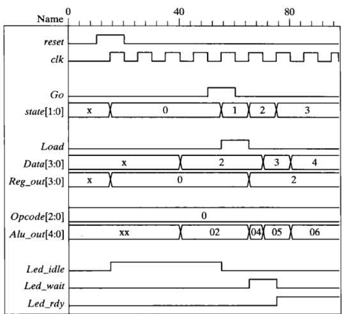


FIGURE P8-7e Simulation results for *ALU_machine_4_bit*.

manufacturer. *Caution*: Although a synthesis tool will map ports to pads (pins) automatically, the result might not be compatible with the fixed pad locations on the prototyping board and may even vary from run to run. It is advisable to constrain the pad mapping. In this application, the pads of the FPGA have been hardwired to certain pins of the prototyping board. *It is critical that the correct signals be mapped to the pins that are being used by the application.*

The datapath of *ALU_machine_4_bit* will be controlled by the finite-state machine *Toggle_Button* (previously designed and verified). The signal *Go* initiates the activity of loading *Data* into *Register*. *Data* and *Opcode* are to be control by the outputs of the manual slide switches. While the state of *Toggle_Button* is *S_3* the machine asserts *Led_rdy*, and the slide switches for *Data* and *Opcode* can be exercised to test the machine by presenting different words to the datapaths of the ALU. A value can be loaded into *Register*, then a different value can be arranged for *Data* by changing the slide switches after *Led_rdy* is asserted.

The input port signals of *ALU_machine_4_bit* must be mapped to the slide switches and push buttons of the prototyping board; the output ports are to be mapped to the LEDs. Figure P8-7(f) shows the (a) slide-switch configuration, (b) push-button configuration, and (c) the LED configuration used for a demonstration FPGA. These resources are connected to pins at a connector on the board. *Note*: Consult the data sheet to determine which side of the slide switch (e.g., the side that is closest to the nearest edge of the board) is logical 1.

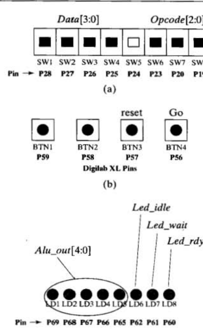


FIGURE P8-7f Example of pin assignments: (a) slide switches, (b) push buttons, and (c) LEDs.

Figure P8-7f shows the LED, button and switch pin assignments that have been specified for *ALU_machine_4_bit*, which are listed in Table P8-7(c). Note: Similar data must be produced for the particular FPGA and board used in your exercise.

Write a test plan that specifies how the FPGA prototype circuit will be tested, making specific reference to the board resources (e.g., slide switch configurations, LED readouts, special instrumentation, etc.). Develop test cases demonstrating that the ALU works correctly. Verify that the *Go* button, the reset button, and the LEDs function correctly. After successfully completing all of the above steps, create the bit-stream file and download it to the prototyping board. Using some of the test cases developed for the test plan, verify the functionality of the machine by executing (1) ALU and Opcode tests, (2) *Led_idle, Led_wait*, and *Led_rdy* assertion tests, (3) *reset* assertion test, and (4) any other test that will provide evidence that the prototype operates correctly.

8. FPGA-Based Design Exercise: Some Ring Counters

Using the code fragment below write and verify a Verilog model of *Counter8_Prog*, a parameterized and programmable 8-bit counter that implements various display patterns to exercise the LEDs of a prototyping board. The implementation is to use a separate Verilog function for each pattern (e.g., *ring1_count*). Develop a test plan

TABLE P8-7c Pin assignments for the prototype board.

Port	Pin
Alu_out [4]	P69
Alu_out [3]	P68
Alu_out [2]	P67
Alu_out [1]	P66
Alu_out [0]	P65
Led_idle	P62
Led_wait	P61
Led_rdy	P60
Data [3]	P28
Data [2]	P27
Data [1]	P26
Data [0]	P25
Opcode [2]	P23
Opcode [1]	P20
Opcode [0]	P19
clk	P13
Go	P56
reset	P57

clearly listing each functional feature that is to be tested. Develop a carefully documented testbench and execute the testplan to debug and verify the model and generate final graphical results. (*Note*: Organize the display to have the format shown Figure P8-8(a).

```
module Counter8_Prog (
 output reg [7: 0] count, input [1: 0] mode, input direction, enable, clk, reset);
 parameter       start_count      = 1;      // Sets initial pattern of the
                                               display to LSB of count

 // Mode of count
 parameter              binary  = 0;
 parameter              ring1   = 1;
 parameter              ring2   = 2;
 parameter              jump2   = 3;

 // Direction of count
 parameter              left    = 0;
 parameter              right   = 1;
 parameter              up      = 0;
 parameter              down    = 1;

 always @ (posedge clk or posedge reset)
  if (reset ==1)count <= start_count;
  else if (enable ==1)
   case (mode)
```

```
    binary:     count <= binary_count    (count, direction);
    ring1:      count <= ring1_count     (count, direction);
    ring2:      count <= ring2_count     (count, direction);
    jump2:      count <= jump2_count     (count, direction);
   default:     count <= binary_count    (count, direction);
  endcase
 function        [7: 0]   binary_count;
  input          [7: 0]   count;
  input                   direction;
  begin
   if (direction == up) binary_count = count +1; else binary_count = count−1;
  end
 endfunction

// Other functions are declared here.
endmodule
```

At the active edge of the clock, an 8-bit count will be updated under the control of mode and direction, which selects one of four different functions to form the next value of *count*.

binary: A binary count pattern controlled by *direction* to count up or down.
*ring*1: A ring counter controlled by *direction* to move left (up) or right (down).

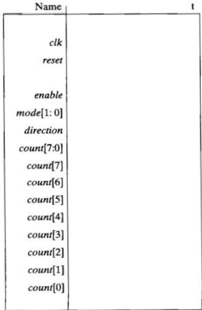


FIGURE P8-8a Display format for ring counter simulation results.

*ring*2: A ring counter like *ring*1, but that moves two adjacent cells at a time.

*jump*2 A ring counter that jumps by two cells.

The patterns that are to be implemented for *ring*2 and *jump*2 are illustrated in Figure P8-8b.

Design—Prototype Synthesis and Implementation

The frequency of the clock signal at Pin 13 on the prototyping board is 25 + MHz. Model and verify a clock divider that will produce an internal clock signal whose frequency will be low enough to allow changes in the LEDs to be visible.

Your design is to be encapsulated in a module, *TOP*, having the following structure:

```
module TOP (count, mode, direction, enable, clk, reset);
input ...
output ...
Clock_Divider M0 (clk_internal, clk);
Counter8_Prog UUT (count, mode, direction, enable, clk_internal, reset);
endmodule
```

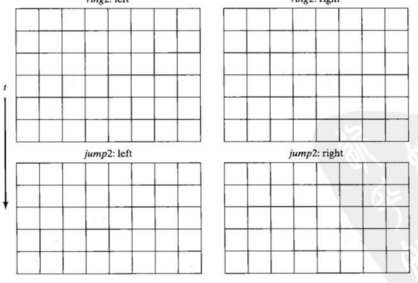


FIGURE P8-8b Patterns to be generated by *ring*2 and by *jump*2.

Synthesize and implement your design on a prototyping board. Develop a hardware test plan and use it to test the operation of the prototype.

Using the pin assignments shown in Figure P8-8(c), connect the ports of the design to the pins of the FPGA on the prototyping board. (An example of pin assignments is shown in Figure P8-8(c). If necessary, consider a clock divider to enable the transitions of the counters to be visible.)

Develop and verify a Verilog model of *Jumper*, a module that generates the pattern in Figure P8-8(d). Synthesize and verify a hardware prototype.

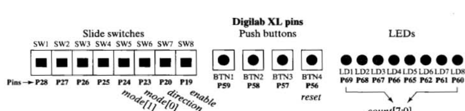


FIGURE P8-8c Pin assignments for *Counter8_Prog*.

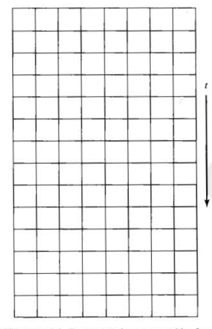


FIGURE P8-8d Patterns to be generated by *Jumper*.

9. FPGA-Based Design Exercise: SRAM with Controller

The static RAM modeled by *SRAM_2048_8* (see Example 8.5) is asynchronous. Many applications require a synchronous interface to an SRAM. One such interface (controller) is illustrated in Figure P8-9(a) below, where a host processor provides an active-low address strobe, *ADS_b*, a read/write signal *R_W*, a clock signal, and a reset to a state machine, *SRAM_Con*, which forms the signals *OE_b, CS_b*, and *WE_b*, which control the SRAM, and a signal, *Rdy*, which asserts for one clock cycle at the end of a read or write sequence.

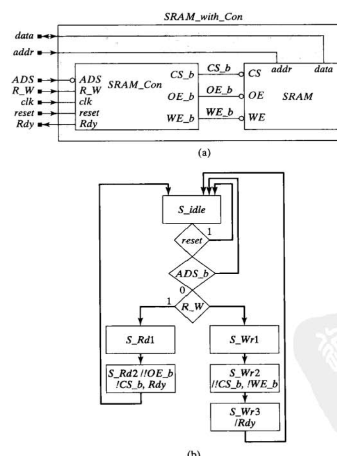


FIGURE P8-9a Synchronous controller for a SRAM: (a) hierarchical block diagram with interface signals and (b) ASM chart for the controller.

The ASM chart in Figure P8-9(b) describes the controller. (*Note*: the notation *!CS_b* indicates that the active low chip-select signal has $CS_b = 0$; its default value is $CS_b = 1$.)

Form a generic module, *SRAM*, by renaming *SRAM_2048_8* and re-using its code. Using the headers and testbench below, develop a generic (parameterized) controller module, *SRAM_Con* that implements the behavior of the ASM chart in Figure P8-9(b), then instantiate *SRAM* and *SRAM_Con* within *SRAM_with_Con*. Use the results shown in Figure P8-9(b) to organize the displayed information

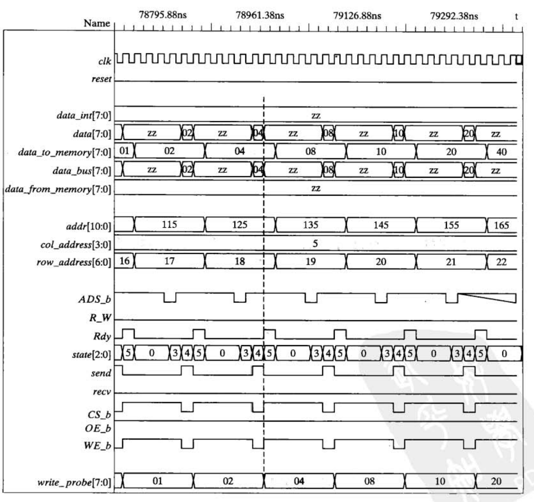


FIGURE P8-9b Simulation results for *SRAM_Con: writing to memory.*

(including the bus activity) produced by the simulator. Figure 8-9(b) shows an example of a write to memory, with *ADS_b* launching the activity. With *send* high, *recv* low, and *R_W* low, the bidirectional bus is controlled for writing; with *send* low, *recv* high, and *R_W* low, the bus is configured for reading data from memory. *Rdy* assert for one cycle after the data is written to memory. The testbench includes *write_probe*, which monitors the contents of memory at the location specified by *col_address* and *row_address* shows that *data_to_memory* (04) is written successfully. Figure P8-9(c) shows signal activity for a read operation, and Figure 8-9(d) shows the bidirectional interface to the testbench.

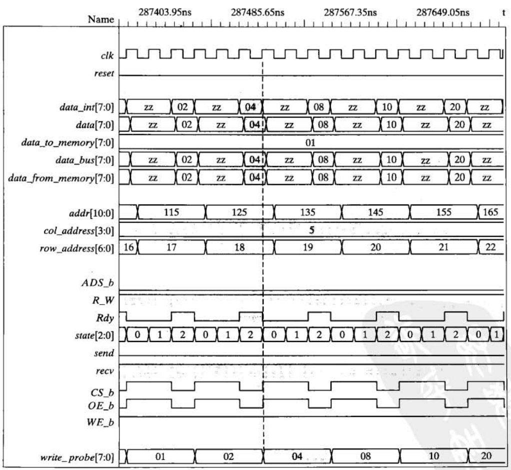


FIGURE P8-9c Simulation results for *SRAM_Con: reading from memory.*

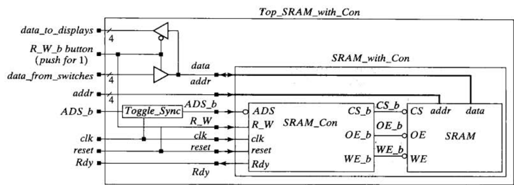


FIGURE P8-9d FPGA implementation of an SRAM with a synchronous controller.

Note that the testbench contains commented statements that would cause the model to fail, because the bus activity presents a high-impedance condition before the SRAM can latch the data. The remedy is shown in the code for the testbench, and consists of conditioning the bus to enter a high-impedance condition after the rising latching edge of *CS_b* or *WE_b* in the SRAM.

Design—Prototype Synthesis and Implementation

After verifying the functionality of the model for the parameters of *SRAM_2048_8*, choose parameters that will size the model to fit in the an available FPGA for implementation of the integrated unit on a prototyping board. Consider the structure shown in Figure P8-9(d), where the inbound data and the address are mapped to the pins of the board's slide switches, and *ADS_b*, *R_W*, and reset are mapped to push buttons. *Rdy* is to be mapped to a LED. The outbound data is to be mapped to the board's seven segment displays (a decoder will be needed) or to the LEDs. Consider the following issues: (1) bus contention, (2) display of the machine state (consider the LEDs), and (3) clock speed. The module *Toggle_Synch* is to synchronize the asynchronous input *ADS_b* and to have logic that generates a single pulse on assertion of *ADS_b* regardless of how long the push button is pressed.

```
'timescale 1ns / 10ps

module SRAM_with_Con #(parameter word_size = 8, addr_size = 11)(
  inout   [word_size -1: 0] data,
  input   [addr_size -1: 0] addr,
  output Rdy,
  input   ADS_b, R_W,
  input   clk, reset
);
  SRAM_Con M0 (Rdy, CS_b, OE_b, WE_b, ADS_b, R_W, clk, reset);
  SRAM M1 (data, addr, CS_b, OE_b, WE_b);
endmodule
```

```
module SRAM_Con (output reg Rdy, CS_b, OE_b, WE_b, input ADS_b,
  R_W, clk, reset);
  // Your code goes here
endmodule

module SRAM #(parameter
  word_size = 8,
  addr_size = 11,
  mem_depth = 128,
  col_addr_size = 4,
  row_addr_size = 7,
  Hi_Z_pattern = {word_size{1'bz}}
  )(
  inout [word_size-1: 0]    data,
  input [addr_size-1: 0]    addr,
  input                     CS_b, OE_b, WE_b
);
  reg [word_size-1: 0] data_int;
  reg [word_size-1: 0] RAM_col0 [mem_depth-1: 0];
  reg [word_size-1: 0] RAM_col1 [mem_depth-1: 0];
  reg [word_size-1: 0] RAM_col2 [mem_depth-1: 0];
  reg [word_size-1: 0] RAM_col3 [mem_depth-1: 0];
  reg [word_size-1: 0] RAM_col4 [mem_depth-1: 0];
  reg [word_size-1: 0] RAM_col5 [mem_depth-1: 0];
  reg [word_size-1: 0] RAM_col6 [mem_depth-1: 0];
  reg [word_size-1: 0] RAM_col7 [mem_depth-1: 0];
  reg [word_size-1: 0] RAM_col8 [mem_depth-1: 0];
  reg [word_size-1: 0] RAM_col9 [mem_depth-1: 0];
  reg [word_size-1: 0] RAM_col10 [mem_depth-1: 0];
  reg [word_size-1: 0] RAM_col11 [mem_depth-1: 0];
  reg [word_size-1: 0] RAM_col12 [mem_depth-1: 0];
  reg [word_size-1: 0] RAM_col13 [mem_depth-1: 0];
  reg [word_size-1: 0] RAM_col14 [mem_depth-1: 0];
  reg [word_size-1: 0] RAM_col15 [mem_depth-1: 0];

  wire [col_addr_size-1: 0]     col_addr = addr[col_addr_size-1: 0];
  wire [row_addr_size-1: 0]     row_addr = addr[addr_size-1:
                                 col_addr_size];
  assign data = ((CS_b == 0) && (WE_b == 1) && (OE_b == 0))
    ? data_int: Hi_Z_pattern;

  always @ (data, col_addr, row_addr, CS_b, OE_b, WE_b)
    begin
     data_int = Hi_Z_pattern;
     if ((CS_b == 0) && (WE_b == 0))          // Priority write to memory
      case (col_addr)                                // column address
       0: RAM_col0[row_addr] = data;
       1: RAM_col1[row_addr] = data;
       2: RAM_col2[row_addr] = data;
       3: RAM_col3[row_addr] = data;
       4: RAM_col4[row_addr] = data;
```

```
      5: RAM_col5[row_addr] = data;
      6: RAM_col6[row_addr] = data;
      7: RAM_col7[row_addr] = data;
      8: RAM_col8[row_addr] = data;
      9: RAM_col9[row_addr] = data;
      10: RAM_col10[row_addr] = data;
      11: RAM_col11[row_addr] = data;
      12: RAM_col12[row_addr] = data;
      13: RAM_col13[row_addr] = data;
      14: RAM_col14[row_addr] = data;
      15: RAM_col15[row_addr] = data;
    endcase

   else if ((CS_b == 0) && (WE_b == 1) && (OE_b == 0)) // Read from
    memory
    case (col_addr)
     0: data_int = RAM_col0[row_addr];
     1: data_int = RAM_col1[row_addr];
     2: data_int = RAM_col2[row_addr];
     3: data_int = RAM_col3[row_addr];
     4: data_int = RAM_col4[row_addr];
     5: data_int = RAM_col5[row_addr];
     6: data_int = RAM_col6[row_addr];
     7: data_int = RAM_col7[row_addr];
     8: data_int = RAM_col8[row_addr];
     9: data_int = RAM_col9[row_addr];
     10:data_int = RAM_col10[row_addr];
     11:data_int = RAM_col11[row_addr];
     12:data_int = RAM_col12[row_addr];
     13:data_int = RAM_col13[row_addr];
     14:data_int = RAM_col14[row_addr];
     15:data_int = RAM_col15[row_addr];
    endcase
  end
///* Comment out of the model for a zero delay functional test.
specify

 // Parameters for the read cycle
 specparam t_RC = 10;          // Read cycle time
 specparam t_AA = 8;           // Address access time
 specparam t_ACS = 8;          // Chip select access time
 specparam t_CLZ = 2;          // Chip select to output in low-z
 specparam t_OE = 4;           // Output enable to output valid
 specparam t_OLZ = 0;          // Output enable to output in low-z
 specparam t_CHZ = 4;          // Chip de-select to output in hi-z
 specparam t_OHZ = 3.5;        // Output disable to output in hi-z
 specparam t_OH = 2;           // Output hold from address change

 // Parameters for the write cycle
 specparam t_WC = 7;           // Write cycle time
 specparam t_CW = 5;           // Chip select to end of write
```

```
  specparam t_AW = 5;          // Address valid to end of write
  specparam t_AS = 0;          // Address setup time
  specparam t_WP = 5;          // Write pulse width
  specparam t_WR = 0;          // Write recovery time
  specparam t_WHZ = 3;         // Write enable to output in hi-z
  specparam t_DW = 3.5;        // Data set up time
  specparam t_DH = 0;          // Data hold time
  specparam t_OW = 10;         // Output active from end of write

 //Module path timing specifications
  (addr *> data) = t_AA;                              // Verified in simulation
  (CS_b *> data) = (t_ACS, t_ACS, t_CHZ);
  (OE_b *> data) = (t_OE, t_OE, t_OHZ);               // Verified in simulation

 //Timing checks (Note use of conditioned events for the address setup,
 //depending on whether the write is controlled by the WE_b or by CS_b.

 //Width of write/read cycle
  $width (negedge addr, t_WC);

 //Address valid to end of write

  $setup (addr, posedge WE_b &&& CS_b == 0, t_AW);
  $setup (addr, posedge CS_b &&& WE_b == 0, t_AW);

 //Address setup before write enabled

  $setup (addr, negedge WE_b &&& CS_b == 0, t_AS);
  $setup (addr, negedge CS_b &&& WE_b == 0, t_AS);

 //Width of write pulse
  $width (negedge WE_b, t_WP);

 //Data valid to end of write
  $setup (data, posedge WE_b &&& CS_b == 0, t_DW);
  $setup (data, posedge CS_b &&& WE_b == 0, t_DW);

 //Data hold from end of write
  $hold (data, posedge WE_b &&& CS_b == 0, t_DH);
  $hold (data, posedge CS_b &&& WE_b == 0, t_DH);

 //Chip sel to end of write
  $setup (CS_b, posedge WE_b &&& CS_b == 0, t_CW);
  $width (negedge CS_b &&& WE_b == 0, t_CW);

 endspecify
//*/
endmodule

module test_SRAM_with_Con ();
 parameter      word_size = 8;
 parameter      addr_size = 11;
 parameter      mem_depth = 128;
 parameter      col_addr_size = 4;
 parameter      row_addr_size = 7;
```

```
parameter     num_col = 16;
parameter     initial_pattern = 8'b000_0001;
parameter     Hi_Z_pattern = 8'bzzzz_zzzz;
parameter     stop_time = 290000;
parameter     latency = 248000;
reg           [word_size -1: 0]        data_to_memory;
reg           ADS_b, R_W, clk, reset;
reg           send, recv;
integer       col, row;
wire          [col_addr_size -1: 0]    col_address = col;
wire          [row_addr_size -1: 0]    row_address = row;
wire          [addr_size -1: 0]        addr = {row_address,
                                        col_address};

// Three-state, bi-directional bus
  wire [word_size -1: 0] data_bus = send? data_to_memory: Hi_Z_pattern;
  wire [word_size -1: 0] data_from_memory = recv? data_bus: Hi_Z_pattern;
  SRAM_with_Con M1 (data_bus, addr, Rdy, ADS_b, R_W, clk, reset);  // UUT

  initial #stop_time $finish;
  initial begin reset = 1; #1 reset = 0; end

  initial begin
   #0 clk = 0;
  forever #10 clk = ~clk;
  end

// Non-Zero delay test: Write walking ones to memory
  initial begin
   ADS_b = 1;
   R_W = 0;
   send = 0;
   recv = 0;
   for (col= 0; col<= num_col -1; col = col +1) begin
   data_to_memory =initial_pattern;

   for (row = 0; row <= mem_depth-1; row = row + 1) begin
    @ (negedge clk);
    @ (negedge clk);
    @ (negedge clk) ADS_b = 0; R_W = 0;  // writing
    @ (negedge clk) ADS_b = 1;
    @ (posedge clk) send = 1;
    // @ (posedge clk) send = 0;      // Does not work
    @  (posedge M1.M1.WE_b or posedge M1.M1.CS_b) send = 0;
    //@ (posedge clk) #1 send = 0; //Replacing above line with this works too.
    @ (posedge clk) data_to_memory =
     {data_to_memory[word_size-2:0],data_to_memory[word_size -1]};

    end
   end
  end
```

```
// Non-Zero delay test: Read back walking ones from memory
 initial begin
  #latency;
  ADS_b = 1;
  R_W = 1;
  send = 0;
  recv = 1;
  ADS_b = 0;
  for (col= 0; col <= num_col-1; col = col +1) begin
   for (row = 0; row <= mem_depth-1; row = row + 1) begin
    #60;
   end
  end
 end

// Testbench probe to monitor write activity
 reg [word_size -1:0] write_probe;

 always @ (posedge M1.M1.WE_b, posedge M1.M1.CS_b)
  case (M1.M1.col_addr)

   0: write_probe = M1.M1.RAM_col0[M1.M1.row_addr];
   1: write_probe = M1.M1.RAM_col1[M1.M1.row_addr];
   2: write_probe = M1.M1.RAM_col2[M1.M1.row_addr];
   3: write_probe = M1.M1.RAM_col3[M1.M1.row_addr];
   4: write_probe = M1.M1.RAM_col4[M1.M1.row_addr];
   5: write_probe = M1.M1.RAM_col5[M1.M1.row_addr];
   6: write_probe = M1.M1.RAM_col6[M1.M1.row_addr];
   7: write_probe = M1.M1.RAM_col7[M1.M1.row_addr];
   8: write_probe = M1.M1.RAM_col8[M1.M1.row_addr];
   9: write_probe = M1.M1.RAM_col9[M1.M1.row_addr];
   10:write_probe = M1.M1.RAM_col10[M1.M1.row_addr];
   11:write_probe = M1.M1.RAM_col11[M1.M1.row_addr];
   12:write_probe = M1.M1.RAM_col12[M1.M1.row_addr];
   13:write_probe = M1.M1.RAM_col13[M1.M1.row_addr];
   14:write_probe = M1.M1.RAM_col14[M1.M1.row_addr];
   15:write_probe = M1.M1.RAM_col15[M1.M1.row_addr];
  endcase
endmodule
```

10. FPGA-Based Design Exercise: Programmable Lock

The objective of this exercise is to design and implement a hardware prototype of a programmable digital combination lock, using a prototyping board and a hexadecimal keypad. The top-level block diagram of the programmable lock is shown in Figure P8-10a. The lock has a six-keystroke, factory-programmed combination, but allows the owner to reprogram a new combination. LEDs display the status of the machine and prompt the user to take action.

The programmable lock has two modes: normal and programming. The action of *reset* is to place the machine into the reset state (*S_idle*), where *Ready*

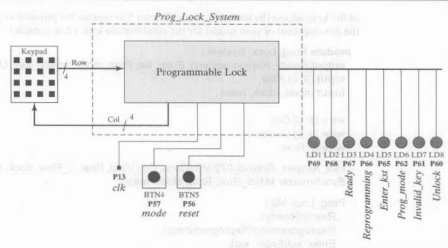


FIGURE P8-10a Top-level block diagram for a programmable combination lock.

and *Enter_kst* LEDs are asserted. The combination of the lock can be reprogrammed by pressing *mode* once while the machine is in *S_idle* to put the machine in the program mode state (*S_prog*). In the program mode, the *Prog_mode* and *Reprogramming* LEDs are asserted, and the machine accepts a sequence of eight entries of code from a hex keypad. This eight-keystroke combination (its reprogramming code) is also factory-programmed and cannot be changed. If the eight keystrokes that are entered in the programming mode match the reprogramming code, the machine asserts the *Enter_kst and Reprogramming* LEDs and awaits entry of six more code values from the keypad. If the sequence of eight values is not correct, the machine de-asserts the *Prog_mode* LED, returns to the reset state, and asserts the *Ready* and *Enter_kst* LEDs. The six values entered after the *Reprogramming* and *Enter_kst* LEDs are asserted will be the reprogrammed key for the lock. After the six keystrokes are entered, the machine returns to the reset state and the normal mode automatically.

In the normal mode, a user must enter a sequence of six hex characters. If the sequence of characters matches the key, the *Unlock* LED is asserted for one clock cycle before the machine returns to *S_idle*. If the sequence does not match the combination, the machine asserts the *Invalid_key* LED for one clock cycle, and then returns to *S_idle*. For security, the machine must not signal *Invalid_key* until the entire key has been entered. The machine must a time-out constraint—if the combination or the reprogramming codes are not entered with a specified time the machine aborts and returns to *S_idle*. The machine does not support backspace/erase of keystrokes, and does not monitor or prevent attempts by hackers.

Develop an ASMD chart for a programmable lock meeting the specifications given above. Using the chart and the ***module . . . endmodule*** encapsulations and ports given below, develop and verify a Verilog model of *Prog_Lock*. Incorporate *Hex_Keypad_Grayhill_072* from Chapter 5 to decode the keystrokes

of the keypad; use the testbench from Chapter 5 to replace the physical keypad in the development of your model for the combination lock. *Hint*: consider

```
module Prog_Lock_System (
  output Ready, Reprogramming, Enter_kst, Prog_mode, Invalid_key, Unlock,
  input [3: 0] Row,
  input mode, clock, reset
);
  wire [3: 0] Col;
  wire [3: 0] Code;
  wire S_Row;

  Hex_Keypad_Grayhill_072 M0 (Code, Col, Valid, Row, S_Row, clock, reset);
  Synchronizer M1(S_Row, Row, clock, reset);

  Prog_Lock M2 (
    .Ready(Ready),
    .Reprogramming(Reprogramming),
    .Enter_kst(Enter_kst),
    .Prog_mode(Prog_mode),
    .Invalid_key(Invalid_key),
    .unlock(Unlock),
    .code(Code),
    .mode(mode),
    .kst(Valid),
    .clock(clock),
    .reset(reset)
  );
endmodule
```

Design—Issues

Consider debouncing the keypad and/or reducing the frequency of the clock provided on the prototyping board. Contact bounce is 4 ms at make and 10 ms at break. Specifications for the Grayhill 072 keypad are available at www .grayhill.com.

Design—Prototype Synthesis and Implementation

A sample configuration for connecting the Grayhill 072 keypad to the prototyping board is shown below in Figure P8-10(b). This sample configuration uses the same pins that connect to the slide switches on the board. Therefore, the slide switches must be place in the 0 position (away from the nearest edge of the board), and may not be used for any other function.

11. FPGA-Based Design Exercise: Keypad Scanner with FIFO Storage

The objective of this multistage exercise is to systematically design and implement an FPGA-based keypad scanner integrated with a FIFO for data storage and retrieval, a display mux, and the seven-segment displays, slide switches, and

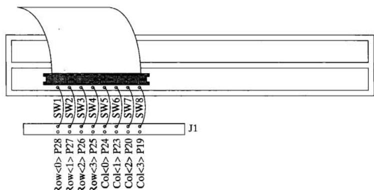


FIGURE P8-10b Digilab board pin mapping for the Grayhill 072 Hexadecimal keypad.

LEDs of a prototyping board. The top-level block diagram of the system is shown in Figure P8-11(a), with the partitioned system. The hardware prototype will be verified to operate with the Grayhill 072 hex Keypad.

The user interface consists of the following inputs: a mode toggle button, a read button, a reset button, and a hexadecimal keypad. The outputs are two seven-segment displays and eight LEDs. The button *mode_toggle* will be used to toggle between display states so that more than eight signals can be presented for view. When a button of the hex keypad is pressed, the system must decode the button and store the data in an internal FIFO. The *read* button will be used to read data from the FIFO and to display the data on the seven-segment displays. The LEDs will display the status of the FIFO and other information.

The system architecture is subject to future engineering change orders (ECOs) as the customer's specifications evolve to accommodate a rapidly changing marketplace. Note that the specification has not considered the need for a switch debounce circuit and that it does not address the constraints that will be imposed by the prototyping board's circuitry for the seven-segment displays.

Design: FIFO

The design will use the keypad decoder and synchronizer that were presented in Chapter 5. This part of the exercise will integrate a FIFO[26] with the keypad scanner. A FIFO (first-in, first-out) buffer is a dedicated memory stack consisting of a fixed array of registers. The FIFO that is to be used in this exercise is shown in Figure P8-11b. The registers of the stack operate synchronously (rising edge) with a common clock for reading and writing, subject to reset. The reset action does not affect the contents of the stack. The stack has two pointers (addresses), one pointing to the next word to which data will be written and another pointing to the next word that will be read, subject to write and read inputs, respectively. In the implementation given here, the action to read from

[26]Chapter 9 provides a more detailed discussion of FIFOs.

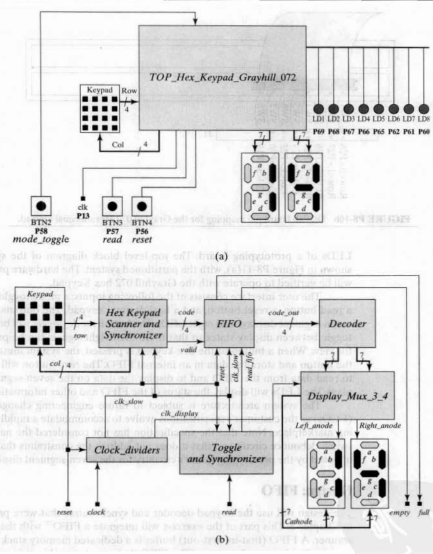


FIGURE P8-11a FIFO Keypad Scanner: (a) top-level block diagram and (b) system partition and architecture.

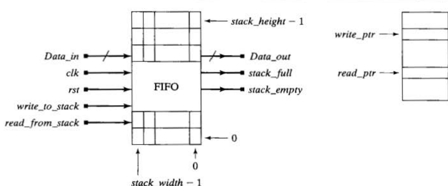


FIGURE P8-11b FIFO Buffer: Signal Interface.

the FIFO has priority over the action to write data to the FIFO. The FIFO has input and output datapaths, and two bit-lines serving as flags to denote the status of the stack (full or empty).

Using the FIFO model and testbench provided below, verify the operation of the FIFO. Configure the FIFO to have a stack of eight registers of 4 bits width.

```
// Note: Adjust stack parameters
// Note: Model does not support simultaneous read and write.

module FIFO #(parameter
  stack_width = 4,                          // Width of stack and data paths
  stack_height = 8,                         // Height of stack (in # of words)
  stack_ptr_width = 3                       // Width of pointer to address
                                               stack
)(output reg [stack_width -1: 0] Data_out,  // Data from FIFO
  output stack_empty, stack_full,           // FIFO status flags
  input [stack_width -1: 0] Data_in,        // Data to FIFO
  input write_to_stack, read_from_stack,
  input clk, rst
);
  reg [ stack_ptr_width -1: 0] read_ptr, write_ptr;  // Pointers (addresses) for
                                                      // reading and writing
  reg [ stack_ptr_width : 0] ptr_diff;                // Gap between ptrs
  reg [stack_width -1: 0] stack [stack_height -1 : 0]; // memory array

  assign stack_empty = (ptr_diff == 0) ? 1'b1 : 1'b0;
  assign stack_full = (ptr_diff == stack_height) ? 1'b1: 1'b0;
```

```
  always @ (posedge clk, posedge rst) begin: data_transfer
   if (rst) begin
    Data_out <= 0;
    read_ptr <= 0;
    write_ptr <= 0;
    ptr_diff <= 0;
   end
   else begin
    if ((read_from_stack) && (!stack_empty)) begin
     Data_out <= stack [read_ptr];
     read_ptr <= read_ptr + 1;
     ptr_diff <= ptr_diff -1;
    end
    else if ((write_to_stack) && (!stack_full)) begin
     stack [write_ptr] <= Data_in;
     write_ptr <= write_ptr + 1;          // Address for next clock edge
     ptr_diff <= ptr_diff + 1;
    end
   end
  end // data_transfer
endmodule

module t_FIFO ();
 parameter stack_width = 4;
 parameter stack_height = 8;
 parameter stack_ptr_width = 3;

 wire [stack_width -1 : 0] Data_out;
 wire stack_empty, stack_full;
 reg [stack_width -1 : 0] Data_in;
 reg clk, rst, write_to_stack, read_from_stack;
 wire    [11:0]    stack0, stack1, stack2, stack3, stack4, stack5, stack6,
                   stack7;

 assign stack0 = M1.stack[0];             // Probes of the stack
 assign stack1 = M1.stack[1];
 assign stack2 = M1.stack[2];
 assign stack3 = M1.stack[3];
 assign stack4 = M1.stack[4];
 assign stack5 = M1.stack[5];
 assign stack6 = M1.stack[6];
 assign stack7 = M1.stack[7];

 FIFO M1 (Data_out, stack_empty, stack_full, Data_in, write_to_stack,
  read_from_stack, clk, rst);

 always begin clk = 0; forever #5 clk = ~clk; end
 initial #1500 $stop;

 initial begin
  #10 rst = 1;
```

```
    #40 rst = 0;
    #420 rst = 1;
    #460 rst = 0;
  end
  initial fork
    #80 Data_in = 1;
    forever #10 Data_in = Data_in + 1;
  join
  initial fork
    #80 write_to_stack = 1;
    #480 write_to_stack = 0;

    #250 read_from_stack = 1;
    #350 read_from_stack = 0;

    #420 write_to_stack = 1;
    #480 write_to_stack = 0;
  join
endmodule
```

Design: Decoder

The decoder must be designed to decode a word read from the FIFO and create driving signals for the active-low, seven-segment displays on a prototyping board. The decoder must generate display signals for the 16 codes of the *Grayhill* 072 keypad. Using the ***module . . . endmodule*** encapsulation and port given below, write a Verilog module, *Decoder_L*, of a functional unit that forms the left and right (active-low) codes of two seven-segment displays. The MSB of *Left_out* and *Right_out* must be mapped to the "*a*" segment of the display, and the LSB must be mapped to the "*g*" segment of the string "abcdefg."

```
module Decoder_L (output [6: 0] Left_out, Right_out, input [3: 0]
  Code_in); // active low displays
  ...
endmodule
```

Write a test plan specifying how *Decoder_L* is to be tested. Using the test plan, write a testbench, *t_Decoder_L*, that verifies the functionality of *Decoder_L*, and execute the test plan.

Design—Display Units

The next objective is to develop functional units supporting the reading and displaying of the contents of the FIFO on the seven-segment displays of the prototyping board. The seven-segment displays on the Digilab prototyping board have a common anode. Each unit has seven cathode pins, corresponding to the segments string "abcdefg." We wish to implement the structure shown in Figure P8-11c below. The output of the FIFO will be decoded to form the active-low code of the left and right displays. A mux will select between the two codes and route them to the appropriate segment simultaneously with an assertion of the appropriate

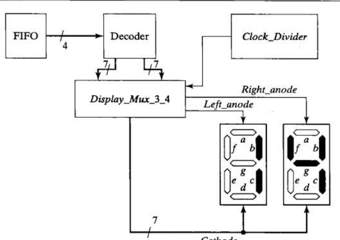


FIGURE P8-11c Seven-segments display architecture for the FIFO readout.

anode. A clock divider must be used to strobe the displays at a frequency that is high enough to eliminate the flicker effect (i.e., at a frequency above the bandwidth of the human eye) and low enough to be displayed by the LED.

The FIFO module will read its data on the active edge of the clock while the read input signal is asserted. The prototyping boards may have clocks that are running at 25 or 50 MHz, depending on the model. In either case, a single push of the button would cause the entire content of the FIFO to be dumped before the button could be released. Therefore, a machine must be designed to accept the button signal and assert a read signal at the FIFO for only one clock signal. The button must be de-asserted before another read can occur.

Design—Clock Divider

Now we will design a parameterized clock divider that can be used to strobe the mux controlling the seven-segment displays and to operate the system at a suitable frequency. Using the ***module. . . endmodule*** encapsulation below, develop a Verilog module of *Clock_Divider*, a parameterized clock divider having a default division by 2^{24}.

```
module Clock_Divider (output clk_out, input clk_in, reset);
...
endmodule
```

Design — Asynchronous User Interface

Two additional units must be designed: (1) a synchronizer for the "read" signal controlling the FIFO, and (2) a toggle unit that allows only one cell of the FIFO to be read at a time. Using the ***module. . . endmodule*** encapsulation below, develop a Verilog module of *Synchro_2*, a two-stage synchronizer for the signal that reads the FIFO. The output of *Synchro_2* should be synchronized to the negative edge of *clk*, because the state machine is active on the positive edge.

```
module Synchro_2 (output synchro_out, input synchro_in, clk, reset);
...
endmodule
```

Using the ***module. . . endmodule*** encapsulation below, develop a Verilog module of *Toggle* a state machine that accepts the synchronized signal directing that the FIFO be read, and asserts a signal that reads only one cell of the FIFO, regardless of whether the user holds the "read" button for more than one clock cycle or not. The machine is to ensure that only one cell of the FIFO is read each time the button is pushed.

```
module Toggle (output read_fifo, read_synch, input clk, reset);
...
endmodule
```

Write a test plan specifying how *Toggle* is to be tested. Using the test plan, write a testbench, *t_Toggle*, that verifies the functionality of *Toggle*, and that it operates correctly with the FIFO. Execute the test plan.

We will now design a multiplexer to control the common-anode, seven-segment displays of the Digilab prototyping board. Using the ***module. . . endmodule*** encapsulation below, develop a Verilog module of ***Display_Mux_3_4***, a functional unit that selects between two cathode codes and asserts the selected code at its output, and also asserts the appropriate anode of the rightmost pair of seven-segment displays on the board.

```
module Display_Mux_3_4 (output [7: 0] Cathode, output Left_anode,
  Right_anode,
input [7; 0] Display_3, Display_4, sel
);
...
endmodule
```

Write and execute a test plan for verifying *Display_Mux_3_4*.

The following objectives remain: (1) integrate the functional units of the FIFO keypad system that were designed above, (2) synthesize the integrated system and implement it in an FPGA, and (3) conduct a hardware verification of the working system.

Design—System Integration

We will integrate the previously designed and verified functional units, and verify that the integrated system functions correctly. Using the top-level encapsulating module *TOP_Keypad_FIFO* given below, form the integrated system. Using a simulator, eliminate any syntax errors from the integration. Pay careful attention to the mapping of formal and actual port names.

```
module TOP_Keypad_FIFO (
  output [6: 0]   Cathode,
  output [3: 0]   Col,
  output          Left_anode, Right_anode,
  output          empty, full,
  input  [3: 0]   Row,
  input           read,
  input           clk, reset
);
  wire   [3: 0]   Code, Code_out;
  wire            S_Row;
  wire            valid;
  wire   [6: 0]   Left_out, Right_out;
  wire            clk_slow, clk_display;
  wire            read_fifo, read_synch;

   Synchronizer M0 (
        .S_Row(S_Row),
        .Row(Row),
        .clock(clk_slow),
        .reset(reset));

   Hex_Keypad_Grayhill_072 M1(
        .Code(Code),
        .Col(Col),
        .Valid(valid),
        .Row(Row),
        .S_Row(S_Row),
        .clock(clk_slow),
        .reset(reset));

   FIFO M2 (
        .Data_out(Code_out),
        .Data_in(Code),
        .stack_empty(empty),
        .stack_full(full),
        .clk(clk_slow),
        .rst(reset),
        .write_to_stack(valid),
        .read_from_stack(read_fifo));

   Decoder_L M3 (
        .Left_out(Left_out),
```

```
            .Right_out(Right_out),
            .Code_in(Code_out));

    Display_Mux_3_4 M5 (
            .Cathode(Cathode),
            .Left_anode(Left_anode),
            .Right_anode(Right_anode),
            .Display_3(Left_out),
            .Display_4(Right_out),
            .sel(clk_display));

    Clock_Divider #(7) M6 (             // DEFAULT WIDTH = 24
            .clk_out(clk_slow),         // Use 20 for slow/visible operation
            .clk_in(clk),
            .reset(reset));

    Clock_Divider #(20) M7 (
            .clk_out(clk_display),
            .clk_in(clk),
            .reset(reset));

    Toggle M8 (
            .toggle_out (read_fifo),
            .toggle_in (read_synch),
            .clk(clk_slow),
            .reset(reset));

    Synchro_2 M9 (
            .synchro_out(read_synch),
            .synchro_in(read),
            .clk(clk_slow),
            .reset(reset));

endmodule

module Row_Signal (
  output reg        [3: 0]    Row,
  input             [15: 0]   Key,
  input             [3: 0]    Col
);

// Scan for row of the asserted key
always @ (Key, Col) begin //Asynchronous behavior for key assertion
   Row[0] = Key[0] && Col[0] || Key[1] && Col[1] || Key[2] && Col[2] ||
     Key[3] && Col[3];
   Row[1] = Key[4] && Col[0] || Key[5] && Col[1] || Key[6] && Col[2] ||
     Key[7] && Col[3];
   Row[2] = Key[8] && Col[0] || Key[9] && Col[1] || Key[10] && Col[2] ||
     Key[11] && Col[3];
   Row[3] = Key[12] && Col[0] || Key[13] && Col[1] || Key[14] && Col[2] ||
     Key[15] && Col[3];
 end
endmodule
```

Using the testbench modules given below, verify the functionality of the integrated system. Pay careful attention to the formation of the graphic user interface (GUI) displaying waveforms to display results in a user-friendly format.

```
module t_TOP_keypad_FIFO ();
 wire    [6: 0]   Cathode;
 wire    [3: 0]   Col;
 wire             Left_anode, Right_anode;
 wire             valid;
 wire             empty;
 wire             full;
 wire    [3: 0]   Row;
 reg              read;
 reg              clock, reset;
 reg     [15: 0]  Key;
 integer          j, k;
 reg     [39: 0]  Pressed;
 parameter                [39: 0] Key_0 = "Key_0";
 parameter                [39: 0] Key_1 = "Key_1";
 parameter                [39: 0] Key_2 = "Key_2";
 parameter                [39: 0] Key_3 = "Key_3";
 parameter                [39: 0] Key_4 = "Key_4";
 parameter                [39: 0] Key_5 = "Key_5";
 parameter                [39: 0] Key_6 = "Key_6";
 parameter                [39: 0] Key_7 = "Key_7";
 parameter                [39: 0] Key_8 = "Key_8";
 parameter                [39: 0] Key_9 = "Key_9";
 parameter                [39: 0] Key_A = "Key_A";
 parameter                [39: 0] Key_B = "Key_B";
 parameter                [39: 0] Key_C = "Key_C";
 parameter                [39: 0] Key_D = "Key_D";
 parameter                [39: 0] Key_E = "Key_E";
 parameter                [39: 0] Key_F = "Key_F";
 parameter                [39: 0] None = "None";
/*
 wire stack0 = UUT.M2.stack[0];              // Probes of the stack
 wire stack1 = UUT.M2.stack[1];
 wire stack2 = UUT.M2.stack[2];
 wire stack3 = UUT.M2.stack[3];
 wire stack4 = UUT.M2.stack[4];
 wire stack5 = UUT.M2.stack[5];
 wire stack6 = UUT.M2.stack[6];
 wire stack7 = UUT.M2.stack[7];

*/
 always @ (Key) begin
  case (Key)
   16'h0000:    Pressed = None;
   16'h0001:    Pressed = Key_0;
```

```
        16'h0002:     Pressed = Key_1;
        16'h0004:     Pressed = Key_2;
        16'h0008:     Pressed = Key_3;
        16'h0010:     Pressed = Key_4;
        16'h0020:     Pressed = Key_5;
        16'h0040:     Pressed = Key_6;
        16'h0080:     Pressed = Key_7;
        16'h0100:     Pressed = Key_8;
        16'h0200:     Pressed = Key_9;
        16'h0400:     Pressed = Key_A;
        16'h0800:     Pressed = Key_B;
        16'h1000:     Pressed = Key_C;
        16'h2000:     Pressed = Key_D;
        16'h4000:     Pressed = Key_E;
        16'h8000:     Pressed = Key_F;
        default:      Pressed = None;
      endcase
    end
  TOP_Keypad_FIFO M_UUT
  (Cathode, Col, Left_anode, Right_anode, empty, full, Row, read, clock,
    reset);
  Row_Signal M2 (Row, Key, Col);
   initial #42000 $finish;
   initial begin clock = 0; forever #5 clock = ~clock; end
   initial begin reset = 1; #10 reset = 0; end
   initial begin for (k = 0; k <= 1; k = k+1) begin Key = 0; #25 for (j = 0; j
    <= 16; j = j+1) begin
    #67Key[j] = 1; #160 Key = 0; end end end
   initial begin forever begin
     #307 read = 1;
     #20 read = 0;
    end
  end
 endmodule
```

Design—Prototype Synthesis and Implementation

Synthesize the integrated system and target the design into the FPGA for the available prototyping board. Download the bitmap file into the FPGA and conduct a demonstration of the functionality of the system.

12. Modify the keypad scanner circuit from the previous exercises to incorporate protection against switch bounce. Consider requiring a switched input to hold its value for a sufficiently long time before the machine accepts the input (e.g., 20 ms). Write and verify a Verilog model of the modified circuit. Synthesize the modified circuit and verify that the debounce circuitry works on the prototyping board.

13. FPGA-Based Design Exercise: Serial Communications Link with Error Correction

The objective of this exercise is to implement a serial communication link between a pair of FPGA prototyping boards and to demonstrate the functionality of an error correction unit. The UART that was presented in Chapter 7 is to be used, and include an extended Hamming encoder and an extended Hamming decoder. The block diagram in Figure P8-13(a) shows the configuration of the transmitter board and the receiver board. At the transmitter board, a sender interacts with the FPGA by pressing a key of the keypad. The keypad decoder, together with a two-stage synchronizer, produces a 4-bit code corresponding to the pressed key.[27] The Hamming encoder accepts a 4-bit input and generates an 8-bit encoded output word; a pair of push buttons can be used to deliberately inject errors into the code, for subsequent decoding and error correction at the receiver board. The UART transmitter will send the output of the Error Injection Unit.

The Error Injection Unit is to be hardwired to corrupt bits 1 and 5, depending on whether the push-button switches are pressed. The data stream bits can be *XOR*-ed with the logic value presented by the condition of the push-button switch. When not pressed, the switch presents a logical 0. The selected bits allow a data bit and a parity bit to be corrupted.

The Hamming decoder at the receiver board accepts an 8-bit word and forms a 4-bit output word. The unit is to be implemented with combinational logic and operate fast enough to form its output in a single cycle of the clock. The decoder is to detect and correct a single-bit error and display the corrected data word on the seven-segment displays. An LED will also be illuminated to indicate the occurrence of such an error. The extended Hamming code includes an additional bit to allow the decoder to detect, but not correct, the occurrence of a double-bit error. Such a condition will be indicated by illumination of another LED.

The Hex keypad interface is to form a unique code for each pressed key. A clock divider forms additional clock signals from the board's nominal 50 MHz clock signal. The information presented to the set of seven-segment displays will have to be time-multiplexed. The clock signals to be used in the design are shown in Table P8-13a.

[27] See Sections 5.16 and 5.17.

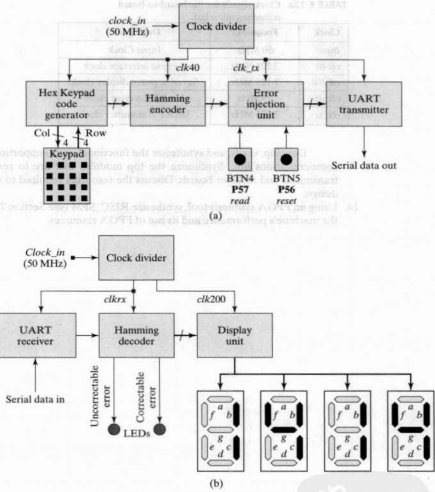


FIGURE P8-13 Serial communication link between prototyping boards: (a) transmitter unit and (b) receiver unit.

TABLE 8-13a Clock signals for the board-to-board communication link.

Clock	Frequency	Description
Input	50 MHz	Input Clock
clk40	3.125 NMz	Keypad interface clock
clk200	3.125 MHz	Seven-segment display clock
clk_rx	25 MHz	UART Receiver clock
clk_tx	3.125 MHz	UART Transmitter clock

Develop, verify, and synthesize the functional units supporting the serial communications link. Synthesize the top modules that are to reside on the transmitter and receiver boards. Discuss the resources required to support the design.

14. Using an FPGA synthesis tool, synthesize RISC_SPM (see Section 7.3). Discuss the machine's performance and its use of FPGA resources.

CHAPTER 9

Algorithms and Architectures for Digital Processors

An algorithm is a sequence of processing steps that create and/or transform data objects in memory. A general-purpose machine, or processor, can be programmed (via a high-level language or assembly language) to execute a variety of algorithms, but its architecture might not yield the highest performance for a particular application, might be underused by some applications, and might not have a good balance between the processor's speed and its input–output (I/O) throughput over the domain of application [1]. Compared to dedicated application-specific integrated circuits (ASICs), a general-performance processor might consume more power, require more area, and have a higher unit cost (depending on volume of sales). Dedicated processors will have simpler instruction sets and simpler microcode.

ASICs are designed to optimize the execution of particular algorithms for a specific application. The hardwired, customized architecture of an ASIC achieves a performance/cost trade-off that weighs in favor of an ASIC chip, rather than a general-purpose processor, in a specific application.

ASIC chips sacrifice flexibility for performance, because their architecture is fixed. Field-programmable gate arrays (FPGAs), at least in principle, can be configured to execute any algorithm. With ASICs and FPGAs, the architecture that implements an algorithm is an important consideration. ASICs are especially suited for applications in digital signal processing (DSP), image processing, and data communications, in which parallel datapaths and concurrent processing abound. We saw in Chapter 8 that FPGAs can be configured for a variety of applications, and reconfigured repeatedly. The choice between an FPGA and an ASIC-based implementation is often determined by a

bottom-line unit cost, but in some cases, the choice depends on the relative performance offered by an ASIC versus that of an FPGA in the application.

A processor can be viewed as an array, or architecture, of elemental function processors or functional units (FUs), which, as a network, realize an algorithm by executing their individual tasks in a coordinated, synchronized manner. For example, an architecture of adders, multipliers, and registers, together with other logic, might implement the algorithm of a lowpass digital filter. High-level design is concerned with implementing an architecture that realizes an algorithm to accomplish in a hardwired architecture what would be accomplished by executing a program on a general-purpose processor.

High-level design accomplishes two primary tasks: (1) it constructs an algorithm that realizes a behavioral specification (e.g., create a lowpass filter with given performance characteristics) and (2) it maps the algorithm into an architecture (i.e., a structure of FUs) that implements the behavior in hardware. The high-level design space is complex because alternative algorithms may exhibit the same behavior, and because multiple architectures may implement a given algorithm, possibly with different throughput and latency. In this chapter, we will consider the overall design process that leads to an application-specific architecture, and we will assume that the high-level design task has been accomplished (i.e., our starting point will be a computational algorithm that must be implemented by a host processor). We will focus on (1) developing an algorithm processor (i.e., a fixed architecture that implements a given algorithm), (2) exploring architectural tradeoffs (both for a network of FUs and for fine-grained implementations of the FUs themselves), (3) developing Verilog descriptions of the architectures, and (4) synthesizing the architectures.[1]

9.1 Algorithms, Nested-Loop Programs, and Data Flow Graphs

Algorithmic processors are composed of FUs, each executing in an environment of coordinated data flow. A sequential algorithm can be described by a nested-loop program (NLP) [1,2], which consists of a set of nested *for* loops, as depicted in Figure 9-1, and a loop body written in a programming language such as C, in a pseudo language, or in a hardware description language (HDL) such as Verilog (i.e., a cyclic behavior).[2] NLPs are always computable, so such a starting point for a realization of a specification is attractive. Moreover, an NLP provides an unambiguous and executable specification for the machine whose behavior is to be realized.

The sequential ordering of operations and the dependencies of data in an NLP for a given algorithm can be represented by a *data flow graph* (DFG) [3,4]. A DFG can be developed manually, and a language parser can extract a DFG from an NLP or from

[1] Performance issues will also be addressed in Chapter 11.

[2] A generic *for* loop executes repeatedly under the control of lower and upper bound expressions and a loop index mechanism. The Verilog ***for*** loop is an example.

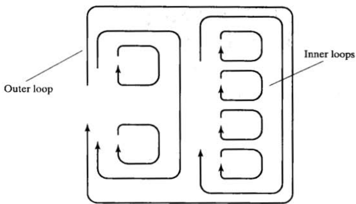


FIGURE 9-1 A nested-loop program consists of a set of nested *for* loops.

an HDL-based behavioral description by analyzing the activity flow of the statements and the lifetime of the variables, together with the semantics of the language's constructs. We will see that a DFG is a key tool in developing an architecture for an algorithmic processor and for exploring alternative equivalent architectures.

A DFG is a *directed acyclic graph*, $G(V, E)$, where V is the set of nodes of the graph, and E is the set of edges of the graph [3]. Each node $v_i \in V$ represents a FU, which operates on its inputs (data) to produce its outputs. An FU might consist of a single operation or a more complex, ordered composition of operations, in which case the FU itself might be represented by a DFG giving a fine-grained view. Each directed edge $e_{ij} \in E$ originates at a node $v_i \in V$ and terminates at node $v_j \in V$. Given an edge $e_{ij} \in E$, the data produced by node v_i is consumed by node v_j [5].

A data dependency exists between a pair of nodes ($v_i \in V$, $v_j \in V$) corresponding to edge e_{ij} if the FU v_j uses the results of FU v_i, and if the operation of v_j cannot begin until the operation of v_i has finished (i.e., the directed edges of the graph imply a precedence for execution).[3] Thus, the DFG reveals the producers of data, the order in which the data will be generated, and the consumers of data. It also exposes parallelism in the dataflow, which reveals opportunities for concurrent computation and has implications for the lifetime of variables (i.e., memory). In a synchronous dataflow a node consumes its data before new data are presented [5]. The designer's task, in general, is to transform the DFG for an algorithm into a structure of hardware, typically a partitioned structure consisting of a datapath unit and a control unit, as represented by an algorithmic state machine and datapath (ASMD) chart.[4] Given that the control unit

[3]The edges of the graph can be annotated to indicate a time delay (e.g., a register buffer delay) that must elapse before the node receiving data from a provider can commence execution.

[4]See Chapter 7.

must control the datapath, the initial task is to specify an architecture for a datapath unit which, if controlled to respect the constraints implied by the DFG, will implement the algorithm. Then the control unit can be designed to coordinate the data flow of the algorithm.[5]

Beginning with the DFG, the high-level synthesis task of *datapath allocation* consists of transforming the DFG of an algorithm into an architecture of processors, datapaths, and registers from which a synthesizable register transfer level (RTL) model can be developed in Verilog or VHDL.[6] A baseline architecture that implements a given DFG can always be formed as a set of FUs connected in a structure that is isomorphic to the DFG [18].[7] This design is hardware-intensive, and it serves only as a starting point because other architectures may realize the same algorithm with higher performance and less hardware. Datapath allocation binds the FUs of the DFG to a given (selected) set of datapath resources, and schedules their use of the resources.

Many different architectures can implement the same algorithm. They will be distinguished not only by their datapath resources, but also by a temporal schedule for using the resources. The high-level task of *resource scheduling* assigns resources and a time slot to each node of a DFG. Given the parallelism of a DFG, there are many schedules that could map nodes into time slots (control steps), creating several alternatives for realizing the corresponding algorithm in hardware.[8] Scheduling must be conflict-free (i.e., a resource cannot be allocated to multiple FUs in the same time slot). The overall design flow is shown in Figure 9-2.

Three general approaches are used to reorganize the baseline architecture obtained from the DFG: recomposition, pipelining, and replication [18]. *Recomposition* segments the FU into a sequence of functions that execute one after the other to implement the algorithm. The sequence of execution may be further distributed over space (hardware units) and time. In the former case, the nodes of the DFG are mapped isomorphically to the FUs; in the latter case, a single FU executes over as many clock cycles as required to complete the operations represented by the DFG. This approach saves hardware by replicating the activity of a single FU over multiple time steps, rather than using multiple processors that execute concurrently in a single step. *Pipelining* inserts registers into a datapath to shorten computational paths and thereby increase the throughput of a system, incurring a penalty in latency and the number of registers. In contrast to recomposition, *replication* uses multiple, identical, concurrently executing

[5]We will rely on ASM charts to design a controller; control-flow data graphs and control-data flow graphs can also be used [5].

[6]Design tools are now available to automate these steps – manual retranslation is potentially a source of error in the design flow, because the behavior of the machine synthesized from the RTL model might not match the intended behavior expressed by the specification. See high level synthesis at, for example, www.mentor.com.

[7]I am grateful to Dr. Hubert Kaeslin, of the Swiss Federal Institute of Technology (ETH), Zurich for correspondence about this approach to design.

[8]See references [3] and [5] for a discussion of various scheduling algorithms that are used in high-level synthesis.

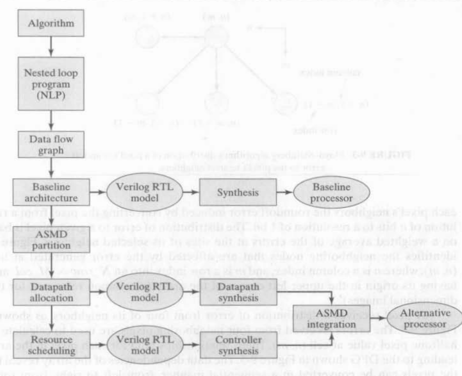


FIGURE 9-2 Design flow for algorithm-based synthesis of a sequential machine.

processors to improve performance, but at the expense of hardware. For an in-depth treatment of these approaches, see [18].

9.2 Design Example: Halftone Pixel Image Converter

As an example of how an architecture can be synthesized from an algorithm, we will demonstrate the main steps of synthesizing a halftone image converter. The circuit has been used elsewhere [1] to illustrate concepts in automated synthesis of algorithms for DSP and will be our platform for developing and comparing architectures for an image converter. We will design a baseline machine and then consider an alternative machine that uses less hardware, but executes over multiple clock cycles.

The so-called Floyd–Steinberg algorithm converts an image consisting of an *N_col* × *M_row* array of pixels, each having a resolution of *pixel_size* pixel_sizebits, into an array having only black or white pixels, while incorporating a subjective measure of the quality of the image [1]. The algorithm distributes to a selected subset of

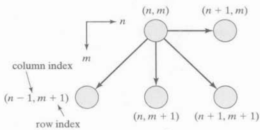


FIGURE 9-3 Floyd–Steinberg algorithm's distribution of a pixel's roundoff error to the pixel's nearest neighbors.

each pixel's neighbors the roundoff error induced by converting the pixel from a resolution of n bits to a resolution of 1 bit. The distribution of error to a given pixel is based on a weighted average of the errors at the sites of its selected neighbors. Figure 9-3 identifies the neighboring nodes that are affected by the error generated at node (n, m), where n is a column index, and m is a row index into an $N_row \times M_col$ array having its origin in the upper left corner of the array (a common reference for two-dimensional images).

A pixel receives a distribution of error from four of its neighbors, as shown in Figure 9-4. The errors received from four neighboring pixels are used to calculate the halftone pixel value at cell (n, m). These relationships hold for each pixel in the array, leading to the DFG shown in Figure 9-5. The data dependencies of the array reveal that the pixels can be converted in a sequential manner, from left to right, from top to bottom, beginning with producer of data at the top left corner and proceeding to the consumer at the bottom right corner.

The FUs (nodes) of the DFG execute the pixel conversion, according to the following pseudocode description of the sequence of calculations that will ultimately be described by a cyclic behavior in Verilog. At each pixel location, (n, m), a weighted average of the (previously calculated) error, e, at the selected neighbors is formed according to

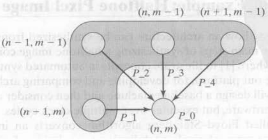


FIGURE 9-4 Nearest neighbors for updating a pixel in a halftone image converter based on the Floyd–Steinberg algorithm.

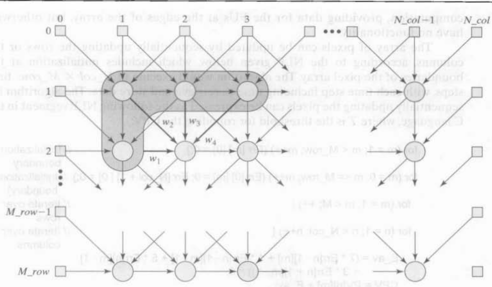


FIGURE 9-5 Data flow graph for a $N \times M$ halftone pixel image converter.

$$E_av = (w1 * e[n-1, m] + w2 * e[n-1, m-1] + w3 * e[n, m-1] + w4 * e[n+1, m-1])/w_T$$

where $w1, \ldots, w4$ are subjective nonnegative weights, and $w_T = w1 + w2 + w3 + w4$. This weighted average is used to calculate a corrected pixel value (CPV):

$$CPV = PV[n, m] + E_av$$

and then we round CPV to 0 or 1, according to a threshold, CPV_thresh. So

$$CPV_round = 0 \text{ if } CPV < CPV_thresh$$

and

$$CPV_round = CPV_max, \text{ otherwise}$$

where $CPV_max = 255$ and $CPV_thresh = 128$, for an image with 8-bit resolution. Next, we form a halftone pixel value and save the error:

$$HTPV = 0 \text{ if } CPV_round = 0, \text{ otherwise } HTPV = 1;$$
$$Err[n, m] = CPV - HTPV$$

The main array of nodes in Figure 9-5 is isomorphic to the array of pixels. We have added a column of boundary nodes at the left and right edges of the graph and a row at the top of the graph. These additional nodes are used to initialize the

computation, providing data for the FUs at the edges of the array, but otherwise have no functionality.

The array of pixels can be updated by sequentially updating the rows or the columns, according to the NLP given below, which includes initialization at the boundaries of the pixel array. The algorithm would execute in $N_col \times M_row$ time steps, with each time step including time to retrieve and store data. The algorithm for sequentially updating the pixels can be expressed as the following NLP segment in the C language, where T is the threshold for rounding the CPV:

```
for (m = 1; m < M_row; m++) {Err [m] [0] = 0;}                        // Initialization of
                                                                         boundary
for (m = 0; m <= M_row; m++) {Err [0] [m] = 0; Err [N_col + 1] [ 0] = 0;} // Initialization of
                                                                         boundary)
 for (m = 1; m < M; ++) {                                               // Iterate over
                                                                         rows
 for (n = 1; n < N_col; n++) {                                          // Iterate over
                                                                         columns
   E_av = (7 * Err[n−1][m] + 1 * Err[n−1][m−1] + 5 * Err[n][m−1]
          + 3 * Err[n + 1][m−1]) /16;
   CPV = PV[n][m] + E_av;
   CPV_round = (if CPV < T then 0 else 255); // Threshold = 128;
   HTPV [n][m] if (CPV_round == 0 then 0 else 1);
   Err [n][m] = CPV - CPV_round;
  }
 }
```

9.2.1 Baseline Design for a Halftone Pixel Image Converter

There are various options for forming a hardware implementation of an NLP realizing the image conversion algorithm. Since the computations at each FU are combinational in nature, the simplest architecture is a structure of FUs that is isomorphic to the DFG, with input data consisting of the array of pixel values, and output data consisting of the halftone pixel and error values at each location. The Verilog description of the FU, referred to as the pixel processor datapath unit *PPDU*, and *Image_Converter_Baseline*, are given below.[9] The model is hardware-intensive and structural, consisting of a systolic array[10] of 48 identical processors hard-wired for the dataflow of the DFG. T_{Baseline}, the cycle time of the host processor providing images to the pixel processor, will be limited by the longest path through the array, from the top-left pixel to the bottom-right pixel.[11] The implementation is parameterized, portable and synthesizable as combinational logic, and requires no controller. The input to the converter is a vector consisting

[9]The Verilog *generate* construct is used here to represent, elaborate, and connect structural elements of the model.
[10]A systolic array has a set of identical FUs, with high local connectivity and multiple data flows.
[11]This value is not evident in a zero-delay simulation.

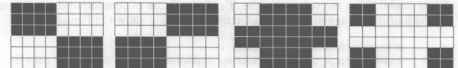


FIGURE 9-6 Sharp test images for the halftone image converter.

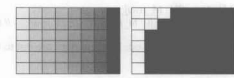


FIGURE 9-7 Graduated test image and halftone image produced by *Image_Converter_Baseline* with weights $(w1, w2, w3, w4) = (2, 8, 4, 2)$.

of the data words of the array of pixels, organized sequentially from the top left to the bottom right corner of the array. The testbench is also shown. It generates the array of pixels for a given pattern, and creates the vector of pixel data for the image converter. The testbench also extracts and displays the rows of pixel halftone values from the vector *HTPV_bits* which is produced by the converter.

The testbench produces the sharp contrast image patterns shown in Figure 9-6, and the graduated image shown in Figure 9-7. The halftone image of the graduated image is also shown in Figure 9-7. The simulation results shown in Figure 9-8 show that the halftone images and the graduated image produced by *Image_Converter_Baseline* match the corresponding sharp contrast images. To interpret the simulation results, note that *HTPV_Row*_1[1:8] ... *HTPV_Row*_3[1:8] having the value 8'hf0 and *HTPV_Row*_4[1:8] ... *HTPV_Row*_6[1:8] having the value 8'h0f in the first column in Figure 9-8 corresponds to the leftmost image in Figure 9-6, with 1 being the value of

Name	0	500	1000	1500	2000	2500 t
HTPV_Row_1[1:8]	f0	0f	3c	c3	1f	
HTPV_Row_2[1:8]	f0	0f	3c	c3	3f	
HTPV_Row_3[1:8]	f0	0f	ff	00	7f	
HTPV_Row_4[1:8]	0f	f0	ff	00	7f	
HTPV_Row_5[1:8]	0f	f0	3c	c3	7f	
HTPV_Row_6[1:8]	0f	f0	3c	c3	7f	

FIGURE 9-8 Simulation results for *Image_Converter_Baseline* with weights $(w1, w2, w3, w4) = (2, 8, 4, 2)$.

black, and 0 being the value of white. The last column of simulation data corresponds to the halftone image of the graduated image that was shown in Figure 9-7.

```
// Isomorphic array of processors
module Image_Converter_Baseline # (parameter pixel_size = 8, N_col = 8,
  M_row = 6)(
  output [1: N_col*M_row] HTPV_bits,
  input [1: pixel_size * N_col * M_row] pixel_bits
);

  wire HTPV [1: N_col][1: M_row];
  wire [pixel_size-1: 0] Err [0: N_col + 1] [0: M_row];    // Core and boundary values

// Initialize boundary values (Top, left, and right are set to 0)
  genvar n, m;
  generate
   for (n = 0; n <=N_col + 1; n = n + 1) begin: top_border assign Err[n][0] = 1'b0; end
   for (m = 1; m <= M_row; m = m + 1) begin: left_border assign Err[0][m] = 1'b0; end
   for (m = 1; m <= M_row; m = m + 1) begin: right_border assign Err[N_col + 1][m] =
    1'b0; end

// Instantiate array of pixel processors
  for (m = 1; m <= M_row; m = m + 1) begin: row_loop
   for (n = 1; n <= N_col; n = n + 1) begin: column_loop

    PPDU M (Err[n][m], HTPV[n][m], Err[n-1][m], Err[n-1][m-1], Err[n][m-1],
     Err[n + 1][m-1],
     pixel_bits[(m-1)*N_col*pixel_size +(n-1)*pixel_size +1: (m-1)*
      N_col*pixel_size + n*pixel_size]);
     end
    end
  endgenerate

// Pack bits into output vector
   generate
    for (m = 1; m <=M_row; m = m + 1) begin: HTPV_row_loop
     for (n = 1; n <= N_col; n = n + 1) begin: HTPV_col_loop
      assign HTPV_bits [(m-1)*N_col + n] = HTPV[n][m];
     end
    end
   endgenerate
  endmodule

// Pixel Processor Datapath Unit
module PPDU #(parameter pixel_size = 8)(
  output [pixel_size-1: 0] Err_0,
  output HTPV,
  input  [pixel_size-1: 0]Err_1, Err_2, Err_3, Err_4, PV
);
  wire   [pixel_size + 1: 0]CPV, CPV_round, E_av;
   // Weights for the average error; choose for compatibility with divide-by-16 (>> 4)
   parameter      w1 = 2, w2 = 8, w3 = 4, w4 = 2;
```

```
parameter      Threshold = 128;

assign E_av =          (w1 * Err_1 + w2 * Err_2 + w3 * Err_3 + w4 * Err_4 ) >> 4;
assign CPV =           PV + E_av;
assign CPV_round =     (CPV < Threshold) ? 0: 255;
assign HTPV =          (CPV_round == 0) ? 0: 1;
assign Err_0 =         CPV-CPV_round;
endmodule

module t_Image_Converter_Baseline();
 parameter pixel_size = 8, N_col = 8, M_row = 6;
 wire   [1: N_col]                        HTPV_Row_1, HTPV_Row_2,
                                          HTPV_Row_3;
 wire   [1: N_col]                        HTPV_Row_4, HTPV_Row_5,
                                          HTPV_Row_6;
 reg    [1: pixel_size*N_col*M_row]       pixel_bits;
 wire   [1: N_col*M_row]                  HTPV_bits;

 // Form rows of hafltone values
 assign HTPV_Row_1 = HTPV_bits[1:8];
 assign HTPV_Row_2 = HTPV_bits[9:16];
 assign HTPV_Row_3 = HTPV_bits[17:24];
 assign HTPV_Row_4 = HTPV_bits[25:32];
 assign HTPV_Row_5 = HTPV_bits[33:40];
 assign HTPV_Row_6 = HTPV_bits[41:48];

 Image_Converter_Baseline M1 (HTPV_bits, pixel_bits);     // Instantiate image
                                                             converter

initial fork
 begin: Image_Pattern_1
  pixel_bits = {  8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00,
                  8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00,
                  8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00,
                  8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff,
                  8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff,
                  8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff};
 end
 #500 begin: Image_Pattern_2
  pixel_bits = {  8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff,
                  8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff,
                  8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff,
                  8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00,
                  8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00,
                  8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00};
 end
 #1000 begin: Image_Pattern_3_Cross
  pixel_bits = {  8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h0,
                  8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h00,
                  8'hff, 8'hff, 8'hff, 8'hff, 8'hff, 8'hff, 8'hff, 8'hff,
                  8'hff, 8'hff, 8'hff, 8'hff, 8'hff, 8'hff, 8'hff, 8'hff,
```

```
                     8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h0,
                     8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h0};
   end

   #1500 begin: Image_Pattern_4_Bar_Cross
     pixel_bits = {  8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff,
                     8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff,
                     8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00,
                     8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00,
                     8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff,
                     8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff};
   end

   #2000 begin: Image_Pattern_5_Graduated_Left_to_Right
     pixel_bits = {  8'h1f, 8'h3f, 8'h5f, 8'h8f, 8'h9f, 8'hbf, 8'hdf, 8'hff,
                     8'h1f, 8'h3f, 8'h5f, 8'h8f, 8'h9f, 8'hbf, 8'hdf, 8'hff,
                     8'h1f, 8'h3f, 8'h5f, 8'h8f, 8'h9f, 8'hbf, 8'hdf, 8'hff,
                     8'h1f, 8'h3f, 8'h5f, 8'h8f, 8'h9f, 8'hbf, 8'hdf, 8'hff,
                     8'h1f, 8'h3f, 8'h5f, 8'h8f, 8'h9f, 8'hbf, 8'hdf, 8'hff,
                     8'h1f, 8'h3f, 8'h5f, 8'h8f, 8'h9f, 8'hbf, 8'hdf, 8'hff};
   end
  join
  initial begin #4000 $finish; end
endmodule
```

9.2.2 NLP-Based Architectures for the Halftone Pixel Image Converter

The baseline design of the halftone pixel image converter is at one extreme of the hardware-performance spectrum, for it replicates identical hardware (structural) units, each implemented as combinational logic blocks, which execute concurrently in a single long clock cycle (i.e., a single-loop NLP) and require no memory and no datapath controller. The structure of the array of processors in the baseline design is identical to the structure of the DFG, with a dedicated processor for each pixel of the array. The NLP itself suggests another alternative model: a structure-free, level-sensitive behavior that executes the NLP. This style is given below as *Image_Converter_0*. The model[12] synthesizes to combinational logic equivalent to *Image_Converter_Baseline*, and converts an image in time T_{Baseline}, where T_{Baseline} is the time associated with the longest datapath through the array. In both cases, the system clock that presents images to the unit must have a cycle time greater than T_{Baseline}.

[12] A location-dependent (i.e., (column and row dependent) word of pixel_size bits is selected from *pixel_bits* using the +: range delimiter.

```
// Behavioral model of isomorphic array of processors
module Image_Converter_0 # (parameter pixel_size = 8, N_col = 8, M_row = 6)(
  output reg [1: N_col*M_row] HTPV_bits,
  input [1: pixel_size*N_col*M_row] pixel_bits
);
  parameter        w1 = 2, w2 = 8, w3 = 4, w4 = 2;
  parameter        Threshold = 128;
  integer n, m;
  reg HTPV [1: N_col][1: M_row];
  reg [pixel_size -1: 0] Err [0: N_col +1] [0: M_row];      // Core and boundary values
  reg [pixel_size + 1: 0] CPV, CPV_round, E_av;

// Initialize boundary values (Top, left, and right are set to 0)
   always @ (pixel_bits) begin
    for (n = 0; n <=N_col+1; n = n + 1)        begin: top_border Err[n][0] = 1'b0; end
    for (m = 1; m <= M_row; m = m + 1)         begin: left_border Err[0][m] = 1'b0; end
    for (m = 1; m <= M_row; m = m + 1)         begin: right_border Err[N_col+1]
                                                 [m] = 1'b0; end

// Halftone calculations (Note use of blocking assignment operator (=)
    for (m = 1; m <= M_row; m = m + 1) begin: row_loop
     for (n = 1; n <= N_col; n = n + 1) begin: column_loop
      E_av = (w1*Err[n -1][m] + w2*Err[n -1][m -1] + w3*Err[n][m -1]
               + w4*Err[n + 1][m -1]) >> 4;
      CPV = pixel_bits[(m -1)*N_col*pixel_size +(n -1)*pixel_size + 1 +: pixel_size] +
        E_av;
      CPV_round = (CPV < Threshold) ? 0: 255;
      HTPV[n][m] = (CPV_round == 0) ? 0: 1;
      HTPV_bits [(m -1)*N_col + n] = HTPV[n][m];
      Err[n][m] = CPV - CPV_round;
    end          // column_loop
   end           // row_loop
  end            // always
endmodule
```

The NLP for the image converter also suggests a synchronous implementation. This architecture will require memory, but releases the resources of the system bus while the processor is executing. At one extreme, a design would use a single FU, together with memory and a controller, to convert the entire image in N_col *M_row clock cycles. The cycle time of the image converter would be limited by the longest path through a single FU, rather than the time to process the entire array. A (naive) alternative is to write a single synchronous cyclic behavior that describes the NLP of the algorithm (same algorithm as in *Image_Converter_0*) and let a synthesis tool create an architecture. The expectation is that there is no need to design a controller because a synthesis tool will synthesize the NLP by unrolling the loops and forming a structure that implements the algorithm. The machine would transform the image in one long clock cycle of length *T_Baseline*. A Verilog model of a

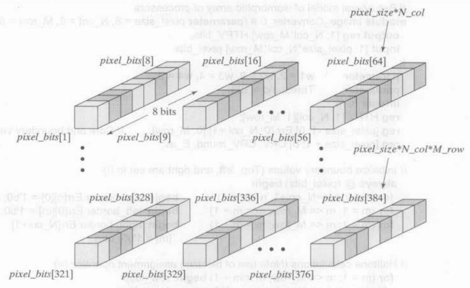


FIGURE 9-9 Structure of an $N \times M$ pixel array with an 8-bit resolution.

machine implementing the algorithm in this style is given by *Image_Converter*_1, below, along with its testbench.

*Image_Converter*_1 updates the entire array of halftone pixel values in one clock cycle. The rows of the array are processed top to bottom, and the elements of each row are processed left to right, as shown in Figure 9-9. The order of the statements reflects the data dependencies of the statements that must be executed by an FU, and the assignments to variables are made with the procedural assignment operator (=), not the nonblocking assignment operator (<=). (=)To illustrate the sequential ordering of the processing steps in simulation, a delay (#50) was placed before the column loop. This *reveals the evolution of the computations of the rows, but the delay must be removed before attempting synthesis*. The sequential ordering of the loops that update the rows of the array implements the NLP and convert the image in a single long clock cycle. A input signal, *Go*, is asserted by an external agent to launch the image converter, and an output signal, *Done*, is asserted when the image has been converted. The signal *reset* flushes the memory of residual values of pixel values and errors to prepare for a new image.

```
// Single-cycle sequential model (Non-synthesizable)

module Image_Converter_1 # (parameter pixel_size = 8, N_col = 8, M_row = 6)(
  output reg [1: N_col*M_row] HTPV_bits,
  output reg Done,
  input [1: pixel_size*N_col*M_row] pixel_bits,
  input Go, clk, reset
);
```

```
  parameter      w1 = 2, w2 = 8, w3 = 4, w4 = 2;
  parameter      Threshold = 128;
  integer n, m;
  reg HTPV [1: N_col][1: M_row];
  reg [pixel_size -1: 0] Err [0: N_col +1] [0: M_row];      // Core and boundary values
  reg [pixel_size + 1: 0] CPV, CPV_round, E_av;

// Initialize boundary values (Top, left, and right are set to 0)
  always begin: wrapper_for_synthesis
    @ (posedge clk) begin: pixel_converter
     if (reset) begin: reset_action    // Initialize borders of the array
      Done = 0;
      for (n = 0; n <=N_col+1; n = n + 1)      begin: top_border Err[n][0] = 1'b0; end
      for (m = 1; m <= M_row; m = m + 1)      begin: left_border Err[0][m] = 1'b0; end
      for (m = 1; m <= M_row; m = m + 1)      begin: right_border Err[N_col+1][m]
       = 1'b0; end
     end
     else if (Go) begin      // Halftone calculations – preserve sequential ordering
      for (m = 1; m <= M_row; m = m + 1) begin: row_loop
       #50 for (n = 1; n <= N_col; n = n + 1) begin: column_loop    // Delay only for
                                                                       illustration
        E_av = (w1 * Err[n-1][m] + w2 * Err[n-1][m-1] + w3 * Err[n][m-1]
                + w4 * Err[n + 1][m-1]) >> 4;
        CPV = pixel_bits[(m -1)*N_col*pixel_size +(n -1)*pixel_size + 1 +:
          pixel_size] + E_av;
        CPV_round = (CPV < Threshold) ? 0: 255;
        HTPV[n][m] = (CPV_round == 0) ? 0: 1;
        HTPV_bits [(m -1)*N_col + n] = HTPV[n][m];
        Err[n][m] = CPV - CPV_round;
      end        // column_loop
      // Used for Image_Converter_Work_Around (Temporal separation of row
        computations)
      // @ (posedge clk) if (reset) disable pixel_converter;
      end                  // row_loop
      Done = 1;
     end                   // Halftone calculations
    end                    // pixel_converter
  end                      // wrapper_for_synthesis
endmodule

module t_Image_Converter_1();
  parameter pixel_size = 8, N_col = 8, M_row = 6;
  wire    [1: N_col]                     HTPV_Row_1, HTPV_Row_2,
                                           HTPV_Row_3;
  wire    [1: N_col]                     HTPV_Row_4, HTPV_Row_5,
                                           HTPV_Row_6;
  reg     [1: pixel_size*N_col*M_row]    pixel_bits;
  reg     Go, clk, reset;
  wire    [1: N_col*M_row]               HTPV_bits;
  wire    Done;
```

```
assign HTPV_Row_1 = HTPV_bits[1:8];
assign HTPV_Row_2 = HTPV_bits[9:16];
assign HTPV_Row_3 = HTPV_bits[17:24];
assign HTPV_Row_4 = HTPV_bits[25:32];
assign HTPV_Row_5 = HTPV_bits[33:40];
assign HTPV_Row_6 = HTPV_bits[41:48];

Image_Converter_1 M1 (HTPV_bits, Done, pixel_bits, Go, clk, reset); // Instantiate
                                                                       image converter

initial begin clk =0; forever #5 clk = ~clk; end

initial fork
 #0 reset= 1;
 #10 reset = 0;
join

initial fork
 #15 Go = 1;              // Image #1
 #35 Go = 0;

 #480 reset = 1;          // Image #2
 #490 reset = 0;
 #500 Go = 1;
 #520 Go = 0;

 #980 reset = 1;          // Image #3
 #990 reset = 0;
 #1000 Go = 1;
 #1020 Go = 0;

 #1480 reset = 1;         // Image #4
 #1490 reset = 0;
 #1500 Go = 1;
 #1520 Go = 0;

 #1980 reset = 1;         // Image #5
 #1990 reset = 0;
 #2000 Go = 1;
 #2020 Go = 0;
join

initial fork
 begin: Image_Pattern_1
  pixel_bits = {  8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00,
                  8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00,
                  8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00,
                  8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff,
                  8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff,
                  8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff};
 end
```

```
#500 begin: Image_Pattern_2
  pixel_bits = {  8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff,
                  8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff,
                  8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff,
                  8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00,
                  8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00,
                  8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00};
end

#1000 begin: Image_Pattern_3_Cross
  pixel_bits = {  8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h0,
                  8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h00,
                  8'hff, 8'hff, 8'hff, 8'hff, 8'hff, 8'hff, 8'hff, 8'hff,
                  8'hff, 8'hff, 8'hff, 8'hff, 8'hff, 8'hff, 8'hff, 8'hff,
                  8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h0,
                  8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h0};
end

#1500 begin: Image_Pattern_4_Bar_Cross
  pixel_bits = {  8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff,
                  8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff,
                  8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00,
                  8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00,
                  8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff,
                  8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff};
end

#2000 begin: Image_Pattern_5_Graduated_Left_to_Right
  pixel_bits = {  8'h1f, 8'h3f, 8'h5f, 8'h8f, 8'h9f, 8'hbf, 8'hdf, 8'hff,
                  8'h1f, 8'h3f, 8'h5f, 8'h8f, 8'h9f, 8'hbf, 8'hdf, 8'hff,
                  8'h1f, 8'h3f, 8'h5f, 8'h8f, 8'h9f, 8'hbf, 8'hdf, 8'hff,
                  8'h1f, 8'h3f, 8'h5f, 8'h8f, 8'h9f, 8'hbf, 8'hdf, 8'hff,
                  8'h1f, 8'h3f, 8'h5f, 8'h8f, 8'h9f, 8'hbf, 8'hdf, 8'hff,
                  8'h1f, 8'h3f, 8'h5f, 8'h8f, 8'h9f, 8'hbf, 8'hdf, 8'hff};
end
join
initial begin #4000 $finish; end
endmodule
```

The results of simulating *Image_Converter_*1 with delay between the rows are shown in Figure 9-10. The sequential evolution of the processing of rows from top to bottom is evident. The final halftone image is identical to that produced by *Image_Converter_Baseline and Image_Converter_*0.

It is important to note that *Image_Converter_Baseline* and *Image_converter_*0 synthesize to equivalent combinational logic, but the sequential machine *Image_Converter_*1 cannot be synthesized, even though it uses the same algorithm to update the pixels. The sequential ordering of its procedural assignments in the cyclic behavior updates the values stored in memory as the simulation evolves by immediately overwriting the residual data, ensuring that fresh error data are used at subsequent steps.

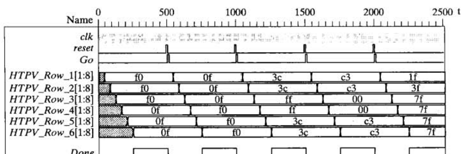


FIGURE 9-10 Simulation results for *Image_Converter_1* with weights ($w1$, $w2$, $w3$, $w4$) = (2, 8, 4, 2).

In the algorithm for *Image_Converter_1*, the storage for *CPV*, *CPV_round*, and *E_av* is shared between the rows. A physical machine would have to store data in memory and fetch it when needed. These operations cannot execute in a single clock cycle, which explains why the model cannot be synthesized. As a work-around, *Image_Converter_Work_Around* is identical to *Image_Converter_1*, except that it provides temporal separation of the processing of the rows of the array by embedding an event control expression at the end of the inner (column) loop. In general, feedback loops in the DFG must be separated temporally. This ensures that data used in a row is not overwritten in memory before it is used by the next row. *Image_Converter_Work_Around* updates one row of the array in a single clock cycle and requires eight processors (one for each pixel in the row). The machine shares its resources between the rows, but in a given clock cycle all of the processors are dedicated to a single row of pixels. Since *Image_Converter_1* cannot be synthesized, it is not appropriate to associate a cycle time with its computations. *Image_Converter_Work_Around* will require an image cycle time of $T_c = 6 \times 8 \times T_{FU} = 48\, T_{FU}$ (i.e., the machine saves hardware (i.e., uses fewer processors) compared to *Image_Converter_0*, but not time). It will require a more elaborate control structure to steer data to and from memory and the shared FUs. Unfortunately, the FPGA synthesis tool[13] did not support embedded event-control expressions within a ***for*** loop and could not produce an implementation.

9.2.3 Minimum Concurrent Processor Architecture for a Halftone Pixel Image Converter

We will now consider an alternative hardware implementation of the image converter algorithm by partitioning the image converter into an algorithmic state machine with a

[13]Xilinx ISE 3.1i.

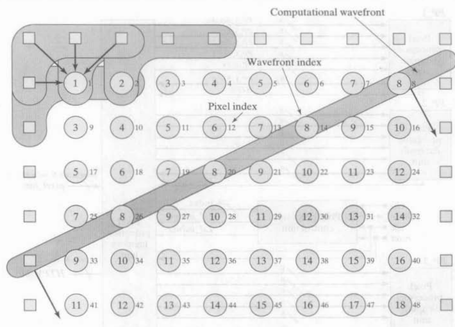


FIGURE 9-11 Locus of concurrently executing functional units in the DFG for an 8 × 6 halftone pixel image converter.

datapath (i.e., a ASMD). This sets the stage for discovering a configuration of concurrent processors that will process the image in the least multiple of the time step of a single processor, i.e., the configuration is optimal. It is maximally concurrent without wasting resources and clock cycles.

The data dependencies that are evident in the DFG shown in Figure 9-11 reveal a parallelism that can be exploited to reduce the number of clock cycles and the number of processors required to update the array, at the additional expense of requiring memory. The shaded region defines a *computational wavefront* (i.e., a locus of DFG nodes that can execute concurrently and independently in a given time step). Each node in Figure 9-11 is annotated with a *wavefront index* denoting the time step in which it may execute. The array of nodes can still be processed sequentially, in the order of ascending indexes, but nodes with an identical wavefront index can execute concurrently (i.e., in the same time step).

The wavefront indices of the DFG partition the temporal domain and identify clock boundaries. Within a clock boundary, the graph reveals the resources required by a synchronous machine that would implement the processor and exploit the parallelism. For example, a machine with the wavefront indexes shown in Figure 9-11 would require a maximum of four identical FUs operating concurrently, with each

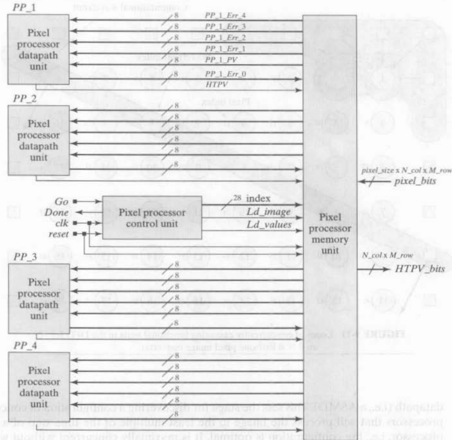


FIGURE 9-12 Alternative architecture for exploiting the concurrency of the DFG of the 8 × 6 halftone pixel processor shown in Figure 9-10.

unit implementing the fine-grained logic required to update a pixel. *The machine would update the entire image in only 18 time steps, while using fewer FU resources and requiring design of a more complex controller than the baseline design.*[14]

The architecture of the machine's datapath will be designed to exploit the concurrency that is evident in the DFG. Then a controller will be designed for the datapath. We will pursue by manual methods what a behavioral synthesis tool should accomplish and then compare the result to the baseline design in more detail.

[14]The baseline sequential design updates the array in 48 time steps.

TABLE 9-1 A reservation table for mapping DFG nodes to time slots and processors of a 8 × 6 halftone image converter.

		Time slots																	
		t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}
Processors	P_1	1	2	3	4	5	6	7	8	15	16	23	24						
	P_2			9	10	11	12	13	14	21	22	29	30	31	32				
	P_3					17	18	19	20	27	28	35	36	37	38	39	40		
	P_4							25	26	33	34	41	42	43	44	45	46	47	48

Our alternative architecture is shown in Figure 9-12. It has four pixel processors (FUs), a memory unit, and a controller. The inputs of each processor provide the value of the pixel at the location indicated by the address passed from the controller and the errors at the locations of the pixel's selected neighbors. The controller is implemented as a sequencer, which creates the addresses of the pixels that are to be updated in a given time step. The entire array of pixels is downloaded from the environment to the memory unit in a single clock cycle.

Having identified the need for at most four FUs, we address the resource mapping task by constructing Table 9-1, a *reservation table*, which shows one of many possible mappings of the nodes of the DFG of the 8 × 6 halftone image converter to a set of four processors that exploit the concurrency that was exposed by the DFG. The columns of the table establish a linear execution schedule specifying the DFG nodes that will execute in a given time; the rows of the table establish a binding between the nodes of the DFG and the processors of the architecture. The table establishes a space–time partition in which every node has a unique (processor, time slot) pair.

The architecture represented by the reservation table can process a single image in 18 time slots, but thereafter the pipelining that is evident in the table presents an opportunity to stream the images with a throughput of only 12 time slots (see Problem 8 at the end of the chapter). This particular architecture, however, cannot exploit this opportunity because the memory cannot be selectively loaded to partially update its contents, say in t_{13}, to access the next image cell.[15] In general, the number of clock cycles consumed by a time slot is strongly influenced by bus resources and memory operations.

An ASM chart for the control unit of the machine is shown in Figure 9-13. The state boxes list the values of *index* corresponding to the entries in Table 9-1. This implementation does not exploit the pipelining that is apparent in Table 9-1.

The Verilog description of *Image_Converter_Concurrent Processors*, the alternative image converter, ands its testbench are given below. The signal *Ld_Image* is added to the interface to allow an image to be stored on command, thus freeing up the external bus that provides the image. Once an image is loaded into memory, the machine waits until *Go* is asserted and then converts the image.

[15]A more elaborate implementation could buffer the images and form the datapaths required to realize minimum latency.

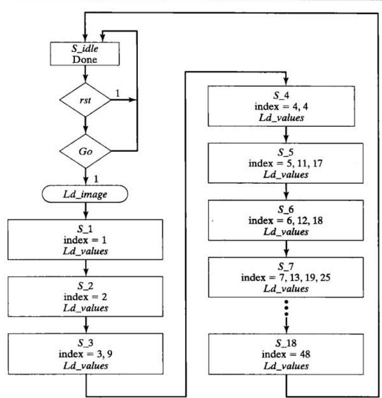


FIGURE 9-13 ASM chart for the control unit of an alternative halftone image converter.

Simulation results for *Image_Converter_Concurrent_Processors* are shown in Figure 9-14 for the test images that were given in Figure 9-6. The halftone images match those produced by *Image_Converter_Baseline, Image_Converter_*0, and *Image_Converter_*1. Figure 9-15 shows the simulation activity for the graduated image in Figure 9-7, and displays the reservation table of the four concurrently executing processors.

```
module Image_Converter_Concurrent_Processors # (parameter
  pixel_size = 8, N_col = 8, M_row = 6)(
  output [1: N_col*M_row] HTPV_bits,
  output Done,
  input [1: pixel_size * N_col * M_row] pixel_bits,
  input Go, clk, reset
);
```

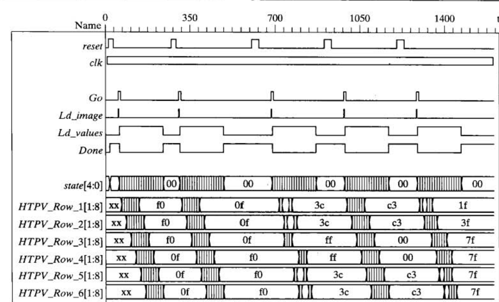


FIGURE 9-14 Simulation results for *Image_Converter_Concurrent_Processors* operating on the test images given in Figure 9-6.

```
wire       [23: 0]          index;
wire                        Ld_image, Ld_values;
wire       [pixel_size -1: 0]
           PP_1_Err_1, PP_1_Err_2, PP_1_Err_3, PP_1_Err_4,
           PP_1_PV,
           PP_2_Err_1, PP_2_Err_2, PP_2_Err_3, PP_2_Err_4,
           PP_2_PV,
           PP_3_Err_1, PP_3_Err_2, PP_3_Err_3, PP_3_Err_4,
           PP_3_PV,
           PP_4_Err_1, PP_4_Err_2, PP_4_Err_3, PP_4_Err_4,
           PP_4_PV;
wire       [pixel_size -1: 0]
           PP_1_Err_0, PP_2_Err_0, PP_3_Err_0, PP_4_Err_0;
wire       PP_1_HTPV, PP_2_HTPV, PP_3_HTPV, PP_4_HTPV;
// Instantiate Pixel Processor Datapath Units Control Unit, Memory Unit
  PP_Datapath_Unit M1_Datapath
   (PP_1_Err_0, PP_1_HTPV, PP_1_Err_1, PP_1_Err_2, PP_1_Err_3, PP_1_Err_4,
     PP_1_PV);
  PP_Datapath_Unit M2_Datapath
```

```
  (PP_2_Err_0, PP_2_HTPV, PP_2_Err_1, PP_2_Err_2, PP_2_Err_3,
    PP_2_Err_4, PP_2_PV);

 PP_Datapath_Unit M3_Datapath
  (PP_3_Err_0, PP_3_HTPV, PP_3_Err_1, PP_3_Err_2, PP_3_Err_3,
    PP_3_Err_4, PP_3_PV);

 PP_Datapath_Unit M4_Datapath
  (PP_4_Err_0, PP_4_HTPV, PP_4_Err_1, PP_4_Err_2, PP_4_Err_3,
    PP_4_Err_4, PP_4_PV);

 PP_Control_Unit M0_Controller (index, Ld_image, Ld_values, Done, Go, clk, reset);
 PP_Memory_Unit M5_Memory (
  HTPV_bits,
  PP_1_Err_1, PP_1_Err_2, PP_1_Err_3, PP_1_Err_4, PP_1_PV,
  PP_2_Err_1, PP_2_Err_2, PP_2_Err_3, PP_2_Err_4, PP_2_PV,
  PP_3_Err_1, PP_3_Err_2, PP_3_Err_3, PP_3_Err_4, PP_3_PV,
  PP_4_Err_1, PP_4_Err_2, PP_4_Err_3, PP_4_Err_4, PP_4_PV,
  PP_1_Err_0, PP_2_Err_0, PP_3_Err_0, PP_4_Err_0,
  PP_1_HTPV, PP_2_HTPV, PP_3_HTPV, PP_4_HTPV,
  pixel_bits, index, Go, Ld_image, Ld_values, clk, reset
 );
endmodule

module PP_Control_Unit (output reg [23: 0] index, output reg Ld_image, Ld_values,
  Done, input Go, clk, reset);
 reg        [4: 0]      state, next_state;

 parameter
  S_idle = 5'd0, S_1 = 5'd1, S_2 = 5'd2, S_3 = 5'd3, S_4 = 5'd4, S_5 = 5'd5,
  S_6 = 5'd6, S_7 = 5'd7, S_8 = 5'd8, S_9 = 5'd9, S_10 = 5'd10, S_11 = 5'd11,
  S_12 = 5'd12, S_13 = 5'd13, S_14 = 5'd14, S_15 = 5'd15, S_16 = 5'd16,
  S_17 = 5'd17, S_18 = 5'd18;

 always @ (posedge clk) if (reset) state <= S_idle; else state <= next_state;

 always @ (state or Go) begin
  Ld_values = 0; next_state = S_idle;
  case (state)
   S_idle:   if (Go) next_state = S_1;
   S_18:     begin next_state = S_idle; Ld_values = 1; end
   default: begin next_state = state + 1; Ld_values = 1; end
  endcase
 end

 always @ (state, Go) begin
  Done = 0;
  Ld_image = 0;
  if (state == S_idle) begin Done = 1; if (Go) Ld_image = 1; end
 end

 always @ (state) begin
  index = 0;
```

```
  case (state)
    S_idle:    index = {{6'd0}, {6'd0}, {6'd0}, {6'd0}};
    S_1:               index = {{6'd1}, {6'd0}, {6'd0}, {6'd0}};
    S_2:               index = {{6'd2}, {6'd0}, {6'd0}, {6'd0}};
    S_3:               index = {{6'd3}, {6'd9}, {6'd0}, {6'd0}};
    S_4:               index = {{6'd4}, {6'd10}, {6'd0}, {6'd0}};
    S_5:               index = {{6'd5}, {6'd11}, {6'd17}, {6'd0}};
    S_6:               index = {{6'd6}, {6'd12}, {6'd18}, {6'd0}};
    S_7:               index = {{6'd7}, {6'd13}, {6'd19}, {6'd25}};
    S_8:               index = {{6'd8}, {6'd14}, {6'd20}, {6'd26}};
    S_9:               index = {{6'd15}, {6'd21}, {6'd27}, {6'd33}};
    S_10:      index = {{6'd16}, {6'd22}, {6'd28}, {6'd34}};
    S_11:      index = {{6'd23}, {6'd29}, {6'd35}, {6'd41}};
    S_12:      index = {{6'd24}, {6'd30}, {6'd36}, {6'd42}};
    S_13:      index = {{6'd0}, {6'd31}, {6'd37}, {6'd43}};
    S_14:      index = {{6'd0}, {6'd32}, {6'd38}, {6'd44}};
    S_15:      index = {{6'd0}, {6'd0}, {6'd39}, {6'd45}};
    S_16:      index = {{6'd0}, {6'd0}, {6'd40}, {6'd46}};
    S_17:      index = {{6'd0}, {6'd0}, {6'd0}, {6'd47}};
    S_18:      index = {{6'd0}, {6'd0}, {6'd0}, {6'd48}};
    default:   index = 0;
  endcase
 end
endmodule

// Pixel Processor Datapath Unit
module PP_Datapath_Unit # (parameter pixel_size = 8)(
 output [pixel_size -1: 0] Err_0,
 output HTPV,
 input      [pixel_size -1: 0]  Err_1, Err_2, Err_3, Err_4, PV
);
 wire       [pixel_size + 1: 0] CPV, CPV_round, E_av;

 // Weights for the average error; choose for compatibility with divide-by-16 (>> 4)
 parameter  w1 = 2, w2 = 8, w3 = 4, w4 = 2;
 parameter  Threshold = 128;

 assign     E_av =          (w1 * Err_1 + w2 * Err_2 + w3 * Err_3 + w4 * Err_4 )
                             >> 4;
 assign     CPV =           PV + E_av;
 assign     CPV_round =     (CPV < Threshold) ? 0: 255;
 assign     HTPV =          (CPV_round == 0) ? 0: 1;
 assign     Err_0 =         CPV - CPV_round;
endmodule

module PP_Memory_Unit # (parameter pixel_size = 8, N_col = 8, M_row = 6)(
 output [1: N_col*M_row] HTPV_bits,
 output reg [pixel_size -1: 0]
  PP_1_Err_1, PP_1_Err_2, PP_1_Err_3, PP_1_Err_4, PP_1_PV,
  PP_2_Err_1, PP_2_Err_2, PP_2_Err_3, PP_2_Err_4, PP_2_PV,
```

```
    PP_3_Err_1, PP_3_Err_2, PP_3_Err_3, PP_3_Err_4, PP_3_PV,
    PP_4_Err_1, PP_4_Err_2, PP_4_Err_3, PP_4_Err_4, PP_4_PV,

  input [pixel_size -1: 0] PP_1_Err_0, PP_2_Err_0, PP_3_Err_0, PP_4_Err_0,
  input PP_1_HTPV, PP_2_HTPV, PP_3_HTPV, PP_4_HTPV,
  input [1: pixel_size * N_col * M_row] pixel_bits,
  input [23: 0] index,

  input Go, Ld_image, Ld_values, clk, reset
);

// Array of pixel data
  reg [pixel_size -1: 0] PV [1: N_col][1: M_row];

// Array of halftone pixel values
  reg HTPV [1: N_col][1: M_row];

// Array of pixel error values
  reg [pixel_size -1: 0] Err [0: N_col +1] [0: M_row];
  genvar nn, mm;
  generate

  // Form vector of output halftone values
   for (mm = 1; mm <= M_row; mm = mm + 1) begin: HTPV_row_loop
    for (nn = 1; nn <= N_col; nn = nn + 1) begin: HTPV_col_loop
     assign HTPV_bits [(mm - 1)*N_col + nn] = HTPV[nn][mm];
    end
   end
  endgenerate

  wire [5: 0]
   index_1 = index [23: 18],
   index_2 = index [17: 12],
   index_3 = index [11: 6],
   index_4 = index [5: 0];

// Retrieve data for pixel processors

  always @ (index_1) begin
   case (index_1)
    1, 2, 3, 4, 5, 6, 7, 8: begin  PP_1_Err_1 = Err [index_1 -1][1];
                                     PP_1_Err_2 = Err [index_1 -1][0];
                                     PP_1_Err_3 = Err [index_1][0];
                                     PP_1_Err_4 = Err [index_1 +1][0];
                                     PP_1_PV = PV [index_1][1];
    end
    15, 16: begin
                                     PP_1_Err_1 = Err [index_1 -1 -8][2];
                                     PP_1_Err_2 = Err [index_1 -1 -8][1];
                                     PP_1_Err_3 = Err [index_1 -8][1];
                                     PP_1_Err_4 = Err [index_1 +1 -8][1];
                                     PP_1_PV = PV [index_1 -8][2];
    end
```

```
    23, 24: begin
                                  PP_1_Err_1 = Err [index_1 -1 -16][3];
                                  PP_1_Err_2 = Err [index_1 -1 -16][3];
                                  PP_1_Err_3 = Err [index_1 -16][3];
                                  PP_1_Err_4 = Err [index_1 +1 -16][3];
                                  PP_1_PV = PV [index_1 -16][3];
    end
    default: begin
                                  PP_1_Err_1 = 8'bx; PP_1_Err_2 = 8'bx;
                                  PP_1_Err_3 = 8'bx;
                                  PP_1_Err_4 = 8'bx; PP_1_PV = 8'bx;
    end
  endcase
end

always @ (index_2) begin
  case (index_2)
    9, 10, 11, 12, 13, 14: begin
                                  PP_2_Err_1 = Err [index_2 -1 -8][2];
                                  PP_2_Err_2 = Err [index_2 -1 -8][1];
                                  PP_2_Err_3 = Err [index_2 -8][1];
                                  PP_2_Err_4 = Err [index_2 +1-8][1];
                                  PP_2_PV = PV [index_2 -8][2];
    end
    21, 22: begin
                                  PP_2_Err_1 = Err [index_2 -1 -16][3];
                                  PP_2_Err_2 = Err [index_2 -1 -16][2];
                                  PP_2_Err_3 = Err [index_2 -16][2];
                                  PP_2_Err_4 = Err [index_2 +1 -16][2];
                                  PP_2_PV = PV [index_2 -16][3];
    end
    29, 30, 31, 32: begin
                                  PP_2_Err_1 = Err [index_2 -1 -24][4];
                                  PP_2_Err_2 = Err [index_2 -1 -24][3];
                                  PP_2_Err_3 = Err [index_2 -24][3];
                                  PP_2_Err_4 = Err [index_2 +1 -24][3];
                                  PP_2_PV = PV [index_2 -24][4];
    end
    default: begin
                                  PP_2_Err_1 = 8'bx; PP_2_Err_2 = 8'bx;
                                  PP_2_Err_3 = 8'bx;
                                  PP_2_Err_4 = 8'bx; PP_2_PV = 8'bx;
    end
  endcase
end

always @ (index_3) begin
  case (index_3)
    17, 18, 19, 20: begin
```

```
                        PP_3_Err_1 = Err [index_3 -1-16][3];
                        PP_3_Err_2 = Err [index_3 -1-16][2];
                        PP_3_Err_3 = Err [index_3 -16][2];
                        PP_3_Err_4 = Err [index_3 +1 -16][2];
                        PP_3_PV = PV [index_3 -16][3];
    end
    27, 28: begin
                        PP_3_Err_1 = Err [index_3 -1 -24][4];
                        PP_3_Err_2 = Err [index_3 -1 -24][3];
                        PP_3_Err_3 = Err [index_3 -24][3];
                        PP_3_Err_4 = Err [index_3 +1 -24][3];
                        PP_3_PV = PV[index_3 -24][4];
    end
    35, 36, 37, 38, 39, 40: begin
                        PP_3_Err_1 = Err [index_3 -1 -32][5];
                        PP_3_Err_2 = Err [index_3 -1 -32][4];
                        PP_3_Err_3 = Err [index_3 -32][4];
                        PP_3_Err_4 = Err [index_3 + 1 -32][4];
                        PP_3_PV = PV [index_3 -32][5];
    end
    default: begin
                        PP_3_Err_1 = 8'bx; PP_3_Err_2 = 8'bx;
                        PP_3_Err_3 = 8'bx;
                        PP_3_Err_4 = 8'bx; PP_3_PV = 8'bx;
    end
  endcase
end

always @ (index_4) begin
  case (index_4)
    25, 26: begin
                        PP_4_Err_1 = Err [index_4 -1 -24][4];
                        PP_4_Err_2 = Err [index_4 -1 -24][3];
                        PP_4_Err_3 = Err [index_4 -24][3];
                        PP_4_Err_4 = Err [index_4 +1 -24][3];
                        PP_4_PV = PV [index_4 -24][4];
    end
    33, 34: begin
                        PP_4_Err_1 = Err [index_4 -1 -32][5];
                        PP_4_Err_2 = Err [index_4 -1 -32][4];
                        PP_4_Err_3 = Err [index_4 -32][4];
                        PP_4_Err_4 = Err [index_4 +1 -32][4];
                        PP_4_PV = PV [index_4 -32][5];
    end
    41, 42, 43, 44,
    45, 46,47, 48:      begin
                        PP_4_Err_1 = Err [index_4 -1 -40][6];
                        PP_4_Err_2 = Err [index_4 -1 -40][5];
```

```
                                  PP_4_Err_3 = Err [index_4 -40][5];
                                  PP_4_Err_4 = Err [index_4 +1 -40][5];
                                  PP_4_PV = PV [index_4 -40][6];
   end
   default: begin
                                  PP_4_Err_1 = 8'bx; PP_4_Err_2 = 8'bx;
                                  PP_4_Err_3 = 8'bx;
                                  PP_4_Err_4 = 8'bx; PP_4_PV = 8'bx;
   end
  endcase
 end

// Synchronous Behavior
 integer n, m;

 always @ (posedge clk)
  if (reset) begin
   for (m = 0; m <= M_row; m = m + 1)
    for (n = 0; n <= N_col + 1; n = n + 1)
     Err [n][m] = 0;

   for (m = 1; m <= M_row; m = m + 1)
    for (n = 1; n <= N_col; n = n + 1)
     PV [n][m] <= 0;
 end

 else if (Ld_image) begin: Array_Initialization
  for (m = 1; m <= M_row; m = m +1) begin: row_loop
   for (n = 1; n <= N_col; n = n +1) begin: col_loop
    Err [n][m] <= 0;          // Note part-select (+:) in next line to form range
    PV [n][m] <= pixel_bits[(m -1)*N_col*pixel_size + (n -1)*pixel_size +
     1 +: pixel_size];
   end // col_loop
  end       // row_loop
 end        // Array Initialization

 else if (Ld_values) begin: Image_Conversion
  case (index_1)
   1, 2, 3, 4, 5, 6, 7, 8:       begin
          Err [index_1][1] <= PP_1_Err_0;
          HTPV [index_1] [1] <= PP_1_HTPV; end

   15, 16: begin
          Err [index_1 -8][2] <= PP_1_Err_0;
          HTPV [index_1 -8][2] <= PP_1_HTPV; end

   23, 24: begin
          Err [index_1 -16][3] <= PP_1_Err_0;
          HTPV [index_1 -16][3] <= PP_1_HTPV; end
  endcase
```

```
case (index_2)
 9, 10, 11, 12, 13, 14: begin
         Err [index_2 -8][2] <= PP_2_Err_0;
         HTPV [index_2 -8][2] <= PP_2_HTPV; end
 21, 22: begin
         Err [index_2 -16][3] <= PP_2_Err_0;
         HTPV [index_2 -16][3] <= PP_2_HTPV; end
 29, 30, 31, 32: begin
         Err [index_2 -24][4] <= PP_2_Err_0;
         HTPV [index_2 -24][4] <= PP_2_HTPV; end
endcase
case (index_3)
 17, 18, 19, 20: begin
         Err [index_3 -16][3]<= PP_3_Err_0;
         HTPV [index_3 -16][3] <= PP_3_HTPV; end
 27, 28: begin Err [index_3 -24][4] <= PP_3_Err_0;
         HTPV [index_3 -24][4] <= PP_3_HTPV; end
 35, 36, 37, 38, 39, 40:     begin
         Err [index_3 -32][5] <= PP_3_Err_0;
         HTPV [index_3 -32][5] <= PP_3_HTPV; end
endcase
case (index_4)
 25, 26: begin
         Err [index_4 - 24][4] <= PP_4_Err_0;
         HTPV [index_4 -24][4] <= PP_4_HTPV; end
 33, 34: begin
         Err [index_4 -32][5] <= PP_4_Err_0;
         HTPV [index_4 -32][5] <= PP_4_HTPV;end
 41, 42, 43, 44, 45, 46, 47, 48: begin
         Err [index_4 -40][6] <= PP_4_Err_0;
         HTPV [index_4 -40][6] <= PP_4_HTPV; end
endcase
end      // Image_Conversion
endmodule

module t_Image_Converter_Concurrent_Processor();
 parameter pixel_size = 8, N_col = 8, M_row = 6;
 wire       Done;
 reg        [1: pixel_size*N_col*M_row]       pixel_bits;
 reg        Go, clk, reset;
 wire       [1: N_col*M_row]                  HTPV_bits;
 wire [1: N_col] HTPV_Row_1 = HTPV_bits[1: 8];
 wire [1: N_col] HTPV_Row_2 = HTPV_bits[9: 16];
 wire [1: N_col] HTPV_Row_3 = HTPV_bits[17: 24];
 wire [1: N_col] HTPV_Row_4 = HTPV_bits[25: 32];
```

```
wire [1: N_col] HTPV_Row_5 = HTPV_bits[33: 40];
wire [1: N_col] HTPV_Row_6 = HTPV_bits[41: 48];

// Instantiate image converter
Image_Converter_Concurrent_Processors M1 (HTPV_bits, Done, pixel_bits, Go,
 clk, reset);
initial begin #1200 $finish; end
initial begin clk = 0; forever #5 clk = ~clk; end
initial fork #10 reset = 1; #30 reset = 0; join
initial fork
  #50 Go = 1; #60 Go = 0;
  #250 Go = 1; #260 Go = 0;
  #450 Go = 1; #460 Go = 0;
  #650 Go = 1; #660 Go = 0;
  #850 Go = 1; #860 Go = 0;
join
initial fork begin: Image_Pattern_1
   pixel_bits = {  8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00,
             8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00,
             8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00,
             8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff,
             8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff,
             8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff};
  end
  #200 begin: Image_Pattern_2
   pixel_bits = {  8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff,
             8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff,
             8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff,
             8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00,
             8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00,
             8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00};
  end
  #400 begin: Image_Pattern_3_Cross
   pixel_bits = {  8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h0,
             8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h00,
             8'hff, 8'hff, 8'hff, 8'hff, 8'hff, 8'hff, 8'hff, 8'hff,
             8'hff, 8'hff, 8'hff, 8'hff, 8'hff, 8'hff, 8'hff, 8'hff,
             8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h0,
             8'h00, 8'h00, 8'hff, 8'hff, 8'hff, 8'hff, 8'h00, 8'h0};
  end
  #600 begin: Image_Pattern_4_Bar_Cross
   pixel_bits = {  8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff,
             8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff,
             8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00,
             8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00,
             8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff,
             8'hff, 8'hff, 8'h00, 8'h00, 8'h00, 8'h00, 8'hff, 8'hff};
  end
```

```
#800 begin: Image_Pattern_5_Graduated_Left_to_Right
  pixel_bits = {  8'h1f, 8'h3f, 8'h5f, 8'h8f, 8'h9f, 8'hbf, 8'hdf, 8'hff,
                  8'h1f, 8'h3f, 8'h5f, 8'h8f, 8'h9f, 8'hbf, 8'hdf, 8'hff,
                  8'h1f, 8'h3f, 8'h5f, 8'h8f, 8'h9f, 8'hbf, 8'hdf, 8'hff,
                  8'h1f, 8'h3f, 8'h5f, 8'h8f, 8'h9f, 8'hbf, 8'hdf, 8'hff,
                  8'h1f, 8'h3f, 8'h5f, 8'h8f, 8'h9f, 8'hbf, 8'hdf, 8'hff,
                  8'h1f, 8'h3f, 8'h5f, 8'h8f, 8'h9f, 8'hbf, 8'hdf, 8'hff};
 end
join
endmodule
```

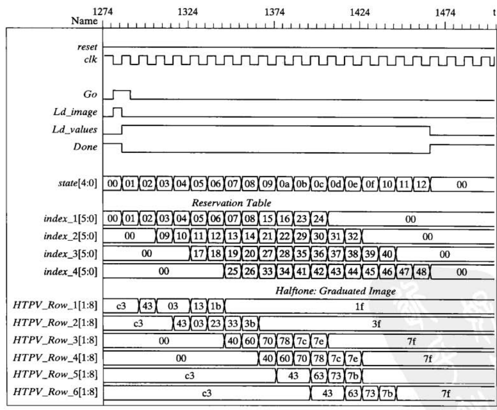


FIGURE 9-15 Simulation results for *Image_Converter_Concurrent_Processors* operating on the graduated test image given in Figure 9-7, showing the scheduling of four concurrently executing processors.

9.2.4 Halftone Pixel Image Converter: Design Tradeoffs

Key trade-offs between the alternative designs of the halftone image converter are summarized in Table 9-2. A more accurate comparison would require synthesis of each implementation. This task is left as an exercise for the reader.

9.2.5 Architectures for Dataflow Graphs with Feedback

The dataflow graph of the pixel processor did not have feedback, so the baseline processor could be realized by feedback-free combinational logic. If the DFG for an algorithm has feedback, the machine will require memory and can be implemented only as a sequential machine.

Example 9.1

An NLP describing the so-called bubble-sort algorithm for sorting a set of N unsigned binary numbers and arranges them in ascending order [6] is given below, in pseudocode.

```
begin
 for i=2 to N_key
 begin
  for j = N_key downto i do
   if A[j-1] > A[j] then
   begin
       temp= A[j-1];
       A[j-1]=A[j];
       A[j]=temp
   end
end
```

TABLE 9-2 A comparison of alternative halftone pixel image converters.

Tradeoffs: 8 × 6 Pixel Halftone Image Converter			
Version	**FU Utilization**	**Memory Utilization**	**Execution Time**
Image_Converter_Baseline[1]	48	None (combinational)	$T_{Baseline}$
Image_Converter_0[2]	48	None (combinational)	$48 \times T_{FU}$
Image_Converter_1[3]	NA*	$6 + 2 \times 48 \times 8$ bytes	$48 \times T_{FU}$
Image_Converter_SR[4]	8	$6 + 2 \times 48 \times 8$ bytes	$48 \times T_{FU}$
Image_Converter_2[5]	4	$6 + 2 \times 48 \times 8$ bytes	$18 \times T_{FU}(12 \times T_{FU})$**

[1]NLP-based Structure of FUs.
[2]NLP-based level-sensitive cyclic behavior.
[3]NLP-based single cycle synchronous.
[4]NLP-based multicycle synchronous.
[5]ASMD-based concurrent processors.
*Not synthesizable.
**Streaming images.

The DFG of a FU for the machine is shown in Figure 9-16(a). We have included additional structure to represent the memory cells that are associated with the data that are manipulated by the algorithm. The FU compares two adjacent numbers stored in memory and determines whether to swap the contents of the storage registers that hold the numbers. The DFG has feedback, because the contents of a memory cell can be written back to the cell. The presence of feedback in the DFG implies the need for data storage so that contending operations can be separated by the clock. If the loops of the NLP program are unrolled, we get the temporal DFG shown in Figure 9-16(b). Each iteration of the nested loops of the algorithm must occur in a separate clock cycle in order for data to be fetched, transformed, and written back to memory by concurrent operations on registers. The shaded nodes and memory cells in Figure 9-16(b) indicate the datapaths that are exercised during a given time step as the machine executes the algorithm.

The structure of the DFG for the bubble-sort algorithm suggests that a baseline implementation of the machine consists of a single FU that executes repeatedly with different data, until the algorithm expires. The ASMD chart for a machine that implements the baseline architecture is shown in Figure 9-17. The machine has a bank of

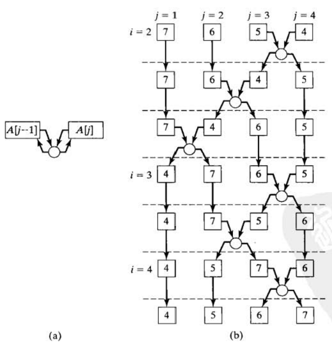


FIGURE 9-16 Bubble sort machine: (a) baseline functional unit, and (b) a temporal DFG for the machine.

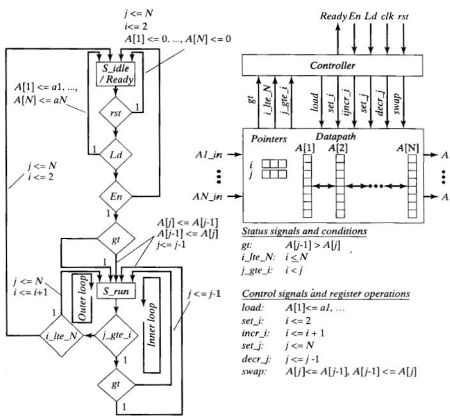


FIGURE 9-17 The ASMD chart and block diagram of a machine that executes the bubble-sort algorithm.

N registers holding the words of data, plus two counters, which index the inner and outer loops of the NLP.

The Verilog model of the bubble sort machine is listed below. For clarity, the interface signals between the controller and the datapath are shown below the block diagram. The simulation results in Figure 9-18 illustrate the execution of the algorithm with a testbench that loads two sets of data (see Problem 9-4).

```
module Bubble_Sort # ( parameter N = 8, word_size = 4)(
  output     [word_size -1: 0]  A1, A2, A3, A4, A5, A6, A7, A8,
  output                        Ready,
  input      [word_size -1: 0]  A1_in, A2_in, A3_in, A4_in, A5_in, A6_in, A7_in,
                                  A8_in,
  input                         En, Ld, clk, rst
);
```

```
  Control_Unit M0_Controller (
   Ready, load, set_i, incr_i, set_j, decr_j, swap, En, Ld, gt, i_lte_N, j_gte_i, clk, rst);

  Datapath_Unit M1_Datapath (
   A1, A2, A3, A4, A5, A6, A7, A8, gt, i_lte_N, j_gte_i,
   A1_in, A2_in, A3_in, A4_in, A5_in, A6_in, A7_in, A8_in,
   load, set_i, incr_i, set_j, decr_j, swap, clk, rst
  );
endmodule

module Control_Unit (output Ready, output reg load, set_i, incr_i, set_j,
   decr_j, swap,
  input En, Ld, gt, i_lte_N, j_gte_i, clk, rst);

  parameter     S_idle = 0, S_sort = 1;
  reg           state, next_state;
  assign        Ready = (state == S_idle);

  always @ (posedge clk) if (rst) state <= S_idle; else state <= next_state;

  always @ (state, En, Ld, gt, i_lte_N, j_gte_i) begin
   next_state = S_idle; load = 0; decr_j = 0; incr_i = 0; set_j = 0; set_i = 0; swap = 0;
   case (state)
    S_idle:     if (Ld) begin next_state = S_idle; load = 1; end
                else if (En) begin
                  next_state = S_sort; if (gt) begin swap = 1; decr_j = 1; end
                end else next_state = S_idle;

    S_sort:     if (j_gte_i) begin next_state = S_sort; decr_j = 1; if (gt) swap = 1;
                 end
                else if (i_lte_N) begin next_state = S_sort; set_j = 1; incr_i = 1; end
                else begin next_state = S_idle; set_j = 1; set_i = 1; end
   endcase
  end
endmodule

module Datapath_Unit # (parameter word_size = 4, N = 8) (
  output [word_size -1: 0] A1, A2, A3, A4, A5, A6, A7, A8,
  output gt, i_lte_N, j_gte_i,
  input [word_size -1: 0] A1_in, A2_in, A3_in, A4_in, A5_in, A6_in, A7_in, A8_in,
  input load, set_i, incr_i, set_j, decr_j, swap, clk, rst
);
  reg     [word_size -1: 0]  A [1: N]; // Array of words
  reg     [word_size -1: 0]  i, j;
  assign A1 = A[1], A2 = A[2], A3 = A[3], A4 = A[4];
  assign A5 = A[5], A6 = A[6], A7 = A[7], A8 = A[8];

  assign gt = (A[j-1] > A[j]);          // compares words
  assign i_lte_N = (i <= N);
  assign j_gte_i = (i <= j);

always @ (posedge clk)                  // Datapath and status registers
   if (rst) begin i <= 0; j <= 0;
```

```
      A[1] <= 0; A[2] <= 0; A[3] <= 0; A[4] <= 0;
      A[5] <= 0; A[6] <= 0; A[7] <= 0; A[8] <= 0; end
    else begin
      if (load) begin i <= 2; j <= N;
        A[1] <= A1_in; A[2] <= A2_in; A[3] <= A3_in; A[4] <= A4_in;
        A[5] <= A5_in; A[6] <= A6_in; A[7] <= A7_in; A[8] <= A8_in;
    end
    if (swap) begin A[j] <= A[j-1]; A[j-1] <= A[j]; end
    if (decr_j) j <= j-1;
    if (incr_i) i <= i+1;
    if (set_j) j <= N;
    if (set_i) i <= 2;
  end
endmodule

module t_Bubble_Sort ();
  parameter word_size = 4;
  wire [word_size -1: 0]    A1, A2, A3, A4, A5, A6, A7, A8;
  wire    Ready;
  reg     En, Ld, clk, rst;
  reg [word_size -1: 0]     A1_in, A2_in, A3_in, A4_in, A5_in, A6_in, A7_in, A8_in;
  parameter                 a1 = 8, a2 = 1, a3 = 8, a4 = 1, a5 = 8, a6 = 1, a7 = 8,
                              a8 = 8;
  parameter                 a21 = 8, a22 = 7, a23 = 6, a24 = 5, a25 = 6, a26 = 3,
                              a27 = 2, a28 = 1;

  Bubble_Sort M0 (A1, A2, A3, A4, A5, A6, A7, A8, Ready,
    A1_in, A2_in, A3_in, A4_in, A5_in, A6_in, A7_in, A8_in,
    En, Ld, clk, rst
);
  initial #1000 $finish;
  initial begin clk = 0; forever #5 clk = ~clk; end
  initial fork
    rst = 1;
    #20 rst = 0; #450 rst = 1; #470 rst = 0;
    #30 Ld = 1; #40 Ld = 0; #500 Ld = 1; #510 Ld = 0;
    #10 En = 1; #20 En = 0; #60 En = 1; #70 En = 0; #500 En = 1; #520 En = 0;
    #30 begin A1_in = a1; A2_in = a2; A3_in = a3; A4_in = a4;
              A5_in = a5; A6_in = a6; A7_in = a7; A8_in = a8;
    end
    #500 begin A1_in = a21; A2_in = a22; A3_in = a23; A4_in = a24;
               A5_in = a25; A6_in = a26; A7_in = a27; A8_in = a28;
    end
  join
endmodule
```

End of Example 9.1

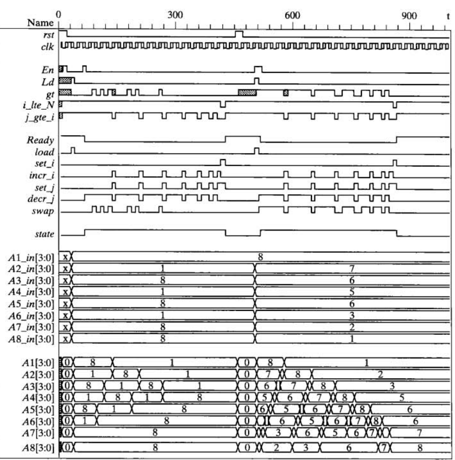


FIGURE 9-18 Simulation results for *Bubble_Sort*, a machine that executes a bubble-sort algorithm, with an initial sort of (8, 7, 6, 5, 4, 3, 2, 1).

Example 9.2

The machine *Bubble_Sort* in Example 9.1 has an input port for each word that is to be sorted. This scheme has two major disadvantages: the port structure is not portable to other implementations—the model must be edited to accommodate more or less ports, and the wide datapath required to support the machine might not be practical in a given application. Figure 9-19 shows an ASMD chart for *Bubble_Sort_Alternative*, a machine having (1) a parameterized word size and (2) an input datapath that is one

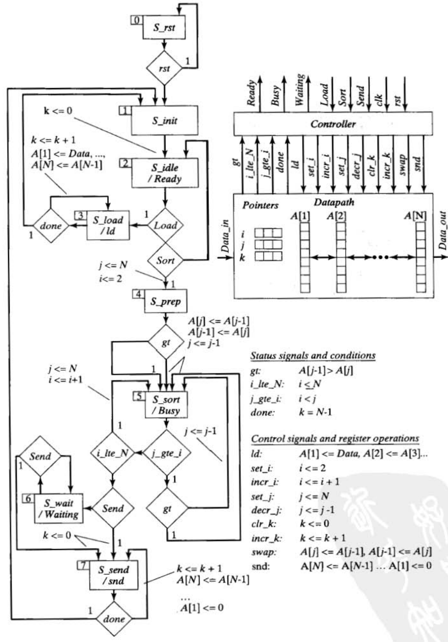


FIGURE 9-19 ASMD chart for *Bubble_Sort_Alternative*.

word wide (see Problem 9-5). As an alternative form of an ASMD chart, a table is added to the chart identifying the register operations associated with control signals, without including the signals in output boxes on the chart. An external agent must manage the data bus and assert a signal, *Load*, to cause the machine to store the contents of the input data bus into a circular buffer of parameterized size until *Load* is de-asserted (wrap-around and overwrite are possible). With *Load* de-asserted, assertion of a signal, *Sort*, will cause the machine to initiate sorting of the contents of the circular buffer using the bubble sort algorithm. After sorting is complete, the agent's assertion of *Send* for one cycle of the clock will cause the machine to successively place each word of the sorted contents on an output databus, in descending sequence, for one cycle. An output signal, *Sending*, is to be asserted by the machine while the words are present on the bus. The bus is to be a three-state bus. After the last word of the sequence occupies the bus for one cycle, the machine is to return to the reset state. The machine is to have synchronous reset.

End of Example 9.2

9.3 Digital Filters and Signal Processors

Digital signal processors (DSPs) are the "brains" of cellular phones, personal digital assistants, still-image cameras, video cameras, video recorders, and myriad portable web-based interactive devices, where they provide superior performance, at lower cost and lower power, as compared with analog circuits. In this section, we will consider the use of Verilog to model FUs that encode, transmit, and transform digital representations of signals.

DSPs can be categorized according to the sampling frequencies that are required by the spectral content of the signals in an application. Shannon's sampling theorem states that a bandlimited signal[16] can be recovered from its time-domain samples if it is sampled at a frequency, f_s, that is greater than twice the highest frequency in the signal's spectrum. A waveform recovered from samples that were obtained by sampling a signal below its Shannon frequency is called an alias of the signal, because the waveform cannot be distinguished from other signals sampled at the same rate. The sample period, in practice, determines the maximum time available for a processor to operate on a sample of data before the next sample arrives. DSPs, therefore, can be classified according to the portion of the frequency domain in which they are intended to operate, as shown in Table 9-3 [7]. The sampling frequency (1) determines the spectral domain over which a signal can be recovered without aliasing effects and (2) determines the time interval available to perform operations on the data. Whether the processor is executing instructions stored in memory or

[16]The spectrum of a bandlimited signal is 0 everywhere but within a finite range of frequencies.

TABLE 9-3 DSP applications and I/O sample rates.

Application	I/O Sampling Rate
Instrumentation	1 Hz
Control	>0.1 kHz
Voice	8 kHz
Audio	44.1 kHz
Video	1–14 MHz

implementing an algorithm in hardware, the sample period constrains the design. A second important constraint arises from the digital nature of the signals that are being processed.

A digital signal is represented by a binary word with a finite word length in the machine. Consequently, the information that is processed is subject to truncation, roundoff, overflow, and underflow errors during processing. For a given dynamic range of a signal, the word length of its digital format determines the resolution (precision) of its values. Thus, performance, precision, and functionality characterize a DSP.

DSPs can be implemented in hardware or software or a combination of both. A software-based approach executes a DSP algorithm on a general-purpose processor. The focus of the design effort is on the software that programs the processor for the tasks supporting an application. Various software tools are available to optimize the program for the machine.[17] A second approach is to implement a DSP algorithm with a special-purpose, hardwired, high-performance, customized processor whose architecture has been designed specifically to accomplish a variety of signal-processing tasks efficiently. The task of optimizing the design is performed by the synthesize tools that create an architecture and synthesize the logic implementing the processor.

A dedicated signal processor can be implemented as an ASIC chip to achieve the most efficient design and the highest performance, but at high unit cost and reduced flexibility. FPGAs provide yet another approach—they can be configured to implement any DSP algorithm, but their performance and density may lag behind that of a special-purpose DSP or an ASIC chip. FPGAs, however, afford the benefits of flexibility, reduced NRE (nonrecurring engineering) costs, rapid prototyping, early market entry and reduced risk.

Signal processors are characterized by high throughput and multiple concurrent operations. DSPs are typically dataflow intensive and have relatively small control units. DSPs have multiple arithmetic and logic unit (ALU)-like FUs, with high-speed support for the operations of multiplication and addition, multiple address and data busses supporting concurrent operations, and multiport registers and random-access memories (RAMs). Dedicated DSPs are dominated by their datapaths; their control units are much simpler than those of general-purpose processors.

DSPs operate synchronously on fixed-word-length samples of data that arrive at regular intervals of time. The instruction sets of a DSP typically includes two

[17] See Texas Instrument's Code Composer Studio.

fundamental arithmetic operations: multiplication and addition, commonly referred to as multiply and accumulate (MAC). MAC FUs must be implemented efficiently and must give high performance.

A DSP unit is constrained by the physical technology in which it is implemented, which fundamentally limits the speed at which its operations can execute and also determines the physical area required to implement devices in hardware. A DSP may be constrained by the number of channels of data, the rate at which data are exchanged with the machine's environment, and the size of the input and output words. External channels with a high data rate are multiplexed within the DSP to reduce the internal data rates to levels compatible with the processor's performance.

The data that drive a DSP unit may originate as an analog signal, which is sampled to form a discrete-time signal (i.e., an indexed sequence of numbers). Then the discrete-time signal is converted to a binary fixed-word representation forming a digital signal (an indexed sequence of numbers with a finite word length). The analog signal itself might have been corrupted by noise, leading to the need for filtering of the received signal. Such filters are commonly implemented within an ASIC or within an FPGA.

Digital filters transform digital representations of analog signals to remove noise and other unwanted signal components and to shape the spectral characteristics of the resulting signal. Digital filters operate on a finite-precision digital representation of a signal. Consequently, their design must consider finite word length effects that result from the representation of the signal samples, the weighting coefficients of the filter, and the arithmetic operations performed by the filter.

The operations of a DSP unit can be distributed spatially (i.e., over multiple hardware units) or temporally (using a single processor), depending on whether the unit executes its operations concurrently, in a single cycle of the clock, or sequentially, over multiple cycles. In the former case, in which the unit operates on the entire word of data, the hardware resources must complete the operation within the period of the clock. In the latter case, the machine operates on part of the data word in each clock cycle, so that the capacity and performance of the individual operational units can be relaxed. Distributing operations over the temporal axis allows the machine to operate with higher throughput, but at the expense of a latency between the arrival of data and the availability of the results. Latency can be tolerated in many applications, such as in digital communications.

Digital filters operate in the time-sequence domain, accepting a sequence of discrete, finite-length words to produce an output sequence of words. The sequence of inputs, $x[n]$, may be the output of an analog-to-digital converter, or the output of some other FU. Two common architectures for linear digital filters are represented by the block diagrams in Figure 9-20. A *finite-duration impulse response* (FIR) filter (Figure 9-20(a)) forms its outputs as a weighted sum of its inputs; an *infinite-duration impulse response* (IIR) filter (Figure 9-20(b)) forms its output from a weighted sum of its inputs and past values of its output [7–11]. Consequently, the block diagram symbol of an IIR filter is shown with feedback from the output to the input. Both types of filters have internal registers to hold samples of the inputs, but the IIR filter has additional memory for samples of the output.

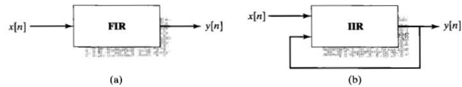


FIGURE 9-20 Digital filters: (a) finite impulse response filter, and (b) infinite impulse response filter.

Digital filters are usually designed to have two important characteristics: causality and linear phase. A filter is *causal* if its impulse response is 0 before the impulse is applied. FIR filters are widely used in practical applications because they can be designed to have a linear phase characteristic,[18] which ensures that the filter's output signal is a time-shifted (delayed), but undistorted copy of its input signal [12].[19] On the other hand, an IIR filter having a *linear phase* characteristic cannot be causal. Noncausal filters cannot be realized in hardware, but they can be implemented in software and have practical utility in offline interpolation of data. Another distinction between the two types of filters is that FIR filters cannot accumulate roundoff error; IIR filters can accumulate roundoff error as the output is successively passed through the filter.

9.3.1 Finite-Duration Impulse Response Filter

A FIR digital filter forms its output as a weighted sum of present and past samples of its input, as described by the *feed-forward difference equation* written below. FIR filters are called moving average filters because their output at any time index depends on a window containing only the most recent M samples of the input, as shown in Figure 9-21. Because its response depends on only a finite record of inputs, a FIR filter will have a finite-length, nonzero response to a discrete-time impulse (i.e., the response of an Mth order FIR filter to an impulse will be 0 after M clock cycles).

$$y_{\text{FIR}}[n] = \sum_{k=0}^{M} b_k x[n-k]$$

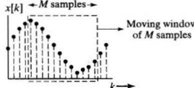


FIGURE 9-21 Sample window for a FIR moving average filter.

[18]The phase characteristic is required to be linear in the passband of the filter's frequency response.
[19]Techniques exist for designing a FIR filter to have symmetric coefficients, which guarantees that the phase characteristic of the filter is linear.

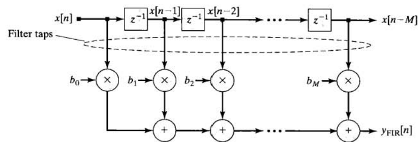


FIGURE 9-22 Functional block diagram for an *M*th-order FIR digital filter.

A FIR filter can be described by the *z*-domain functional block diagram[20] shown in Figure 9-22, where (in synchronous operation) each box labeled with z^{-1} denotes a register cell having a delay of one clock cycle. The diagram represents the datapaths and operations that must be performed by the filter. Each stage of the filter holds a delayed sample of the input, and the connection at the input and the connections at the outputs of the stages are referred to as *taps*, and the set of coefficients $\{b_k\}$ are called the *tap coefficients* of the filter. An *M*th order FIR will have $M + 1$ taps. The samples of data flow through the shift register, and at each clock edge (i.e., time index), *n*, the samples are weighted (multiplied) by the tap coefficients and added together to form the output, $y_{FIR}[n]$. The adders and multipliers of the filter must be fast enough to form $y[n]$ before the next clock, and at each stage, they must be sized to accommodate the width of their datapaths. In applications in which numerical accuracy is a driving consideration, lattice architectures may reduce the effects of finite word length, but at the expense of increased computational cost [12].

In most applications, the goal of the implementation is to do the filtering as fast as possible to achieve the highest sampling frequency [7]. The longest signal path through the combinational logic includes *M* stages of addition and one stage of multiplication.

The architecture of a FIR must specify a finite word length for each of the machine's arithmetic units, and manage the flow of data during operation. The architecture shown in Figure 9-23 consists of a shift register, multipliers, and adders implementing an *M*th-order FIR. The datapaths must be wide enough to accommodate the output of the multipliers and adders. The samples are encoded as finite-length words and then shifted in parallel through a series of *M* registers. A cascaded chain of MACs form the machine.

9.3.2 Digital Filter Design Process

A process for designing an ASIC- or FPGA-based digital filter has the main steps shown in Figure 9-24. A design begins with development of performance specifications for cutoff

[20]See Stearns and David [12] for other architectures and a discussion of their merits.

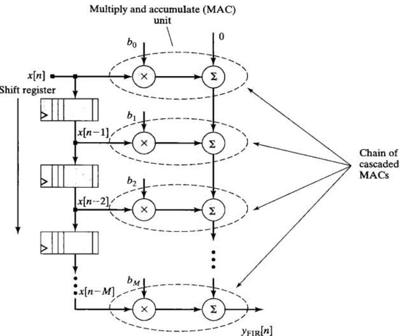


FIGURE 9-23 MAC-based architecture for a type-I, *M*th-order FIR filter.

frequency, transition band limits, in-band ripple, minimum stop band attenuation, etc. The filter is described by a C-language specification/algorithm, which must be converted into a Verilog RTL model that can be synthesized into hardware that implements the algorithm. The design flow is not ideal, because the algorithm's description in C must be translated into Verilog, at the risk of introducing errors. In C, the variables can be represented as floating-point numbers, but in Verilog, the parameters and other data values are expressed in a fixed-point, finite-word-length format. New tools are under development by EDA vendors to support design flows that create an executable and directly synthesizable specification, omitting intermediate translations to an HDL.[21] Various architectures implement FIR and IIR filters [7,9,10,12]. For a given architecture, tools such as MATLAB [7,13,14] can be used to determine the filter coefficients that implement a filter that satisfies the specifications of the design. Digital filters operate on finite-word-length representations of physical (analog) values. The finite word length of the data limits the resolution and the dynamic range that can be represented by the filter, leading to quantization errors. Similarly, the representations of the numerical coefficients of the filter have a finite word length, which contributes to additional quantization and truncation error. When data are represented by

[21]See www.synopsys.com

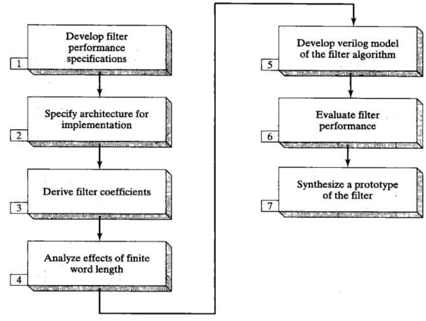


FIGURE 9-24 Design flow for digital filters.

integers, there is an error caused by truncation of the fractional part produced by an arithmetic operation. The arithmetic operations that are performed by the filter can lead to overflow and underflow errors, which must be detected by the machine.

Example 9.3

An eighth-order Gaussian, lowpass FIR filter is modeled by *FIR_Gaussian_Lowpass* on page 561. The design is fully synchronous, with active-high synchronous reset. The filter's tap coefficients are implemented as unsigned 8-bit words (for unsigned integer math), chosen to be even-symmetric to guarantee that the phase characteristic will be linear.

Various algorithms and tools exist for designing lowpass filters to meet specifications on their passband cutoff frequency, stopband frequency, passband gain, stopband attenuation, and sampling rate [7,15]. The coefficients in *FIR_Gaussian_Lowpass* were chosen to give the impulse response of the filter an approximately Gaussian shape. This choice simplifies the design because the coefficients are positive and can be scaled to be represented by unsigned binary values.[22] Their magnitudes are determined by a Gaussian distribution over a range (0–9), with an arbitrarily chosen standard deviation of 2. The fractions obtained from the distribution were scaled in proportion to their size relative to the sum of the weights, and then multiplied by 255, the maximum value for an 8-bit word.

The impulse response of a FIR filter is a sample sequence whose values are the filter's tap coefficients. These can be seen in the waveform for *Data_out* in Figure 9-25. Because *Data_in* is switching at the falling edge of the clock and is sampled at the rising edge, the values of *Data_out* are valid immediately before the rising edge of the clock.[23] Note that *Data_out* is formed as a Mealy output of the machine, and that the value of *Data_out* immediately after the first rising edge of clock reflects the value of *Data_in* and the first stored sample of *Data_in* (i.e., the output is not valid). Also note that the output has a finite duration (equal to the eight sample periods).

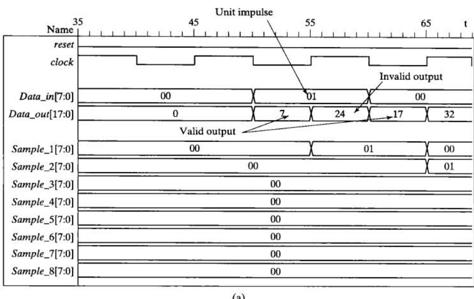
```
// Eighth-order, Gaussian Lowpass FIR
module FIR_Gaussian_Lowpass # (parameter
  order = 8,
  word_size_in = 8,
  word_size_out = 2*word_size_in + 2,
  b0 = 8'd7,          // Filter coefficients
  b1 = 8'd17,
  b2 = 8'd32,
  b3 = 8'd46,
  b4 = 8'd52,
  b5 = 8'd46,
  b6 = 8'd32,
  b7 = 8'd17,
  b8 = 8'd7)(
  output  [word_size_out -1: 0]    Data_out,
  input   [word_size_in-1: 0]      Data_in,
  input                            clock, reset
);
  reg     [word_size_in-1: 0]      Samples[1: order];
  integer                          k;
  assign Data_out = b0 * Data_in + b1 * Samples[1] + b2 * Samples[2]
                    + b3 * Samples[3] + b4 * Samples[4]
                    + b5 * Samples[5] + b6 * Samples[6]
                    + b7 * Samples[7] + b8 * Samples[8];

  always @ (posedge clock)
     if (reset == 1) begin for (k = 1; k <= order; k = k+1) Samples[k] <= 0; end
     else begin
       Samples [1] <= Data_in;
       for (k = 2; k <= order; k = k+1) Samples[k] <= Samples[k-1];
     end
endmodule
```

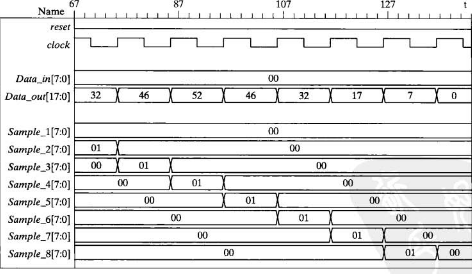
End of Example 9.3

[22]Other schemes lead to signed fractions, which can be represented in a 2s complement Q-format (see Kehtarnavaz, note 4).

[23]The displayed values reflect the scaling that was done in forming the tap coefficients.



(a)



(b)

FIGURE 9-25 Impulse response of *FIR_Gaussian_Lowpass*, an eighth-order Gaussian Lowpass FIR filter: (a) initial nonzero samples of *Data_out*, showing invalid and valid data, and (b) final nonzero samples of *Data_out*.

9.3.3 Infinite-Duration Impulse Response Filter

Infinite-duration impulse response (IIR) filters are the most general class of linear digital filters. Their output at a given time step depends on their inputs and on previously computed outputs (i.e., they have memory) [10]. IIR filters are recursive, and FIR filters are nonrecursive.[24] The output of a IIR filter is formed in the data-sequence domain as a weighted sum according to the Nth-order difference equation shown below:

$$y_{\mathrm{IIR}}[n] = \sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k]$$

The filter is recursive because the difference equation has feedback. Consequently, the filter's response to an impulse may have infinite duration (i.e., it does not become 0 in a finite time).

An IIR filter is modeled in the z domain by its z-domain system function, or transfer function, which is a ratio of polynomials formed as

$$H_{\mathrm{IIR}}(z) = \sum_{k=0}^{M} b_k z^{-k} \Big/ \left(1 - \sum_{k=1}^{N} a_k z^{-k}\right)$$

The z-domain transforms of the input and output time sequences are related by

$$Y(z) = H_{\mathrm{IIR}}(z) X(z).$$

The tap coefficients of the IIR filter form the sets, $\{a_j\}$ and $\{b_k\}$, commonly referred to as the *feedback* and *feedforward* coefficients, respectively. The parameter N is the order of the filter; it specifies the number of prior samples of the output that must be saved to form the current output; it also determines the latency of the output. The value of the parameter M specifies how many prior samples of the input will be used to form the output. The roots of the polynomials of $H_{\mathrm{IIR}}(z)$ determine the location of the filter's poles and zeros in the z domain and shape both the data sequence of the filter's response to its input, and the frequency domain function that specifies how the filter responds to a periodic input [10,15].

Various architectures implement an IIR filter, and exhibit different requirements for physical resources, and different sensitivities to numerical errors caused by finite word length for the data and the parameters. The structure shown in Figure 9-26 is known as a Type-1 IIR, and consists of separate feedforward and feedback blocks implemented as a pair of shift registers—one to hold samples of the input, $x[n]$, and another to hold samples of the output, $y[n]$.

[24]Nonrecursive filters are stable (i.e., their response does not become unbounded); recursive filters may be unstable, depending on the filter's coefficients.

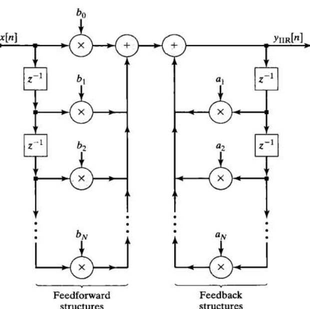


FIGURE 9-26 Functional block diagram for a type-I, *N*th-order IIR filter.

Example 9.4

The Verilog model *IIR_Filter_8* can be used to implement eighth-order IIR filters, depending on the selection of tap coefficients.

```
module IIR_Filter_8 # (parameter
// Eighth-order, Generic IIR Filter
 order = 8,
 word_size_in = 8,
 word_size_out = 2*word_size_in + 2,
 // Feedforward filter coefficients
 b0 = 8'd7, b1 = 0, b2 = 0, b3 = 0, b4 = 0, b5 = 0, b6 = 0, b7 = 0, b8 = 0,
 // Feedback filter coefficients
 a1 = 8'd46, a2 = 8'd32, a3 = 8'd17, a4 = 8'd0, a5 = 8'd17, a6 = 8'd32, a7 = 8'd46,
  a8 = 8'd52)(

 output        [word_size_out -1: 0]     Data_out,
 input         [word_size_in-1: 0]       Data_in,
 input                                   clock, reset
);
```

```
reg       [word_size_in-1: 0]     Samples_in [1: order];
reg       [word_size_in-1: 0]     Samples_out [1: order];
wire      [word_size_out -1: 0]   Data_feedforward;
wire      [word_size_out -1: 0]   Data_feedback;
integer                           k;

assign Data_feedforward =      b0 * Data_in
                             + b1 * Samples_in[1]
                             + b2 * Samples_in[2]
                             + b3 * Samples_in[3]
                             + b4 * Samples_in[4]
                             + b5 * Samples_in[5]
                             + b6 * Samples_in[6]
                             + b7 * Samples_in[7]
                             + b8 * Samples_in[8];

assign Data_feedback =         a1 * Samples_out [1]
                             + a2 * Samples_out [2]
                             + a3 * Samples_out [3]
                             + a4 * Samples_out [4]
                             + a5 * Samples_out [5]
                             + a6 * Samples_out [6]
                             + a7 * Samples_out [7]
                             + a8 * Samples_out [8];

assign Data_out = Data_feedforward + Data_feedback;

always @ (posedge clock)
 if (reset == 1) for (k = 1; k <= order; k = k+1) begin
  Samples_in [k] <= 0;
  Samples_out [k] <= 0;
 end
 else begin
  Samples_in [1] <= Data_in;
  Samples_out [1] <= Data_out;
  for (k = 2; k <= order; k = k+1) begin
   Samples_in [k] <= Samples_in [k-1];
   Samples_out [k] <= Samples_out [k-1];
  end
 end
endmodule
```

End of Example 9.4

Two alternative architectures for an *N*th order IIR are shown in Figure 9-27. They are known as Direct Form II (DF-II) and Transposed Direct Form II (TDF-II) [10].

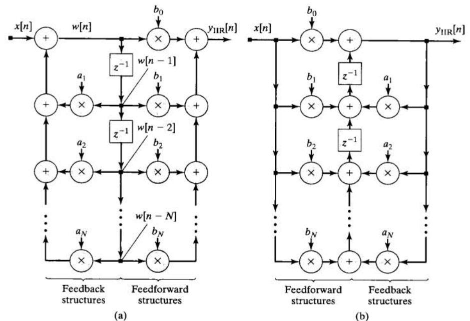


FIGURE 9-27 Functional block diagrams for Nth-order IIR filters: (a) Direct Form II (DF-II), and (b) Transposed Direct Form II (TDF-II).

9.4 Building Blocks for Signal Processors

In this section, we will consider models of basic operations of integration, differentiation, decimation, and interpolation, which are common to datapath units of many digital processors.[25]

9.4.1 Integrators (Accumulators)

Digital integrators are used in a popular type of analog-to-digital converter, called a sigma-delta modulator [7]. Digital integrators accumulate a running sum of sample values. Two implementations are common: parallel and sequential.

Example 9.5

The model *Integrator_Par* below describes an integrator for a parallel datapath. At each clock cycle, the machine adds *data_in* to the content of the register *data_out*. The signal *hold* pauses the accumulation of samples until it is de-asserted.

[25]These examples were motivated by the models presented in Chris Hagan's master's thesis [11].

```
module Integrator_Par # (parameter word_length = 8)(
  output reg      [word_length-1: 0]      data_out,
  input           [word_length-1: 0]      data_in,
  input                                   hold, clock, reset
);

  always @ (posedge clock) begin
    if (reset) data_out <= 0;
    else if (hold) data_out <= data_out;
    else data_out <= data_out + data_in;
  end
endmodule
```

End of Example 9.5

Example 9.6

The architecture of a byte-sequential integrator is shown in Figure 9-28, and a Verilog model of the machine, *Integrator_Seq*, is given below. It is common for a processor to receive data via a narrower datapath than the datapath within the processor. In this example, the unit is to accumulate 32-bit words, but receives data sequentially, in 8-bit bytes. The signal *hold* pauses the accumulation of samples until it is de-asserted. This

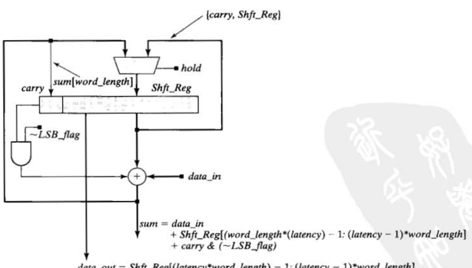


FIGURE 9-28 Architecture of a byte-sequential integrator.

t ↓	*Shft_Reg*			
	Byte_1	*Byte_2*	*Byte_3*	*Byte_4*
	Byte_1 + Byte_5	*Byte_2 + Byte_6*	*Byte_3 + Byte_7*	*Byte_4 + Byte_8*
	Byte_1 + Byte_5 + Byte_9	*Byte_2 + Byte_6 + Byte_10*	*Byte_3 + Byte_7 + Byte_11*	*Byte_4 + Byte_8 + Byte_12*
	Byte_1 + Byte_5 + Byte_9 + ByteE_13	*Byte_2 + Byte_6 + Byte_10 + Byte_14*	*Byte_3 + Byte_7 + Byte_11 + Byte_15*	*Byte_4 + Byte_8 + Byte_12 + Byte_16*

FIGURE 9-29 Accumulation of bytes in *Shft_Reg* over 16 cycles of operation.[26]

architecture performs byte-wide addition, with the current data sample being added to the leftmost byte of the shift register *Shft_Reg*, to form *sum*. At the next clock edge, the content of the shift register is shifted toward its MSByte,[27] and the previously formed sum is loaded into the register's LSByte. These two actions occur concurrently, and the MSByte that the shift register held before the clock is pushed out of the register. The accumulation of bytes in *Shft_reg* is illustrated in Figure 9-29 for a scheme in which four successive bytes compose a word. The input signal *LSB_flag* controls the addition of a carry so that corresponding bytes are added correctly from word to word.

Figure 9-29 demonstrates how successive bytes of *data_in* are aligned within 32-bit words and accumulated in *Shft_Reg*. The simulation results in Figure 9-30 are annotated to show how samples of *data_in* are loaded into *Shft_Reg*, how the leftmost byte of *Shft_Reg* is added to *data_in* to form *sum*, and how *sum* is loaded into the rightmost byte of *Shft_Reg*.

```
module Integrator_Seq (parameter word_length = 8, latency = 4)(
 output        [word_length -1: 0]                data_out;
 input         [word_length -1: 0]                data_in;
 input                                            hold, LSB_flag, clock, reset
);
 reg           [(word_length * latency) -1: 0]    Shft_Reg;
 reg                                              carry;
 wire          [word_length: 0]                   sum;

 always @ (posedge clock) begin
  if (reset) begin
   Shft_Reg <= 0;
   carry <= 0;
  end
```

[26]For simplicity, the carries between bytes are not shown.

[27]MSByte and LSByte will denote the most significant and least significant byte, respectively, of a word.

```
    else if (hold) begin
      Shft_Reg <= Shft_Reg;
      carry <= carry;
    end
    else begin
      Shft_Reg <= {Shft_Reg[word_length*(latency -1) -1: 0], sum[word_length-1: 0]};
      carry <= sum[word_length];
    end
  end

  assign sum = data_in + Shft_Reg[(latency * word_length) -1: (latency -1)*
          word_length] + (carry & (~LSB_flag));

  assign data_out = Shft_Reg[(latency * word_length) -1: (latency -1)*
          word_length];
endmodule
```

End of Example 9.6

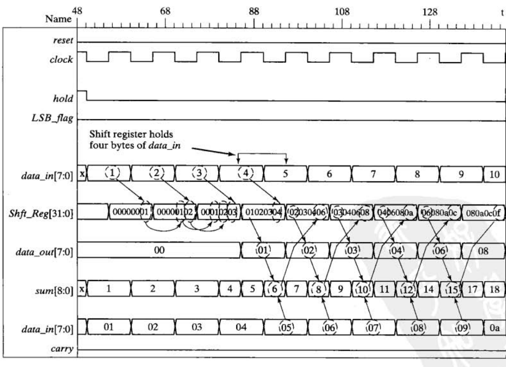


FIGURE 9-30 Simulation results for *Integrator_Seq*, a byte-sequential integrator.

9.4.2 Differentiators

A differentiator provides a measure of the sample-to-sample change in a signal. A bytewide serial differentiator is given below. The backward difference is implemented with a buffer and a subtractor.

```
module differentiator #(parameter word_size = 8)(
  output [word_size -1: 0]     data_out,
  input  [word_size -1: 0]     data_in,
  input                        hold,
  input                        clock, reset
);
  reg    [word_size -1: 0]     buffer;
  assign                       data_out = data_in - buffer;
  always @ (posedge clock) begin
    if (reset) buffer <= 0;
    else if (hold) buffer <= buffer;
    else buffer <= data_in;
  end
endmodule
```

9.4.3 Decimation and Interpolation Filters

Decimation and interpolation filters are used to achieve sample rate conversion in digital signal processors [15]. Decimation filters decrease the sample rate; interpolation filters increase the sample rate. Such conversions are important, because Shannon's Sampling Theorem [10] states that a bandlimited signal that is sampled at a rate greater than twice its upper spectral limit[28] can be recovered from its samples. Interpolation filters enable a signal to be oversampled, thereby reducing or eliminating the effects of aliasing. If a signal is not sampled properly, it cannot be recovered with fidelity. Decimation is used to reduce the bandwidth of a signal that has been oversampled. Decimation achieves sample rate reduction.

Example 9.7

The Verilog model *Decimator_1* describes the behavior of a parallel-in-parallel-out decimator, which samples its input at a rate determined by *clock* unless *hold* is asserted [11]. Note that samples of *data_in* in Figure 9-31 are dropped because *clock* is running at a rate that is slower than the rate at which *data_in* had been sampled.

```
module Decimator_1 # (parameter word_length = 8)(
  output reg    [word_length-1: 0     data_out,
  input         [word_length-1: 0     data_in,
  input                               hold,        // Active high
  input                               clock,       // Positive edge
  input                               reset        // Active high
);
```

[28]The upper spectral limit of a bandlimited signal determines the bandwidth of the signal.

```
always @ (posedge clock)
  if (reset) data_out <= 0;
  else if (hold) data_out <= data_out;
  else data_out <= data_in;
endmodule
```

End of Example 9.7

Example 9.8

The Verilog model *Decimator_2* samples a parallel input and produces a parallel output, but includes an option to form a serial output by shifting the output word through the LSB while *hold* is asserted. This action is apparent in the waveforms shown in Figure 9-32.

```
module Decimator_2 # (parameter word_length = 8)(
  output reg [word_length-1: 0]  data_out,
  input  [word_length-1: 0]      data_in,
  input                          hold,          // Active high
  input                          clock,         // Positive edge
  input                          reset          // Active high
);
  always @ (posedge clock)
    if (reset) data_out <= 0;
      else if (hold) data_out <= data_out >> 1;
      else data_out <= data_in;
endmodule
```

End of Example 9.8

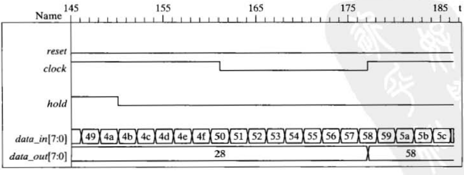


FIGURE 9-31 Simulation results for *Decimator_1*.

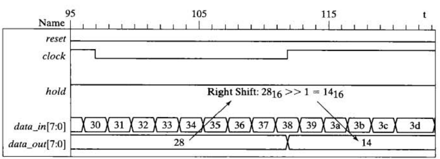


FIGURE 9-32 Simulation results for *Decimator_2*, showing that with *hold* asserted the register holding *data_out* is shifted to the right at successive clock edges.

Example 9.9

The decimator shown in Figure 9-33(a) is designed to work in tandem with a sequential integrator [11]. The decimator's architecture consists of three registers, *Shft_Reg*, *Int_Reg*, and *Decim_Reg*. All three are sized to hold multiple bytes (samples), as determined by a parameter *latency*. Samples from *data_in* are loaded sequentially into the MSByte of *Shft_Reg*, and shifted toward the LSByte on subsequent clocks (Figure 9-33(b)). When *Shft_Reg* is full, two register transfers occur concurrently (Figure 9-33(c)): (1) the contents of *Shft_Reg* are loaded into an intermediate holding register, *Int_Reg* and (2) the LSByte of a new word is loaded into the MSByte of *Shft_Reg*. Subsequent transfers load *Shft_Reg* until it is full. When *Shft_Reg* is full the word can be loaded into the intermediate register, *Int_Reg*. *data_out*, the output data word, is formed from the contents of *Decim_Reg* (Figure 9-33(d)).

The Verilog model, *Decimator_3*, includes two edge-sensitive cyclic behaviors—one to describe the byte-buffering activity and the other to describe the functionality of a decimator. The register transfers of *decimator_3* are shown in Figure 9-33. If *load* is asserted (see Figure 9-33(a)), two register operations occur concurrently: the current sample of *data_in* is loaded into the leftmost bits of *Shft_Reg*, and the contents of *Shft_Reg* are loaded into *Int_Reg*. The decimation register, *Decim_Reg*, holds its contents while *hold* is asserted; otherwise, it gets the contents of the intermediate register. The decimation action of the machine is a consequence of the parameter *latency* (i.e., the difference in the rate at which bytes are sequenced at *data_in* and the rate at which words are formed at *data_out*. Register *Decim_Reg* can be connected directly to the input of a sequential integrator, such as *Integrator_Seq*.

The simulation results shown in Figure 9-33(e) demonstrate that the action of *reset* flushes the registers and overrides the action of *load* and *hold*. The results in Figure 9-33(f) show the data word $bbccddff_{16}$ being shifted into *Shft_Reg* in four consecutive clock cycles. At the next clock, with *load* asserted, the content of *Shft_Reg*

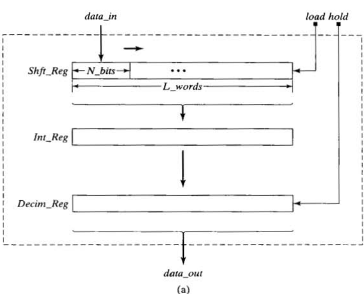


FIGURE 9-33 Sequential decimator: (a) overall architecture, (b) concurrent register transfers of *data_in* to *Shft_Reg* and of *Shft_Reg* to *Int_Reg* with *load* asserted, (c) shifting contents and loading *data_in* into *Shft_Reg* with *load* de-asserted, (d) loading contents from *Int_Reg* to *Decim_Reg*, with *hold* de-asserted, (e) simulation results shown action of reset, and (f) simulation results showing action of *load* and *hold*.

is dumped into *Int_Reg* and the LSByte of *Shft_Reg* gets aa_{16}. At the next clock, with hold de-asserted, the content of *Int_Reg* is dumped into *Decim_Reg*.

```
module Decimator_3 # (parameter word_length = 8, latency = 4)(
  output     [(word_length*latency) -1: 0]     data_out,
  input      [word_length-1: 0]                data_in,
  input                                        load, hold,
  input                                        clock,
  input                                        reset
);
  reg        [(word_length*latency) -1: 0]     Shft_Reg;     // Shift reg
  reg        [(word_length*latency) -1: 0]     Int_Reg;      // Intermediate reg
  reg        [(word_length*latency) -1: 0]     Decim_Reg;    // Decimation reg

  always @ (posedge clock)     // Decimation
   if (reset) begin
    Shft_Reg <= 0;
    Int_Reg <= 0;
   end
   else begin
    if (!load) begin
     Shft_Reg <= {data_in, Shft_Reg[(word_length*latency) -1: word_length]};
    end
```

```
      else begin
        Shft_Reg[(word_length * latency) -1: (word_length*(latency-1))] <= data_in;
        Int_Reg <= Shft_Reg;
      end
    end

  always @ (posedge clock)      // Byte buffering
    if (reset) Decim_Reg <= 0;
    else if (!hold) Decim_Reg <= Int_Reg;

  assign data_out = Decim_Reg;
endmodule
```

data_in
Shft_Reg N_bits •••
L_words
Int_Reg

(b)

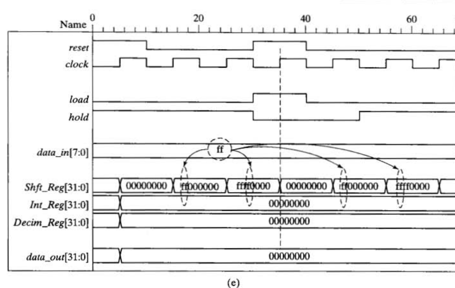
data_in
Shft_Reg ••• N_bits
Shft_Reg N_bits •••

(c)

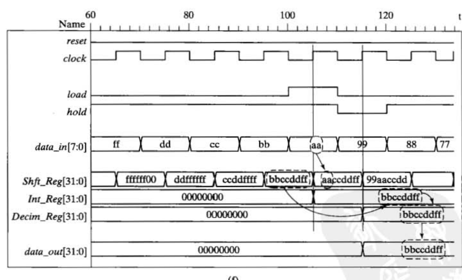
Int_Reg
Decim_Reg
data_out

(d)

FIGURE 9-33 Continued



(e)



(f)

FIGURE 9-33 Continued

End of Example 9.9

9.5 Pipelined Architectures

The shortest cycle time of the clock of a synchronous sequential machine is a measure of its performance, and it is bounded by the propagation delay through the combinational logic of the machine. The throughput of a synchronous machine is the rate at which data is supplied to and produced by the machine [3]. Throughput is ultimately limited by the path with the largest propagation delay between (1) a primary input and a register, (2) a path between a pair of registers, (3) a path from a register to a primary output, or (4) a path from a primary input to a primary output. In each case, combinational logic limits the performance of the machine.

Synthesis engines transform a set of two-level, Boolean functions for combinational logic into a set of multilevel Boolean functions with shared logic. The circuit that results is free of redundant logic and exploits don't-care conditions to achieve a minimal description whose input/output logic is equivalent to the original set of two-level equations. The logic that is produced by this process is minimal because its output functions share common internal Boolean sub-expressions as much as possible, but it might not be as fast as an equivalent realization that has fewer levels of logic. In general, collapsing levels of logic will produce a faster circuit, but this is not always possible, because wide input gates are not practical.

As an alternative approach to gaining performance, pipeline registers can be inserted into the combinational logic datapaths at strategic locations to partition the logic into groups with shorter paths [4,16]. The placement of the registers is determined by the feedforward cutsets (discussed below) of the DFG of the datapath to ensure that data remains coherent. Pipelining reduces the number of levels in the blocks of combinational logic, shortens the datapaths between storage elements, and increases the throughput of the circuit, by allowing the clock to run faster.

Pipelining becomes increasingly important in high-speed, wide-word data transmission and processing. For example, a block of combinational logic in Figure 9-34 is partitioned into two blocks and separated by a pipeline register to form an alternative circuit. Suppose the longest path through the original multilevel combinational logic has time-length T_{max}, and a operating frequency of $f_{multilevel} = 1/T_{max}$. If the partition creates two blocks of (multilevel) logic, each having a maximum time length of $1/2\ T_{max}$, the pipelined circuit can operate at a frequency $f_{pipeline} = 2/T_{max} = 2\ f_{multilevel}$.[29]

A word of caution: *the partition of a datapath must maintain coherency of the data—a datapath traced from any primary input to any primary output must pass through the same number of pipeline registers.* In general, a cutset of a connected graph is a set of branches that, if removed from the graph, isolates a node of the graph. For our purposes, a *pipeline cutset*, or *feedforward cutset*, is a minimum set of edges that, if removed from the graph, partitions it into two connected subgraphs such that there is no path between an input node and an output node. Cutsets are used to determine alternative placements of pipeline registers. The simple DFG in Figure 9-35 illustrates two cutsets, one of which is a pipeline cutset. The edges through which the dashed arc passes specify a locus for

[29]For simplicity, we are neglecting clock skew and the flip-flop's timing constraints.

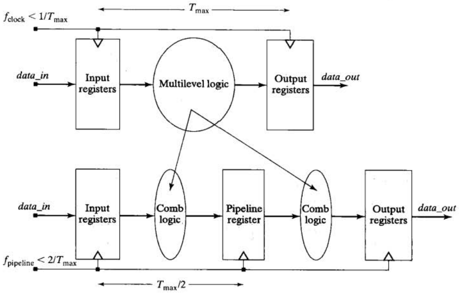


FIGURE 9-34 Partitioning a block of multilevel combinational logic and inserting a register creates a data pipeline.

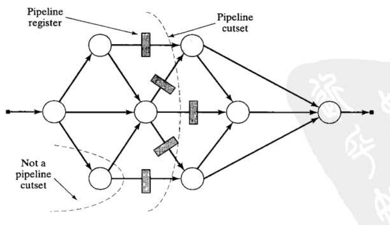


FIGURE 9-35 Cutset-based placement of pipeline registers.

pipeline registers. The feedforward cutset ensures that every path between the input node and the output node passes through the same number of pipeline registers. Removing any of the pipeline registers will destroy the coherency of the data.[30]

Pipelining has a cost. The pipeline registers introduce additional area in the physical layout of an ASIC, and require additional routing of clock resources.[31] This could be an issue for an ASIC, but high-end FPGAs are register-rich, and they readily support pipelined architectures. Partitioning a circuit to create a pipeline must be done carefully, so that balance is achieved in the distribution of path lengths among the groups that are created by the partition. In general, the delay of the slowest combinational logic stage determines the performance of the pipelined circuit and the speed at which the circuit's common clock can run.

Pipelining shortens the clock cycle and increases the throughput, but introduces input–output latency. Each stage of a pipeline adds one cycle of delay before the first output of the circuit will be available. In a two-stage pipeline, the effect of a transition of the input signal will not appear at the output until after two clock cycles. The latency accumulates through the pipeline. Latency effectively introduces a time shift between a circuit's input transitions and its output transitions (i.e., the outputs of the combinational logic after time step N are due to the inputs that were applied at time step $N - m$, where m is the number of stages of pipelining). Latency does not alter the function of the circuit. After the pipeline is full an output is formed at every clock cycle, and its maximum throughput is $1/T_{stage}$, where T_{stage} is the path length of the longest stage of the partitioned DFG.

Pipelining trades spatial (hardware) complexity for temporal complexity (performance) by computing smaller functions in less time. It distributes across multiple, shorter clock cycles the breadth of logic that would be required to implement the complete function in one clock cycle.

There are three major benefits derived from a pipeline of dedicated hardware: (1) dedicated hardware performs the same single task in every clock cycle, without requiring scheduling to coordinate its use among other tasks [17]. The operation begins with the arrival of data at every active edge of the clock, and ends in time to pass the results to the next stage of the pipeline before the arrival of the next clock, (2) the logic to perform a single, dedicated task can be streamlined and optimized as a unit, to meet constraints on performance, area, and power, and (3) the datapaths between adjacent stages of the pipeline are short and direct, reducing the need for shared data buses, control and storage, and having relatively low interconnect capacitance.

The design of a circuit with pipelined datapaths must address the following issues: (1) When should pipelining be considered? (2) Where should the pipeline registers be inserted? and (3) How much latency will be introduced by the pipeline? The design must use a minimum number of pipeline registers to achieve a minimum cycle time. Pipelining should be considered when the timing margins on the critical paths are unsatisfactory, and all other means (e.g., device resizing and alternative architectures) have been

[30]DFGs with no feedback are amenable to pipelining. Those with feedback are difficult to pipeline.

[31]Wave pipelining, a register-free form of pipelining based on coherent signal propagation, will not be considered here.

considered. Unsatisfactory timing margins point to a risk of metastability during operation. Various options exist for placing the registers in the circuit's datapaths. These must be evaluated and used to determine the overall latency of the design. Whether the latency is acceptable depends on the specifications for the system's performance.

9.5.1 Design Example: Pipelined Adder

Digital systems that operate on arrays of data typically contain a large number of adders in an array structure. The processing speed in these applications is usually critical, and may warrant pipelining.

The 16-bit adder in Figure 9-36(a) is formed by chaining two 8-bit adders in a serial connection. If each 8-bit adder has a throughput delay of 100 ns, the worst-case delay of the configuration will be 200 ns. In a synchronous environment, this structure is organized to have all operations occur in the same clock cycle. An alternative structure can be pipelined to operate at a higher throughput by distributing the processing over multiple cycles of the clock. The trade-off between speed and physical resources (more registers) can warrant this approach. The DFG of the 16-bit adder shown in Figure 9-36(b) reveals a single datapath connection between the FUs of the machine, suggesting a variety of options for pipelining. However, a balanced design will be achieved if the cutset between the 8-bit stages is used, resulting in the register placements shown in Figure 9-36(c).

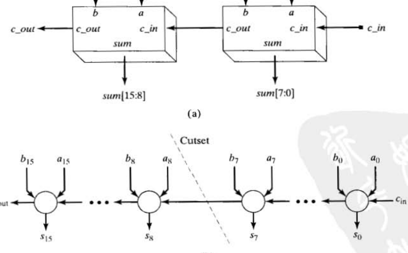


FIGURE 9-36 A pipelined 16-bit adder: (a) serial connection of two 8-bit adders to form a 16-bit adder, (b) DFGs before pipelining, and (c) after pipelining for balanced stage delays.

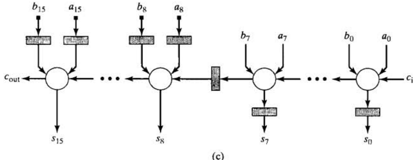


FIGURE 9-36 Continued

The pipelined architecture in Figure 9-37 contains an additional register (*PR*) between the data input register (*IR*) and the data output registers. The structure sequences the data, so that in a given clock cycle a carry bit must propagate through only half of the datapath. The interface to the input datapath still provides the entire word to the unit in a synchronous manner, but the sum of only the rightmost data byte is formed. That sum, together with the leftmost datapaths, is then stored in a 25-bit internal register. In the next clock cycle, the sum of the leftmost data bytes is formed and stored in the pipeline register with the rightmost sum and carried from the previous cycle. With the extra internal register, the pipelined unit can operate at approximately twice the frequency of the original adder, because the longest path supported by the clock interval is through an 8-bit adder instead of a 16-bit adder. After the period of initial latency, a new sum appears at the output of the unit every 100 ns.

The movement of data through the pipelined adder is depicted in Figure 9-38, where $a_L a_R(1)$ denotes the first sample of the left and right bytes of input word *a*. In the simulation results shown in Figure 9-39, note that the unit has a latency of two clock cycles between the application of the input data and the appearance of valid output. The first data words, 1122_h and 3344_h, are formed at $t_{sim} = 100$ ns, sampled and loaded into register *IR* at $t_{sim} = 150$ ns, partially added at $t_{sim} = 250$ ns, and fully added at $t_{sim} = 350$ ns. After the latency period, the data is correctly updated to achieve an overall throughput of approximately twice that of the serially connected 8-bit adders. (The setup times of physical registers will reduce the throughput slightly.)

Nonblocking procedural assignments in Verilog make concurrent assignments to register variables and are the key to modeling concurrent register transfers in architectures that are pipelined to achieve high throughput on datapaths. The Verilog model of the pipelined adder, *add_16_pipe*, uses nonblocking assignments to concurrently sample the datapaths and registers immediately before the active edge of *clock*. These samples are used to form the values that will exist in the registers immediately after the clock event. The model is scaled by the value of *size*, which can be changed to a desired value (must be an even number).

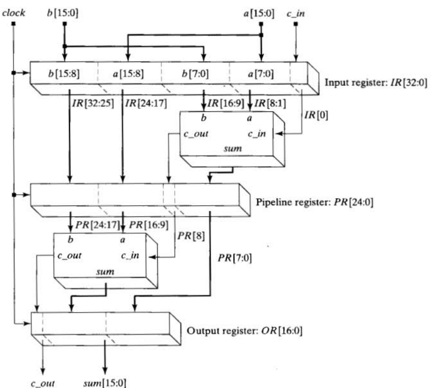


FIGURE 9-37 Pipelined 16-bit adder structure.

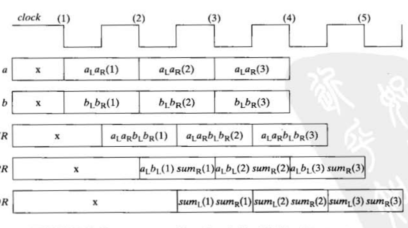


FIGURE 9-38 Data movement through a pipelined 16-bit adder structure.

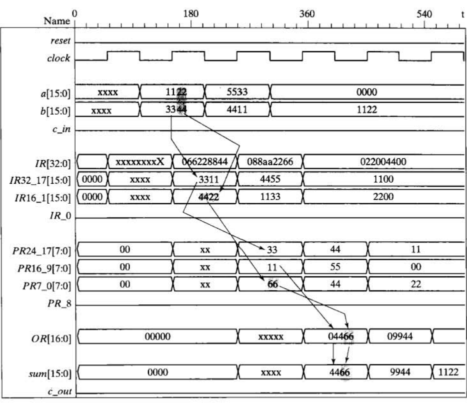


FIGURE 9-39 Simulation results showing movement of data through *add_16_pipe*, a pipelined 16-bit adder.

Figure 9-39 shows the result of simulating *add_16_pipe* in a testbench that uses hierarchical dereferencing to display the contents of the internal registers in a format that reveals the dataflow through the pipeline. The displayed outputs *IR*32_17, *IR*16_1, and *IR*_0 show the segments of *IR*. The displayed outputs *PR*24_17, *PR*16_9, *PR*8, and *PR*7_0 show the segments of *PR*. The waveforms have been annotated to illustrate the register transfers.

```
module add_16_pipe # (parameter
  size        = 16,
  half        = size / 2,
  double      = 2 * size,
  triple      = 3 * half,
  size1 = half -1,        // 7
  size2 = size -1,        // 15
  size3 = half + 1,       // 9
  R1 = 1,
```

```
  L1 = half,
  R2 = size3,
  L2 = size,
  R3 = size + 1,
  L3 = size + half,
  R4 = double - half +1,
  L4 = double
)(
  input      [size2: 0]       a, b;
  input      c_in, clock;
  output     [size2: 0]       sum;
  output     c_out
);
  reg        [double: 0]      IR;
  reg        [triple: 0]      PR;
  reg        [size: 0]        OR;
  assign {c_out, sum} = OR;

  always @ (posedge clock) begin
   // Load input register
   IR[0] <= c_in;
   IR[L1:R1] <= a[size1: 0];
   IR[L2:R2] <= b[size1: 0];
   IR[L3:R3] <= a[size2: half];
   IR[L4:R4] <= b[size2: half];
   // Load pipeline register
   PR[L3: R3] <=IR[L4: R4];
   PR[L2: R2] <=IR[L3: R3];
   PR[half: 0] <= IR[L2:R2] + IR[L1:R1] + IR[0];
   OR <= {{1'b0,PR[L3: R3]} + {1'b0,PR[L2: R2]} + PR[half], PR[size1: 0]};
  end
endmodule
```

For convenience of illustration, the results of synthesizing a 4-bit pipelined adder with asynchronous reset, *add_4_pipe* are shown in Figure 9-40. The D-type flip-flop used (*dffrpqb_a*) has active-low reset.

9.5.2 Design Example: Pipelined FIR Filter

MACs dominate the performance of DSPs. In many applications, long chains of MACs must be pipelined to increase the throughput of the unit. For example, the architecture of the FIR filter that was presented in Figure 9-22 consists of a shift register and an array of cascaded MAC units. The longest path through the circuit is proportional to the length of the chain of MACs between the input and the output. The performance specifications of a filter might require that high-speed multipliers and/or adders be used to implement MACs. Another alternative is to pipeline the datapaths to increase the throughput of the filter.

Pipeline registers can be inserted into the structure at locations determined by cutsets, as shown in Figure 9-41. In Figure 9-41(a), the cutsets of the FIR filter place the

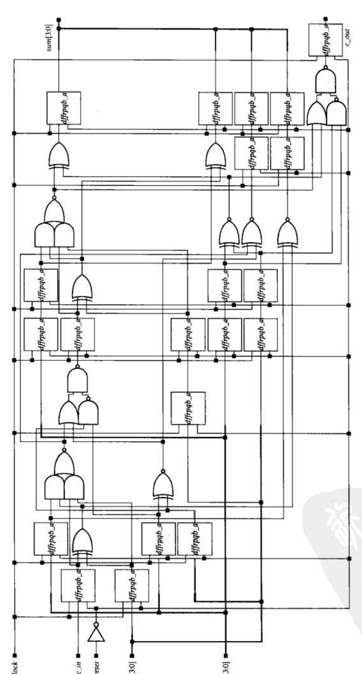


FIGURE 9-40 Circuit synthesized from *add_4_pipe*.

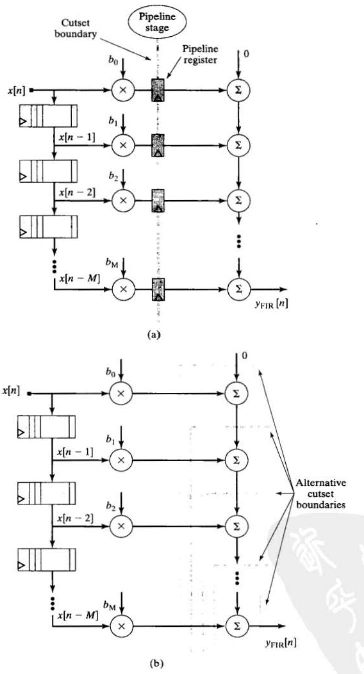


FIGURE 9-41 Alternative pipeline structures for FIR filter, with pipeline registers placed (a) at the outputs of the multipliers and (b) at the inputs of the adders.

pipeline registers at the output of the multipliers. The cutsets of the alternative structure in Figure 9-41(b) have pipeline registers at the inputs of the adders. A possibility having only two additional registers is shown. Given the apparent imbalance in the number of adder stages having pipeline registers and those that do not, the placement will not be desirable because the stage delays will not be balanced. Other implementations can be formed by relocating the pipeline registers, varying the number of registers, and/or balancing the stage delays, at the expense of reduced throughput.

9.6 Circular Buffers

The algorithms of many digital filters and other signal processors repeatedly shift and store samples of data that are taken over a moving window in the time-sequence domain. For example, a filter might form its output from a weighted sum of samples of the present and most recent $N - 1$ samples of the input. Thus, the processor uses N samples at each time step. If the algorithm is implemented in software on a general-purpose processor, these values would be stored and retrieved repeatedly as the algorithm executes, consuming several clock cycles at each cycle of the filter. Instead of this direct approach, and to realize hardware efficiency and speed, circular buffers are used to create the effect of moving an entire window of samples through memory, without actually moving all of the data [7].

Circular buffers use an address mechanism that moves pointers to register cells, instead of moving the actual data. Figure 9-42 illustrates the situation in which the nth sample of the sequence $x[k]$ is to be stored in memory. Only the most recent N samples are kept in an N-cell circular buffer, and an address pointer circulates continuously

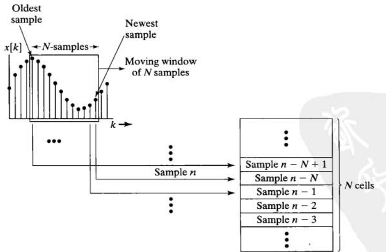


FIGURE 9-42 AN N-cell circular buffer storing data from an N-sample moving window.

through an ascending sequence of addresses, before wrapping around to the bottom (starting) address. When the *n*th sample is received, the data held in the entire array of registers is not shifted. Instead, the cell addressed by the pointer receives the *n*th sample, overwriting the previous contents at the location previously occupied by the *n-N*th sample. The net effect is that the storage of data can occur in significantly fewer memory cycles than would be required to shift data among the cells of a random access memory. However, an ASIC or an FPGA might implement the buffer with a set of dedicated, parallel-load registers, chained together as a word-wide shift register, which would allow concurrent movement of data.

Example 9.10

Two Verilog models of an *N*-sample moving window memory are described below. The first version, *Circular_Buffer_1*, uses an array of parallel-load shift registers to shift the entire contents of the buffer at each time step. The second version, *Circular_Buffer_2*, includes *write_ptr*, which points to the next cell to be read. The contents of the register do not move. The pointer is incremented at each time step, so the data below the pointer is aged. Note that in the simulation results shown in Figure 9-43, the contents of

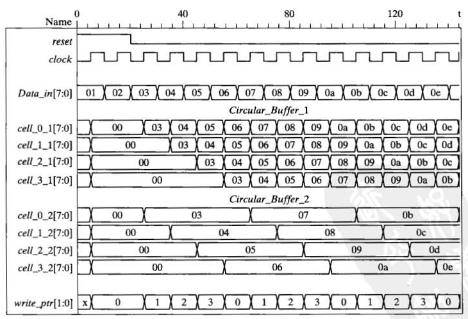


FIGURE 9-43 Results of simulating *Circular_Buffer_1* and *Circular_Buffer_2*, two versions of a data buffer holding an *N*-sample moving window.

Circular_Buffer_1 change at every cycle, while only the one cell addressed by *write_ptr* in *Circular_Buffer_2* changes as new data arrive (at the negative edges of the clock). A digital filter using the data held in *Circular_Buffer_1* would always tap the oldest data at the same location, but the filter using *Circular_Buffer_2* would need logic to track the location of the aged data relative to *write_ptr*.[32]

```
module Circular_Buffer_1 # (parameter buff_size = 4, word_size = 8) (
  output [word_size -1: 0] cell_3, cell_2, cell_1, cell_0,
  input  [word_size -1: 0] Data_in,
  input  clock, reset
);
  reg    [buff_size -1: 0] Buff_Array [word_size -1: 0];
  wire   cell_3 = Buff_Array[3], cell_2 = Buff_Array[2];
  wire   cell_1 = Buff_Array[1], cell_0 = Buff_Array[0];
  integer k;

  always @ (posedge clock) begin
   if (reset == 1) for (k = 0; k <= buff_size -1; k = k+1)
     Buff_Array[k] <= 0;
   else for (k = 1; k <= buff_size -1; k = k+1) begin
     Buff_Array[k] <= Buff_Array[k-1];
     Buff_Array[0] <= Data_in;
   end
  end
endmodule

module Circular_Buffer_2 # (parameter buff_size = 4, word_size = 8) (
  output [word_size -1: 0] cell_3, cell_2, cell_1, cell_0,
  input  [word_size -1: 0] Data_in,
  input  clock, reset
);

  reg    [buff_size -1: 0] Buff_Array [word_size -1: 0];
  wire   cell_3 = Buff_Array[3], cell_2 = Buff_Array[2];
  wire   cell_1 = Buff_Array[1], cell_0 = Buff_Array[0];
  integer k;
  parameter                           write_ptr_width = 2;    // Width of write
                                                                 pointer
  parameter                           max_write_ptr = 3;
  reg    [write_ptr_width -1 : 0]     write_ptr;              // Pointer for writing

  always @ (posedge clock) begin
   if (reset == 1 ) begin
     write_ptr <= 0;
     for (k = 0; k <= buff_size -1; k = k+1) Buff_Array[k] <= 0;
   end
   else begin
```

[32]See Problem 9.15 at the end of the chapter.

```
      Buff_Array[write_ptr] <= Data_in;
      if (write_ptr < max_write_ptr) write_ptr <= write_ptr + 1; else write_ptr <= 0;
    end
  end
endmodule
```

End of Example 9.10

9.7 Asynchronous FIFOs—Synchronization across Clock Domains

A FIFO (first-in, first-out memory) consists of a stack of registers systematically controlled by an external unit that manages the traffic of data to and from the FIFO. The reading and writing operations of a FIFO are similar to those of circular buffers.[33] A FIFO's architecture provides access to at most two independently addressed register cells at a time (one for writing and one for reading). All the cells of a circular buffer are available concurrently. A FIFO has two address pointers, one for writing to the next available cell, and another one for reading the next unread cell. A circular buffer provides a fixed window of samples of a data stream.

FIFOs operate differently from a circular buffer, because a FIFO's output is a single word, which is read on command. The pointers for reading from and writing to a FIFO are relocated independently and dynamically when commands to read and/or write are executed, rather than sequentially and continuously, as in the case of a circular buffer (see Figure 9-42). The stack of registers in the datapath unit of the FIFO shown in Figure 9-44 can receive data until it is full, and data can be read from the FIFO until it is empty. Two pointers, *write_ptr* and *read_ptr*, provide the address of the next available cell for writing and reading, respectively, and move after each associated operation. It is essential that writing not occur when the FIFO is full, and that reading not occur when the FIFO is empty. Two status flags, *stk_full* and *stk_empty* are used by the FIFO's control unit to prevent operations that would corrupt (write to a full stack) or duplicate (read from an empty stack) data.

A FIFO has separate address busses and datapaths supporting simultaneous reading and writing of data, and may have additional status lines indicating the condition of the stack (*stk_almost_full*, *stk_half_full*, etc). In dynamic operation, *read_ptr* "chases" *write_ptr* in a circular path, from bottom to top, from bottom to top, etc. If the pointers are co-located, the stack may be empty or full. A mere comparison of the pointers cannot distinguish between these two conditions. If the reading and writing operations are governed by a common clock, the gap between the pointers can be monitored by an up-down counter that is sized by one bit wider than the pointers.

[33]See Problem 11 in Chapter 8.

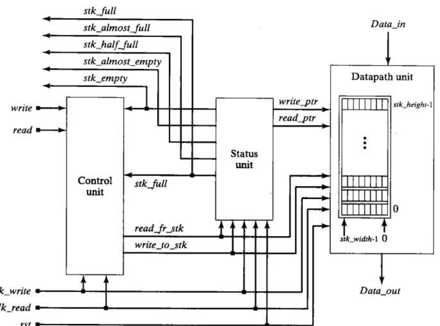


FIGURE 9-44 Block diagram of a FIFO with a status unit and separate clocks for reading and writing.

When a pointer counter is full (i.e., counting from 0 to *stk_height* -1, it holds the value *stk_height* -1), it rolls over to 0 at the next clock; the oversized gap counter does not. Its count will, at most, exceed the stack height by 1 if writing and reading are always done correctly. If the pointers are equal and the gap is 0, the stack is empty. If the pointers are equal and the gap counter is equal to the height of the stack, the stack is full. The status unit in Figure 9-44 contains the counters that monitor writing and reading and generates the status flags.

9.7.1 Simplified Asynchronous FIFO

FIFOs are either synchronous or asynchronous, depending on their clocking scheme. A common clock controls the reading and writing activity of a *synchronous* FIFO. The status unit of a synchronous FIFO is simplified because a simple up-down counter can be controlled by the common clock to monitor the gap between the pointers and detect full and empty conditions. On the other hand, two independent clocks having different frequency and/or phase separately control the read and write pointers of an *asynchronous* FIFO. Consequently, an up/down counter, which can be controlled by only a single clock, cannot be used to monitor the gap between the pointers. Generation of the critical status flags (*stk_full* and *stk_empty*) of an asynchronous FIFO requires a more elaborate status unit.

The simple FIFO, which was introduced in Chapter 8, had a common clock for reading and writing, and could not read and write simultaneously. Here we will first discuss a simplified asynchronous FIFO[34] whose read and write operations are controlled by different clocks, without considering the need to synchronize the transfer of data between the clock domains. Then we will address the issues of metastability and synchronization to model an asynchronous FIFO.

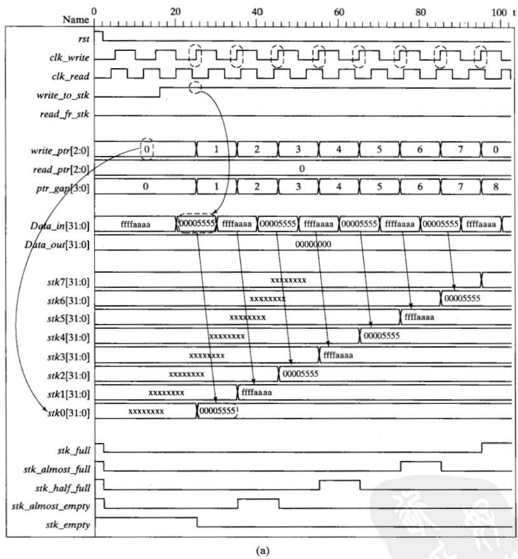
The Verilog model of the FIFO buffer, *FIFO_Dual_Port* and a testbench, *t_FIFO_Dual_Port,* are shown below. The model supports simultaneous reading and writing *at the same location* in the FIFO. Note that in this model the pointers *write_ptr* and *read_ptr* have been sized to wrap at the stack boundary, so the condition of an empty stack cannot be distinguished from the condition of a full stack by comparing the pointers. Instead, the pointers are derived from counters which are one bit wider, by forming *ptr_gap* as the difference between the counts held in *wr_cntr* and *rd_cntr.* The contents of *ptr_gap* are effectively incremented, decremented, or held, depending on the condition of the stack and whether a write operation and/or a read operation is attempted. When the value stored in *ptr_gap* reaches *stk_height* (e.g., 8), the FIFO is full. If the stack is full, only a read is allowed; if the stack is empty, only a write is allowed. The reset signal initializes the counters, but not the memory registers of the FIFO.

The simulation results in Figure 9-45 demonstrate write and read operations of the FIFO and show the activity of the pointers and the status signals, with and without concurrent read and write activity. Figure 9-45(a) shows an initial assertion of *rst*, followed by assertion of *write_to_stk*. Assertion of *rst* causes all of the status flags to be asserted, indicating that the FIFO is not available for either writing or reading. When *rst* is de-asserted, all but *stk_empty* are de-asserted. Assertion of *write_to_stk* causes the contents of *Data_in* to be transferred to *stk*0[31: 0] at the next active edge of *clk_write* (i.e., writing occurs from bottom to top). Concurrently, *stk_empty* is de-asserted. At subsequent active edges of *clk_write, Data_in* is transferred to the FIFO location specified by *write_ptr*, and the status flags are adjusted according to the condition of the FIFO.

Figure 9-45(b) shows simultaneous assertions of *write_to_stk* and *read_fr_stk*. The first active edge of *clk_write* is ignored, because the stack is full. The first active edge of *clk_rd*, with *read_fr_stk* asserted, causes the output register, *Data_out*[31: 0], to be loaded with 00005555_H, the contents of *stk*0[31: 0], as specified by *read_ptr*. This activity de-asserts *stk_full*. Then the next active edge of *clk_write* loads *Data_in*[31: 0] into *stk*0[31: 0], and so forth.

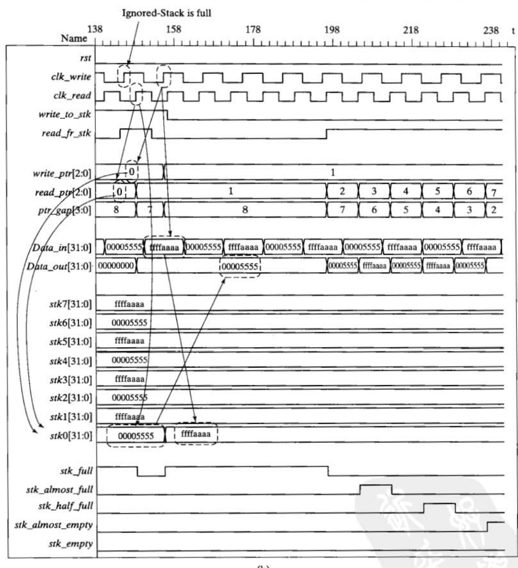
Figure 9-45(c) shows additional activity, including a running reset. Note: These examples demonstrate the operation of the machine without logic for synchronization between the clock domains. The model is useful for exploring the operation of a FIFO, but its simulation cannot provide evidence that the machine will operate correctly in an asynchronous environment, because the lack of coherence between operations in

[34]Logic for synchronization is omitted. Also, note that the output of the FIFO is registered. This is not essential.



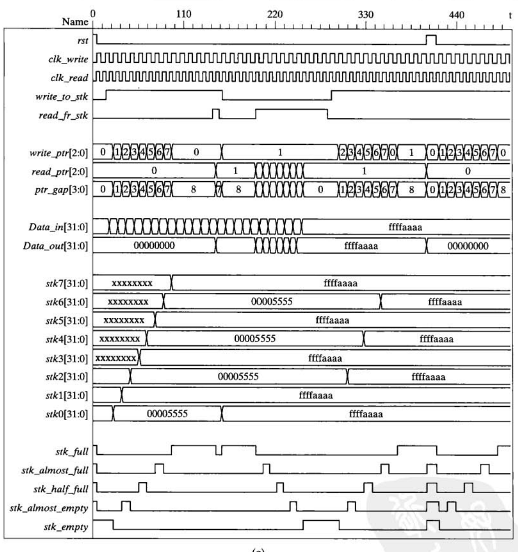
(a)

FIGURE 9-45 Simulation results for a simplified asynchronous FIFO: (a) Initial assert of *rst*, (b) simultaneous assertion of *write*_to_*stk* and *read_fr_stk*, and (c) status signal behavior and recovery from a running reset.



(b)

FIGURE 9-45 Continued



(c)

FIGURE 9-45 Continued

the clock domains guarantees that metastability will be triggered on a regular basis as attempts are made to read from the same cell to which data is being written.

```
module FIFO_Dual_Port # (parameter
  word_width = 32,                                         // Width of stack and data paths
  stk_ptr_width = 3                                        // Width of pointers into stack
)(
  output  [word_width -1: 0]     Data_out,                 // Data path from FIFO
  output                         stk_full,                 // Status flags
                                 stk_almost_full,
                                 stk_half_full,
                                 stk_almost_empty,
                                 stk_empty,
  input   [word_width -1: 0]     Data_in,                  // Data path into FIFO
  input                          write,                    // Flag controlling a write to
                                                              the stack,
                                 read,                     // Flag controlling a read from
                                                              the stack
  input                          clk_write,                // Clock to synchronize writes
                                 clk_read,                 // clock to synchronize reads
                                 rst
);
  wire    [stk_ptr_width -1: 0]  write_ptr, read_ptr;

  FIFO_Control_Unit M0_Controller (
    .write_to_stk(write_to_stk),
    .read_fr_stk(read_fr_stk),
    .write(write),
    .read(read),
    .stk_full(stk_full),
    .stk_empty(stk_empty)
  );
  FIFO_Datapath_Unit M1_Datapath(
    .Data_out(Data_out),
    .Data_in(Data_in),
    .write_ptr(write_ptr),
    .read_ptr(read_ptr),
    .write_to_stk(write_to_stk),
    .read_fr_stk(read_fr_stk),
    .clk_write(clk_write),
    .clk_read(clk_read),
    .rst(rst)
  );
```

```
 FIFO_Status_Unit M2 (
  .write_ptr(write_ptr),
  .read_ptr(read_ptr),
  .stk_full(stk_full),
  .stk_almost_full(stk_almost_full),
  .stk_half_full(stk_half_full),
  .stk_almost_empty(stk_almost_empty),
  .stk_empty(stk_empty),
  .write_to_stk(write_to_stk),
  .read_fr_stk(read_fr_stk),
  .clk_write(clk_write),
  .clk_read(clk_read),
  .rst(rst)
  );
endmodule

// Control Unit
module FIFO_Control_Unit (
 output write_to_stk, read_fr_stk,
 input  write, read, stk_full, stk_empty
);
 assign write_to_stk = write && (!stk_full);
 assign read_fr_stk = read && (!stk_empty);
endmodule

// Datapath Unit
module FIFO_Datapath_Unit # (parameter word_width = 32, stk_height = 8,
  stk_ptr_width = 3)(
 output reg     [word_width -1: 0]     Data_out,
 input          [word_width -1: 0]     Data_in,
 input          [stk_ptr_width -1: 0]  write_ptr, read_ptr,
 input                                 write_to_stk, read_fr_stk,
 input                                 clk_write, clk_read, rst
);
 reg            [word_width -1: 0]     stk [stk_height -1 : 0];    // memory
                                                                    array

 always @ (posedge clk_write) if (write_to_stk) stk [write_ptr] <= Data_in;
 always @ (posedge clk_read) if (read_fr_stk) Data_out <= stk [read_ptr];
endmodule

// Status Unit for Synchronous FIFO
module FIFO_Status_Unit # (parameter stk_ptr_width = 3, stk_height = 8,
 HF_level = stk_height >> 1,                       // Half full level, e.g., 4
 AF_level = (stk_height – HF_level) >> 1,          // Almost full level, e.g., 6
 AE_level = (HF_level) >> 1,                       // Almost empty level, e.g., 2
)(
 output [stk_ptr_width -1: 0]         write_ptr,
 output [stk_ptr_width -1: 0]         read_ptr,
 output                               stk_full, stk_almost_full, stk_half_full,
                                      stk_almost_empty, stk_empty,
```

```
 input                              write_to_stk, read_fr_stk,
 input                              clk_write, clk_read, rst
);
 wire    [stk_ptr_width: 0]         wr_cntr;
 wire    [stk_ptr_width: 0]         wr_cntr_G;
 wire    [stk_ptr_width: 0]         rd_cntr;
 wire    [stk_ptr_width: 0]         ptr_gap = wr_cntr - rd_cntr;     // 2s comp gap
                                                                        between ptrs

 // Stack status signals
 assign stk_full = (ptr_gap == stk_height) || rst;
 assign stk_almost_full = ((wr_cntr - rd_cntr) == AF_level) || rst;
 assign stk_half_full = ((wr_cntr - rd_cntr) == HF_level) || rst;
 assign stk_almost_empty = ((wr_cntr - rd_cntr) == AE_level) || rst;
 assign stk_empty = (wr_cntr == rd_cntr) || rst;

 wr_cntr_Unit M0 (wr_cntr, write_ptr, write_to_stk, clk_write, rst);
 rd_cntr_Unit M1 (rd_cntr, read_ptr, read_fr_stk, clk_read, rst);
endmodule

module wr_cntr_Unit # (parameter stk_ptr_width = 3)(
 output reg [stk_ptr_width: 0] wr_cntr,
 output [stk_ptr_width -1: 0] write_ptr,
 input write_to_stk, clk_write, rst
);
 assign write_ptr = wr_cntr [stk_ptr_width -1: 0];

 always @ (posedge clk_write, posedge rst)
  if (rst) begin wr_cntr <= 0; end
  else if (write_to_stk) begin
   wr_cntr <= wr_cntr + 1;
  end
endmodule

module rd_cntr_Unit # (parameter stk_ptr_width = 3)(
 output reg [stk_ptr_width: 0] rd_cntr,
 output [stk_ptr_width -1: 0] read_ptr,
 input read_fr_stk, clk_write, rst
);
 assign read_ptr = rd_cntr [stk_ptr_width -1: 0];

 always @ (posedge clk_write, posedge rst)
  if (rst) begin rd_cntr <= 0; end
  else if (read_fr_stk) begin
   rd_cntr <= rd_cntr + 1;
  end
endmodule

module t_FIFO_Dual_Port();            // Used to test only the FIFO, without
                                        synchronization
 parameter                           stk_width = 32;
 parameter                           stk_height = 8;
 parameter                           stk_ptr_width = 4;
```

```
 wire    [stk_width -1: 0]          Data_out;
 wire                               write;
 wire                               stk_full, stk_almost_full, stk_half_full;
 wire                               stk_almost_empty, stk_empty;
 reg     [stk_width -1: 0]          Data_in;
 reg                                write_to_stk, read_fr_stk;
 reg                                clk_write, clk_read, rst;
 wire    [stk_width -1: 0]          stk0, stk1, stk2, stk3, stk4,
                                    stk5, stk6, stk7;

 assign stk0 = M1.M1.stk[0];        // Probes of the stk
 assign stk1 = M1.M1.stk[1];
 assign stk2 = M1.M1.stk[2];
 assign stk3 = M1.M1.stk[3];
 assign stk4 = M1.M1.stk[4];
 assign stk5 = M1.M1.stk[5];
 assign stk6 = M1.M1.stk[6];
 assign stk7 = M1.M1.stk[7];

FIFO_Dual_Port M1 (Data_out,
  stk_full, stk_almost_full, stk_half_full,
  stk_almost_empty, stk_empty,
  Data_in,
  write_to_stk, read_fr_stk,
  clk_write, clk_read, rst);

 initial #500 $finish;
 initial fork
  rst = 1; #5 rst = 0;
  #400 rst = 1; #412 rst = 0;
 join

 initial begin clk_write = 0; forever #5 clk_write = ~clk_write; end
 initial begin clk_read = 0; forever #4 clk_read = ~clk_read; end

 // Data transitions
  initial begin Data_in = 32'hFFFF_AAAA;
   @ (posedge write_to_stk);
    repeat (24) @ (negedge clk_write) Data_in = ~Data_in;
  end

 // Write to FIFO
  initial fork
   write_to_stk = 0;
   begin #16 write_to_stk = 1; #140 write_to_stk = 0; end
   begin #286 write_to_stk = 1; end
  join

 // Read from FIFO
 initial fork
  begin #0 read_fr_stk = 0; end
```

```
    begin #144 read_fr_stk = 1; #8 read_fr_stk = 0; end
    begin #196 read_fr_stk = 1; #86 read_fr_stk = 0; end
  join
endmodule
```

9.7.2 Clock Domain Synchronization for an Asynchronous FIFO

The computational activity in the separate domains of an asynchronous FIFO's input and output data are synchronized by separate clocks, with writing to the FIFO governed by one clock, and reading to the FIFO by another clock. Writing and reading can occur simultaneously. This allows the FIFO to act as a buffer between two clock domains. When data are passed from one domain into another and the clock of the destination domain is not related to the clock of the source domain, the active edges of the two clocks are moving relative to each other, and the register operations in the two domains are asynchronous relative to each other. The read and write operations have no effect on each other when the pointers are not co-located, which provides a buffering mechanism for transferring data between the domains. However, if the write and read pointers are ever co-located (a highly likely event), simultaneous writing and reading may result in a condition of metastability in the registers that are to receive data, unless the status unit is modified to deal with co-located pointers and metastability.

To appreciate why metastability can occur when the pointers of a FIFO are co-located, consider the situation in which the FIFO is full: the datapath into the storage register at the location addressed by *write_ptr* would recirculate the data at the output of the register back to its input, at each clock. If *read_ptr* is addressing the same location as *write_ptr* and a read command is executed, *stk_full* could de-assert in the setup window of the flip-flops and the datapath would switch from a configuration of recirculation to one of allowing external data to enter the register. Thus, the datapath of the register could be unstable when it should be stable, and the flip-flops could enter the metastable condition. This situation must be prevented. Conversely, suppose the pointers are co-located and *stk_empty* is asserted. The output register of flip-flops would be holding its data by recirculating it through a mux to the inputs of the register. Executing a write command would de-assert *stk_empty*, causing the datapath into the flip-flops to be addressed by *read_ptr*. This action could happen in the setup interval of the flip-flops, resulting in metastability. These two conditions of metastability can be prevented by (1) synchronizing *rd_cntr* to *clk_write* and comparing the synchronized value to *wr_cntr* to determine whether the stack is full and (2) by synchronizing *wr_cntr* to *clk_read* and comparing the synchronized count to *rd_cntr* to determine whether the stack is empty. Alternatively, handshake signals can be used to govern the exchange of data and circumvent this problem, but the transfer rate is lower than can be achieved by alternatives. In practice, high-performance parallel interfaces between independent clock domains are implemented with a FIFO using a dual-port RAM memory with additional logic to assure synchronization and safeguard against metastability.

The read and write counters of a dual-port FIFO must be synchronized to their opposite domains. Using binary counters would require passing multi-bit data across the boundary of the clock domains, a practice that is to be avoided because it would

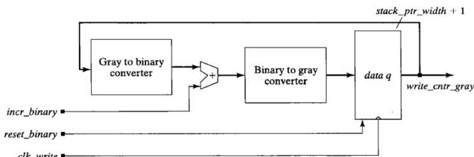


FIGURE 9-46 Code converters for synchronization within an asynchronous FIFO.

require synchronizing multiple bits of data. The dynamic nature of the synchronized binary counts is such that bogus indications of full and/or empty would be highly likely in operation. The alternative is to use Gray code counters because only one bit of a counter would be changing at each clock.

In practice,[35] a register is used to hold the Gray-code count, as shown in Figure 9-46. The Gray-code count is generated by converting a binary count to its Gray-code counterpart. In general, this scheme is simpler than a direct implementation of a Gray-code counter. The additional logic in Figure 9-46 generates the next Gray-code value from the present value by converting it to a binary value, and incrementing the result (shown for generating the Gray code of the words written into the FIFO). The signal *incr_binary* determines whether the binary value is incremented or not, and the size of the increment (usually 1). The incremented binary count is passed through a binary-to-Gray-code converter to the data inputs of the corresponding flip-flops, forming the next Gray code value. The Gray code itself is passed from the flip-flops to synchronizers in the target clock domain, where the status signals are generated to determine whether the FIFO is full or empty. The synchronizer that is used (see Figure 5-38) depends on whether the synchronization is done in the fast clock domain or the slow clock domain, to produce the synchronized gray code values for use in determining the full or empty status of the FIFO.

Figure 9-47 shows the status unit for synchronizing the transfer of data across a clock domain boundary with a dual-port FIFO. The unit has an inner core (shaded background) consisting of the logic for controlling the register operations of the FIFO, together with additional blocks of logic that accomplish the synchronization required by the *stk_full* and *stk_empty* flags. The inner core is identical to the status unit for a synchronous FIFO. To synchronize *wr_cntr* to *clk_read*, the next value of *wr_cntr* (i.e., the input to the register) is converted to a Gray code value and registered by *B2G_reg*. Its registered Gray code output, *wr_cntr_G* is synchronized by the

[35]See technical articles available at www.sunburst-design.com

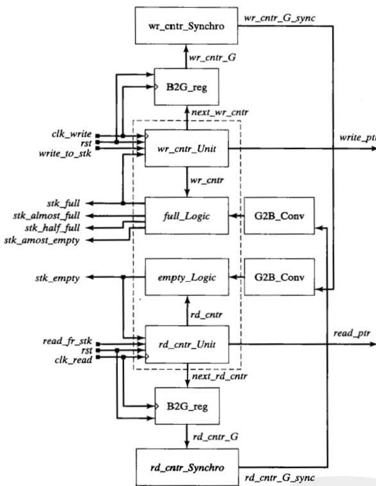


FIGURE 9-47 Status units for synchronous and asynchronous FIFOs.

block *wr_cntr_Synchro* and then converted back to a binary count value by *G2B_Conv* and passed to the logic for forming *stk_empty*.[36] The logic to form *stk_full* is similar, with the exception that the synchronizers are different. The Verilog model (see Example 9.11 below) of the status unit is flexible. It can be used with a common clock and only the inner core of logic to model a synchronous dual-port FIFO, or with

[36]The models for the binary to Gray and Gray to binary code converters are based on those presented in technical papers at www.sunburst-design.com. Testbenches are available at the web site for this text.

separate clocks and the synchronized status unit. The control unit and the datapath unit are the same in both cases.

The status of the asynchronous FIFO can be determined by comparing binary or Gray code values. The model given below compares binary codes, using a synchronized Gray code as an intermediate value. For example, the FIFO is full if *wr_cntr – rd_cntr_B_synch = stack_height*. Similarly, the FIFO is empty if *wr_cntr_B_synch – read_cntr* = 0, i.e., *wr_cntr_synch = rd_cntr*. *re_cntr_B_synch* is derived from a synchronized Gray code value.

Comparing *rd_cntr* to *wr_cntr_synch* allows data to be read from the FIFO at the rate of the fast clock,[37] but guarantees that *empty* is generated at an active edge of *clk_read*, enabling the reading activity to suspend before stale data is re-read. Likewise, comparing *rd_cntr_synch* to *wr_cntr* allows data to be written at the rate of the slow clock, with assurance that *stk_full* will be generated at the active edge of *clk_write* to prevent overwriting data that has not yet been read.

The system for generating *stk_full* and *stk_empty* is inherently pessimistic. The signal *stk_full* is asserted immediately upon the FIFO becoming full, but *rd_cntr_synch* has a latency (e.g., two clock cycles), with the result that the full signal could be asserted longer than necessary. Likewise, *stk_empty* is generated immediately in the domain of the output, but *wr_cntr_synch* arrives with latency. Thus *stk_empty* will be asserted longer than necessary. Neither instance of pessimism is problematic, and both are compatible with the aim of not writing to a full FIFO or reading from an empty FIFO.

Example 9.11

The multichannel circuit in Figure 9-48 has four channels of serial bit streams originating in a 100 MHz clock domain, with each passing through a serial-to-parallel converter that forms a 32-bit word for transfer to a dual-port FIFO. Data are directed to each serial-to-parallel converter at a rate of 100 MHz; a processor operating with a clock of 133 MHz is to read data from the FIFOs and multiplex and interleave the four channels of data onto a common serial datapath. The format of the arriving data has the LSB (least significant bit) of a word arriving first. The commands to write data to a FIFO originate in the domain of *clock_100MHz*; the commands to read data from the FIFOs are issued by the processor (not shown) in the domain of *clock_133MHz*. Data flows continuously from the 100 MHz domain, so the external processor must manage the reading of data to prevent the loss of data, which will occur if a FIFO is full when it receives a write request. We will consider a single channel of data flow.

The serial-to-parallel converter, *Ser_Par_32*, will be implemented as a 32-bit shift register, and will be controlled by the state machine described by the ASMD chart and block diagram in Figure 9-49. The state machine generates the signals *shift* and *incr to* control the datapath register *Data_out* and a counter, *cntr*, respectively. *cntr* indicates the number of serial bits that have been shifted into the datapath register. The asynchronous

[37]Reading data from a FIFO is assumed to occur at a faster rate than writing data to the FIFO.

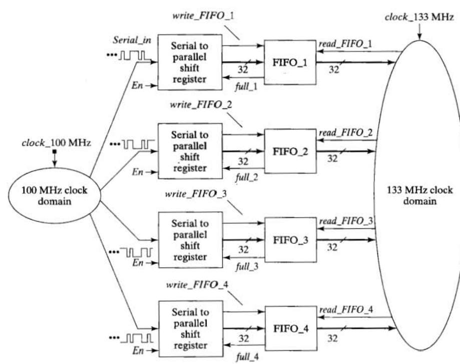


FIGURE 9-48 FIFO-buffered clock domain interface.

reset signal, *rst*, clears *cntr* and *Data_out*, and directs the state of the machine to *S_idle,* where it asserts *ready* and remains until *En* is asserted by an external unit. At the first clock with *En* asserted, the machine shifts a bit into *Data_out*, increments *cntr*, and makes a transition to *S_1*, where it remains until *cntr* reaches the upper limit of its count range (e.g., the limit of a 5-bit counter is 31). When *cntr* reaches its limit, the machine transitions to *S_2* while loading the last bit into the shift register. Signal *write* is asserted in *S_2*, to cause an external unit (e.g., a FIFO) to read the word from *Data_out*. Depending on whether *En* and *full* are asserted by the external environment, the machine transition to *S_1* to stream the bits of the next word of data, or returns to *S_idle*.

The transitions of the serial bit stream of data at the input to the serial-to-parallel converter occur on the falling edge of *clock_100MHz*, and the data are shifted into the register on the rising edge. *write* is asserted for a duration of one cycle of *clk*.[38] The control unit of the serial to parallel converter has two additional outputs, *pause_full* and *pause_En_b*, which communicate with the datapath to handle a situation where *En*

[38]A problem at the end of the chapter addresses the simulation and verification of *Ser_Par_Conv_32*.

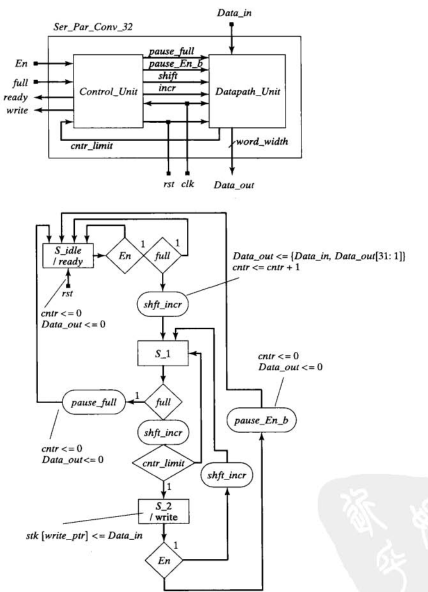


FIGURE 9-49 ASMD chart and block diagram for a serial-to-parallel converter.

is de-asserted, or when the FIFO is full. In this model, it is assumed that the serial-to-parallel conversion will read 32 bits of serial data before checking to see whether *En* remains asserted. If not, the machine returns to *S_idle* to await assertion. Also, the machine returns to *S_idle* if *full* is asserted while the machine is in *S_1*. *pause_En_b* and *pause_full* cause the same datapath operations to reset *cntr* and flush the output register of the serial-parallel converter. However, the testbench distinguishes between the two signals in order to manage the presentation of patterns to the machine and avoid skipping a pattern.

```
module Ser_Par_Conv_32 # (parameter word_width = 32)(
 output [word_width -1: 0] Data_out,
 output                    ready,
 output                    write,
 input                     Data_in, En, full, clk, rst
);
 wire                      pause_full, pause_En_b, shft_incr, cntr_limit;

 Control_Unit M0_Controller (ready, write, pause_full, pause_En_b, shft_incr, En,
   full, cntr_limit, clk, rst);
 Datapath_Unit M1_Datapath (Data_out, cntr_limit, Data_in, pause_full,
   pause_En_b, shft_incr, clk, rst);
endmodule

module Control_Unit (output ready, write,
 output reg        pause_full, pause_En_b, shft_incr,
 input             En, full, cntr_limit, clk, rst);
 parameter         S_idle = 0, S_1 = 1, S_2 = 2;
 reg [1: 0]        state, next_state;
 assign ready = (state == S_idle);
 assign write = (state == S_2);

 always @ (posedge clk, posedge rst)
  if (rst) begin state <= S_idle; end
  else state <= next_state;

 always @ (state, En, full, cntr_limit) begin
  pause_full = 0;
  pause_En_b = 0;
  shft_incr = 0;
  next_state = S_idle;
  case (state)
   S_idle:        if (En && (!full)) begin next_state = S_1; shft_incr = 1; end
                  else next_state = S_idle;

   S_1:           if (full) begin next_state = S_idle; pause_full = 1; end
                  else begin shft_incr = 1;
                   if (cntr_limit) next_state = S_2; else next_state = S_1;
                  end
```

```
    S_2:          if (En) begin shft_incr = 1; next_state = S_1; end
                  else begin pause_En_b = 1; next_state = S_idle; end

    default:      next_state = S_idle;
   endcase
  end
endmodule

module Datapath_Unit # (parameter word_width = 32, cntr_width = 5)(
 output reg [word_width -1: 0]    Data_out,
 output                           cntr_limit,
 input                            Data_in, pause_full, pause_En_b, shft_incr,
                                   clk, rst);
 reg    [cntr_width -1: 0]        cntr;

 always @ (posedge clk, posedge rst)
  if (rst) begin cntr <= 0; end
  else if (pause_full || pause_En_b) cntr <= 0;
  else if (shft_incr) cntr <= cntr +1;

 always @ (posedge clk, posedge rst)
  if (rst) Data_out <= 0;
  else if (pause_full || pause_En_b) Data_out <= 0;
  else if (shft_incr) Data_out <= {Data_in, Data_out [word_width -1: 1]};

 assign cntr_limit = (cntr == word_width -1);
endmodule
```

Each FIFO in Figure 9-48 must generate a synchronized version of its write counter for the purpose of comparing it to the read pulse counter in the domain of the faster clock, *clock_133MHz*, for use in determining whether the FIFO is empty[39]. The synchronizers that were shown in Figure 5-38 are candidates for synchronizing the write counter of the FIFO. To determine which of the two synchronizer circuits to use, note that *write_to_stk* asserts for one cycle of *clock_100MHz*, so the pulse has a width of $\Delta_{write} = 1/T_{clock_100MHz} = 1/(10^8) = 10$ ns. The period of *clock_133MHz* is, as shown in Figure 9-50, $T_{clock_133MHz} = 1/(133 \times 10^6) = 7.5$ ns. Because the width of the asynchronous pulse of a bit of the Gray code value is greater than the period of the clock to which it is to be synchronized, we choose the circuit in Figure 5-38(a), a two-stage shift register synchronizer. A similar analysis leads to the choice of the circuit in Figure 5-38(b) to synchronize *rd_cntr* to *clk*_100.

The machine, *FIFO_Channel*, uses *Ser_Par_Conv_32* and *FIFO_Dual_Port*, but with the status unit given below. Before presenting simulation results for *FIFO_Channel*, we provide a word of caution—thorough timing analysis of FIFOs and their companion circuitry for synchronization across boundaries of clock domains must be done in general. This requires timing analysis of the synthesized gate-level circuit with back annotated delay values extracted from the actual cell placement and routing. Without

[39]Note that the data is written to the FIFO under the control of *wr_to_stk*, which is synchronized to *clock_100MHz*.

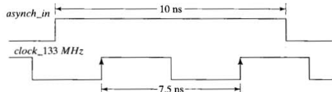


FIGURE 9-50 The duration of the asynchronous write pulse covers two edges of *clock_133_MHz*.

such detailed analysis, the verification of the machine is incomplete. This step will not be demonstrated here.

```
module FIFO_Channel # (
 parameter                  word_width = 32)(      // Width of FIFO stack and
                                                     data paths
 output [word_width -1: 0]  Data_out_FIFO,         // Data path from FIFO
 output                     ready,
 input
  Data_in,                                          // Serial 100 MHz data
  En,                                               // Launches activity
  read,                                             // reads data from FIFO
  clk_write,                                        // Clock to synchronize writes
  clk_read,                                         // clock to synchronize reads
  rst
);
 wire reg [word_width -1: 0] Data_out_Ser_Par;     // Data path from Ser_Par
 wire                       Data_in_Ser_Par;       // Data path into Ser_Par
 wire                       stk_full,              // Status flags
                            stk_almost_full, stk_half_full, stk_almost_empty,
                             stk_empty;

 FIFO_Dual_Port M0 (
  .Data_out(Data_out_FIFO),
  .stk_full(stk_full),
  .stk_almost_full(stk_almost_full),
  .stk_half_full(stk_half_full),
  .stk_almost_empty(stk_almost_empty),
  .stk_empty(stk_empty),
  .Data_in(Data_out_Ser_Par),
  .write(write),
  .read(read),
  .clk_write(clk_write),
  .clk_read(clk_read),
  .rst(rst));

 Ser_Par_Conv_32 M1(
  .Data_out(Data_out_Ser_Par),
  .ready(ready),
```

```
  .write(write),
  .Data_in(Data_in),
  .En(En),
  .full(stk_full),
  .clk(clk_write),
  .rst(rst));
endmodule

// Status Unit for Asynchronous FIFO
module FIFO_Status_Unit # (parameter stk_ptr_width = 3, stk_height = 8,
 HF_level = 6, //(stk_height >> 1),                 // Half full level
 AF_level = 4, //(stk_height - HF_level) >> 1,      // Almost full level
 AE_level = 2 //(HF_level) >> 1                     // Almost empty level
)(
 output [stk_ptr_width -1: 0]      write_ptr,
 output [stk_ptr_width -1: 0]      read_ptr,
 output                            stk_full, stk_almost_full, stk_half_full,
                                   stk_almost_empty, stk_empty,
 input                             write_to_stk, read_fr_stk,
 input                             clk_write, clk_read, rst
);
 wire    [stk_ptr_width: 0]        wr_cntr, next_wr_cntr;
 wire    [stk_ptr_width: 0]        wr_cntr_G;
 wire    [stk_ptr_width: 0]        rd_cntr, next_rd_cntr;
 wire    [stk_ptr_width: 0]        rd_cntr_G;
 wire    [stk_ptr_width: 0]        wr_cntr_G_sync, rd_cntr_G_sync;
 wire    [stk_ptr_width: 0]        wr_cntr_B_sync, rd_cntr_B_sync;

 // Stack status signals
 assign stk_full = ((wr_cntr - rd_cntr_B_sync) == stk_height) || rst;
 assign stk_almost_full = ((wr_cntr - rd_cntr_B_sync) == AF_level) || rst;
 assign stk_half_full = ((wr_cntr - rd_cntr_B_sync) == HF_level) || rst;
 assign stk_almost_empty = ((wr_cntr - rd_cntr_B_sync) == AE_level) || rst;
 assign stk_empty = (wr_cntr_B_sync == rd_cntr) || rst;

 wr_cntr_Unit M0_rw_cntr (next_wr_cntr, wr_cntr, write_ptr, write_to_stk,
   clk_write, rst);

 rd_cntr_Unit M1rd_cntr (next_rd_cntr, rd_cntr, read_ptr, read_fr_stk, clk_read, rst);

 B2G_Reg M2_B2G (
  .gray_out(wr_cntr_G),
  .binary_in(next_wr_cntr),
  .wr_rd(write_to_stk),
  .limit(stk_full),
  .clk(clk_write),
  .rst(rst)
 );

 G2B_Conv M3_G2B (
  .binary(wr_cntr_B_sync),
  .gray(wr_cntr_G_sync)
 );
```

```
B2G_Reg M4_B2G (
  .gray_out(rd_cntr_G),
  .binary_in(next_rd_cntr),
  .wr_rd(read_fr_stk),
  .limit(stk_empty),
  .clk(clk_read),
  .rst(rst)
);

G2B_Conv M5_G2B (
  .binary(rd_cntr_B_sync),
  .gray(rd_cntr_G_sync)
);

// Synchronizer Unit

 generate
 genvar k;
 for (k = 0; k <= stk_ptr_width; k = k+1) begin: write_cntr_synchronization
  Synchro_Long_Asynch_in_to_Short_Period_Clock M0 (
   .Synch_out(wr_cntr_G_sync[k]),
   .Asynch_in(wr_cntr_G[k]),
   .clock(clk_read),
   .reset(rst)
 );
 end

 for (k = 0; k <= stk_ptr_width; k = k+1) begin: read_cntr_synchronization
  Synchro_Short_Asynch_in_to_Long_Period_Clock M1 (
   .Synch_out(rd_cntr_G_sync[k]),
   .Asynch_in(rd_cntr_G[k]),
   .clock(clk_write),
   .reset(rst)
  );
 end
 endgenerate
endmodule

module wr_cntr_Unit # (parameter stk_ptr_width = 3)(
 output reg [stk_ptr_width: 0] next_wr_cntr, wr_cntr,
 output [stk_ptr_width -1: 0] write_ptr,
 input write_to_stk, clk_write, rst
);
 assign write_ptr = wr_cntr [stk_ptr_width -1: 0];

 always @ (posedge clk_write, posedge rst)
  if (rst) begin wr_cntr <= 0; end
  else if (write_to_stk) begin
   wr_cntr <= next_wr_cntr;
  end

 always @ (wr_cntr) next_wr_cntr = wr_cntr + 1;
endmodule
```

```
module rd_cntr_Unit # (parameter stk_ptr_width = 3)(
  output reg [stk_ptr_width: 0] next_rd_cntr, rd_cntr,
  output [stk_ptr_width -1: 0] read_ptr,
  input read_fr_stk, clk_write, rst
);
  assign read_ptr = rd_cntr [stk_ptr_width -1: 0];

  always @ (posedge clk_write, posedge rst)
   if (rst) begin rd_cntr <= 0; end
   else if (read_fr_stk) begin
    rd_cntr <= next_rd_cntr;
   end

  always @ (rd_cntr) next_rd_cntr = rd_cntr + 1;
endmodule

module B2G_Reg # (parameter size = 4)(
  output reg [size -1: 0] gray_out,
  input [size -1: 0] binary_in,
  input wr_rd, limit, clk, rst
);
  wire [size -1: 0] next_gray_out;

  always @ (posedge clk, posedge rst)
   if (rst) gray_out <= 0; else if (wr_rd && (!limit)) gray_out <= next_gray_out;

  B2G_Conv M0 (
   .gray(next_gray_out),
   .binary(binary_in)
  );
endmodule

module B2G_Conv # (parameter size = 4)(
  output [size -1: 0] gray, input [size -1: 0] binary);
  assign gray = (binary >> 1) ^ binary;
endmodule

module G2B_Conv # (parameter size = 4)(
  output reg [size -1: 0] binary, input [size -1: 0] gray);
  integer k;
  always @ (gray) for (k = 0; k < size; k = k +1) binary[k] = ^(gray >> k);
endmodule

module Synchro_Short_Asynch_in_to_Long_Period_Clock (
  output reg Synch_out,
  input Asynch_in, clock, reset
);
  reg q1, q2;
  supply1 Vcc;
  wire Clr_q1_q2 = reset || ((!Asynch_in && Synch_out));

  always @ (posedge clock, posedge reset)
```

```
  if (reset) begin
   Synch_out <= 0;
  end
  else Synch_out <= q2;

 always @ (posedge clock, posedge Clr_q1_q2)
  if (Clr_q1_q2) q2 <= 0;
  else q2 <= q1;

 always @ (posedge Asynch_in, posedge Clr_q1_q2)
  if (Clr_q1_q2) q1 <= 0;
  else q1 <= Vcc;
 endmodule

 module Synchro_Long_Asynch_in_to_Short_Period_Clock (
  output reg Synch_out, input Asynch_in, clock, reset);
  reg Synch_meta;

  always @ (posedge clock, posedge reset)
  if (reset) begin Synch_meta <= 0; Synch_out <= 0; end
  else {Synch_out, Synch_meta} <= {Synch_meta, Asynch_in};
 endmodule
```

The testbench used to verify *FIFO_Channel* is given below. The testbench selects a pattern and then presents it to *FIFO_Channel* one bit at a time.

```
module t_FIFO_Channel # (
 parameter word_width = 32, half_cycle_100_MHz = 4, half_cycle_133_MHz = 3);
 wire [word_width -1: 0]   Data_out_FIFO;
 wire                      ready;
 reg                       En, read, clk_write, clk_read, rst;

 FIFO_Channel M0 (Data_out_FIFO, ready, Data_in, En, read, clk_write,
   clk_read, rst);

 wire [word_width -1: 0]   stk0, stk1, stk2, stk3, stk4, stk5, stk6, stk7;

 assign stk0 = M0.M0.M1.stk[0];            // Probes of the stk
 assign stk1 = M0.M0.M1.stk[1];
 assign stk2 = M0.M0.M1.stk[2];
 assign stk3 = M0.M0.M1.stk[3];
 assign stk4 = M0.M0.M1.stk[4];
 assign stk5 = M0.M0.M1.stk[5];
 assign stk6 = M0.M0.M1.stk[6];
 assign stk7 = M0.M0.M1.stk[7];

 initial #8000 $finish;
 initial begin clk_write = 0; forever #half_cycle_100_
  MHz clk_write = ~clk_write; end                          // 100 MHz clock
 initial begin clk_read = 0; forever #half_cycle_133_
  MHz clk_read = ~clk_read; end                            // 133 MHz clock
```

```
initial fork
 En = 0; #18 En = 1;
 #400 En = 0;
 #960 En = 1;
join

initial fork
 read = 0;
 #3500 read = 1;
 #4200 read = 0;
 #7124 read = 1;
join

initial fork
 #0 rst = 1;
 #2 rst = 0;
 #5050 rst = 1;
 #5075 rst = 0;
join

reg [word_width -1: 0       Pattern_buffer [15: 0;
reg [3: 0                   Pattern_ptr;
reg [word_width -1: 0]      Data_word;

assign Data_in = Data_word [0];

always @ (negedge clk_write, posedge rst)
 if (rst) begin Pattern_ptr <= 1; Data_word <= Pattern_buffer [0]; end
else begin
 if (M0.M1.pause_full) begin Data_word <= Pattern_buffer [Pattern_ptr -1]; end
 if (M0.M1.pause_En_b) begin Data_word <= Pattern_buffer [Pattern_ptr]; end

 if (M0.M0.write_to_stk) begin
  Pattern_ptr <= Pattern_ptr + 1;
  Data_word <= Pattern_buffer [Pattern_ptr];
 end
end

always @ (negedge clk_write, posedge rst)

 if (rst) Data_word <= Pattern_buffer [0];
 else if ((M0.M1.shft_incr) && (M0.M1.M0.state != M0.M1.M0.S_idle)) Data_word
  <= Data_word >> 1;

initial fork

 Pattern_buffer [0] = 32'haaaa_aaaa;
 Pattern_buffer [1] = 32'hbbbb_bbbb;
 Pattern_buffer [2] = 32'hcccc_cccc;
 Pattern_buffer [3] = 32'hdddd_dddd;
 Pattern_buffer [4] = 32'heeee_eeee;
 Pattern_buffer [5] = 32'hffff_ffff;
 Pattern_buffer [6] = 32'haaaa_ffff;
 Pattern_buffer [7] = 32'hbbbb_aaaa;
```

```
      Pattern_buffer [8] = 32'ha5a5_5a5a;
      Pattern_buffer [9] = 32'hb5b5_5b5b;
      Pattern_buffer [10] = 32'hcccc_5555;
      Pattern_buffer [11] = 32'hdddd_5555;
      Pattern_buffer [12] = 32'heeee_5555;
      Pattern_buffer [13] = 32'hffff_5555;
      Pattern_buffer [14] = 32'haaaa_5555;
      Pattern_buffer [15] = 32'hbbbb_5555;
    join
  endmodule
```

Figure 9-51(a) shows the results from simulating a suite of test patterns to verify the correct operation of *FIFO_Channel*, the combined serial-parallel converter and FIFO. The pulses of *write_to_stack* and *read_fr_stack* and other significant features are highlighted by shading. Note that, during the first assertion of *En*, *stk*0 gets $aaaaaaaa_H$, and at the second assertion *stk*1 gets $bbbbbbbb_H$. After a brief pause (while *En* is de-asserted), the third pattern ($cccccccc_H$) is loaded into *stk*2, and so forth, until the stack is full. Then, with *stk_full* and with *read* asserted by the testbench, *stk_empty* asserts intermittently, as reading follows writing, with reading waiting for writing to add a word to the stack. This set of tests shows that the FIFO is able to (1) reconstruct the data patterns from the serial-to-parallel converter correctly, without corrupting bits between words and (2) resume the write operation after a pause by de-assertion of *En* or by assertion of *stk_full*. A reset-on-the-fly event demonstrates that the machine recovers correctly and loads the first pattern at the first opportunity to write to the stack. The shaded data shows that the synchronization, code conversions, and status determinations occur correctly.

Figure 9-51(b) zooms into the waveforms to demonstrate that the read operations execute correctly. Eight cycles of *clk_read* transfer words from the stack to *Data_out_FIFO*, from the addresses specified by *read_ptr*. Note that *stk_full* de-asserts with a latency of two cycles of *clk_write*, and then the state transitions from *S_idle* (0) to *S_1* (1). Once the machine enters *S_1*, it takes 31 additional cycles of *clk_write* to load another word to the stack (not shown), so (even though the stack is not full) a write cannot execute immediately.

Figure 9-51(c) shows a sequence in which *En* and *read* are both asserted and the stack is empty. An assertion of *write_to_stk* loads $a5a55a5a_H$ into *stk*0. Two edges of *clk_read* later, *stk_empty* de-asserts, reflecting the latency in comparing *rd_cntr* to *wr_cntr_B_sync*. A comparison of *rd_cntr* and *wr_cntr* would de-assert *stk_empty* immediately, but risk metastability by allowing the stack to be read too close to the edge of *clk_write*. The interval of latency is indicated in Figure 9-51(c). The content of *stk*0 is read to *Data_out_FIFO* at the first edge of *clk_read* after *stk_empty* has de-asserted. Once a stack is empty, the maximum rate at which words can be transferred across the boundary between the clock domains is limited by the rate at which words can be written to the stack: 100 MHz/32 = 3.125 MHz.

Figure 9-52(a) presents simulation results showing that reading from the stack is synchronized by *clk_read*, and that *read_ptr* determines the data that is loaded into *Data_out_FIFO*. Note, however, that the count of *rd_cntr_B_sync* skips values (counts of 1, 4, 5, and 6 are missing). This behavior is explained by the simulation results in

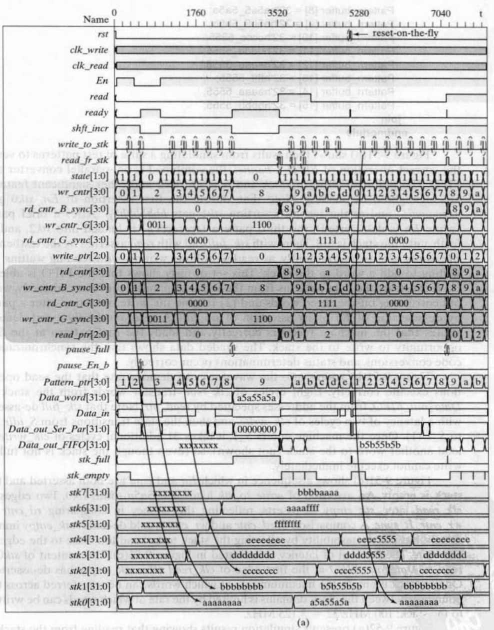


FIGURE 9-51 Simulation results for *FIFO_Channel*: (a) reconstruction of data and correct operation, (b) correct read operation, and (c) operation with assertions of *En* and read while stack is empty.

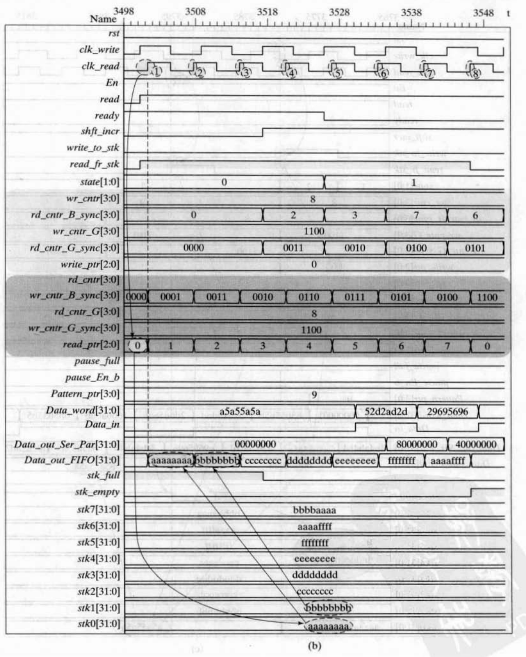


FIGURE 9-51 Continued

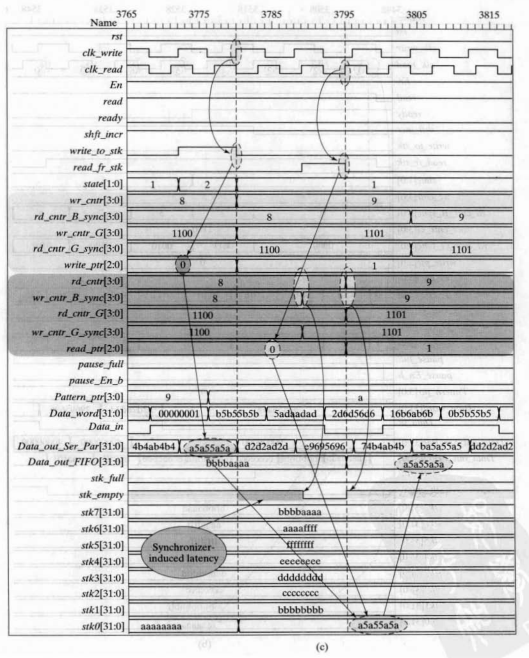
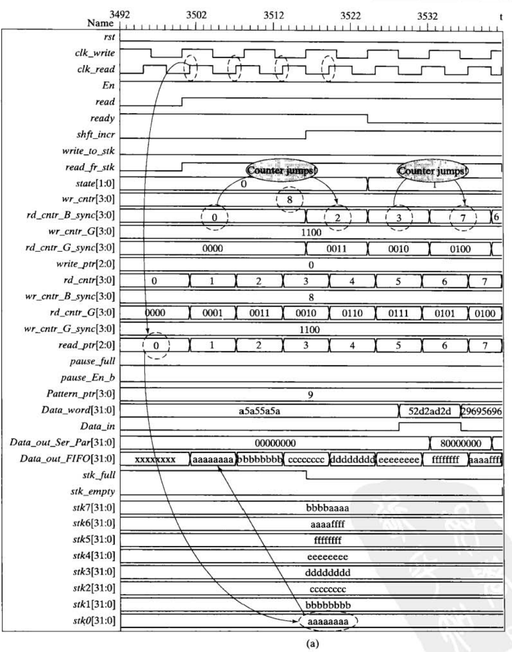
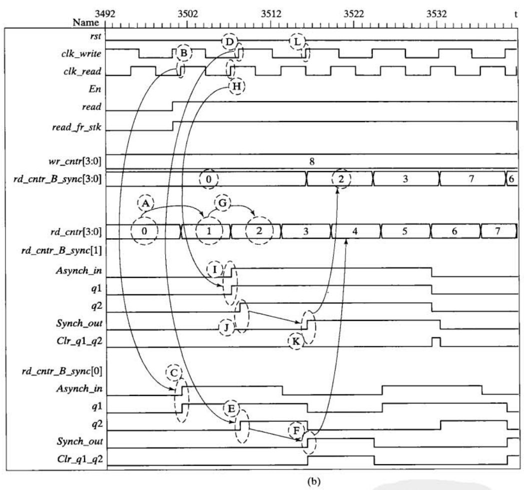


FIGURE 9-51 Continued



(a)

FIGURE 9-52 Simulation of reading from the stack: (a) synchronization by *clk_read*, and (b) accounting for skipped counts of *rd_cntr_B_synch*.



(b)

FIGURE 9-52 Continued

Figure 9-52(b), where the lower two bits of *rd_cntr_B_synch* are shown. With *rd_counter* = 0 (A), the edge of *clk_read* (B) causes the transition to *rd_cntr* = 1 (G). The details of this transition are shown, where the asynchronous value (C) in the domain of *rd_cntr* gets passed to the first stage of the synchronizer (E) at the next edge of *clk_write* (D), and then passes to *Synch_out* (F) and the second edge of *clk_write*. Given the disparity between the clocks frequencies and the alignment of the edges of the clocks, with *clk_read* being faster, the second edge of *clr_read* (H) causes *rd_cntr* to increment to a value of 2 (G). This value passes into its synchronizer (J) and reaches the output of its second stage (K) so that both bits of *rd_cntr_B_synch* have changed in time for the edge of *clk_write* at (L). The intermediate value of *rd_cntr* = 1 is obscured

at *rd_cntr_B_sync* because the relative alignment of edges between *clk_rd* and *clk_write* is such that two bits of the counter can change in the interval of time for the data to pass through two stages of the synchronizers in the slow domain.

End of Example 9.11

REFERENCES

1. van der Hoeven A. *Concepts and Implementation of a Design System for Digital Signal Processor Arrays*. Delft, The Netherlands: Delft University Press, 1990.
2. Bu J. *Systematic Design of Regular VLSI Processors*. Delft, The Netherlands: Delft University Press, 1990.
3. De Micheli G. *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.
4. Gajski D, et al. "Essential Issues in Codesign." In: Staunstrup J., Wolf W, eds. *Hardware/Software Co-Design: Principles and Practices*. Boston, MA: Kluwer, 1997.
5. Gajski D, et al. *High-Level Synthesis: Introduction to Chip Design*. Boston, MA: Kluwer, 1992.
6. Knuth DE. *The Art of Computer Programming*. Vol. 3. *Sorting and Searching*. Reading, MA: Addison-Wesley, 1973.
7. Kehtarnavaz N, Keramat M. *DSP System Design Using the TMS320C6000*. Upper Saddle River, NJ: Prentice-Hall, 2001.
8. Candy JV. *Signal Processing—The Modern Approach*. New York: McGraw-Hill, 1988.
9. Oppenheim AV, Schafer RW. *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1989.
10. McClellan JH, Schafer RW, Yoder MA. *DSP First—A Multimedia Approach*. Upper Saddle River, NJ: Prentice-Hall, 1998.
11. Hagan CJ. *Synthesis of Cascade Integrator Comb Digital Decimation Filters*. Technical Report EAS_ECE_1988_05, Department of Electrical and Computer Engineering, University of Colorado at Colorado Springs, 1998.
12. Stearns SD, David RA. *Signal Processing Algorithms*. Upper Saddle River, NJ: Prentice-Hall, 1988.
13. McClellan JH, et al. *Computer-Based Exercises for Signal Processing Using MATLAB 5*. Upper Saddle River, NJ: Prentice-Hall, 1998.
14. Stonick VL, Bradley K. *Labs for Signals and Systems Using MATLAB*. Boston, MA: PWS, 1996.
15. Kronenburger J., Sebeson J. *Analog and Digital Signal Processing*. Clifton Park, New York: Thomson, 2008.
16. Smith MJ. *Application-Specific Integrated Circuits*. Reading, MA: Addison-Wesley Longman, 1997.
17. Andraka R. "FPGAs Cut Power With 'Pipeline'." *Electronic Engineering Times*, August 7, 2000.
18. Kaeslin, H., *Digital Integrated Circuit Design*. Cambridge: Cambridge University Press, 2008.

PROBLEMS

1. Determine the size of the largest pixel processor array that can be implemented by synthesizing *Image_Converter_Baseline* (see Section 9.2.1) into a Xilinx XCS40/XL FPGA (see Table 8-13).
2. Develop an architecture for implementing a sequential (row-by-row) algorithm realizing the behavior of a halftone image converter having only one processor for an 8×6 array. Develop, verify, and synthesize a Verilog model of the machine. (*Optional*: By performing postsynthesis simulation in a given ASIC technology or an FPGA, determine the maximum rate at which images can be processed by the machine.)
3. The convolution of an N-sample data sequence $\{x[k]\}$ with the impulse response of a digital filter $\{h[k]\}$ with $j = 0, 1, \ldots, N - 1$, produces the filter's output and is defined [19, 20] by

$$y[j] = \sum_{k=0}^{j} x[k]h[j - k]$$

for $j = 0, 1, \ldots, 2N - 2$. (a) Using Verilog constructs, write an NLP describing the convolution algorithm. Note that the NLP can be unrolled to give

$$y[0] = x[0]\,h[0]$$
$$y[1] = x[0]\,h[1] + x[1]\,h[0]$$
$$y[2] = x[0]\,h[2] + x[1]\,h[1] + x[2]\,h[0]$$
$$\ldots$$

A fragment of a DFG for the NLP is given in Figure P9-3; (b) complete the DFG for $N = 3$, (c) using the NLP and the DFG, develop, verify, and synthesize *Convolution_Baseline*, a Verilog model that implements the algorithm, (d) using the DFG, identify a two-stage balanced pipeline architecture for the machine, and (e) develop, verify, and synthesize *Convolution_Pipe*, an implementation of the pipeline determined in (d) and compare the two realizations of the algorithm. What is the minimum number of concurrent processors that could implement the algorithm?

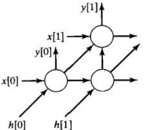
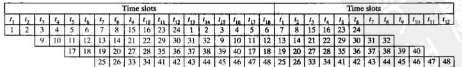


FIGURE P9-3

4. The bubble sort algorithm (see Example 9.1) sorts the elements of an array of N unsigned binary numbers into ascending order. Develop an NLP in pseudocode to (1) search over N unsigned binary numbers to find the largest, (2) then remove it and repeat the search over the remaining $N - 1$ words, etc., until the list is sorted. Develop a temporal DFG for the algorithm and specify an architecture for its implementation. Develop, verify, and synthesize a machine that realizes the algorithm. Compare and discuss the relative merits of the two sorters.
5. Write a Verilog model of *Bubble_Sort_Alternative*, the machine described by the ASMD chart in Figure 9-19. Develop and exercise a testbench to verify the machine.
6. Using the Floyd–Steinberg algorithm (see Section 9.2), develop a Verilog behavioral model of a machine, *GS_Image_Converter*, that implements a Gray scale conversion of an 8×6 array of pixels having a resolution of 8 bits, into a same-sized array of pixels have a resolution of 4 bits. Implement the following versions of the machine: (1) a baseline level-sensitive realization of the algorithm's NLP and (2) an ASMD-based implementation with maximally concurrent processing.
7. Develop an NLP describing an algorithm that computes the histogram of eight equispaced gray levels in an 8×6 pixel image with 8 bits of resolution. Using a DFG for the algorithm, develop, verify, and synthesize (1) a baseline machine implementing the algorithm and (2) an ASMD-based machine realizing the algorithm with maximally concurrent processors.
8. The reservation table in Figure P9-8 reveals how the throughput of *Image_Converter_Concurrent_Processors* could be increased by exploiting the idle time of the processors and concurrently processing two images while conccurrently processing the pixels of each image. Develop, verify and synthesize a machine that will maximize the throughput that can be obtained in processing an 8×6 array, using four concurrent processors. Include an ASMD chart for the control unit of the machine (see Figure 9-13).
9. Develop, verify, and synthesize a frame processor that converts an 8×6 array of pixels with 8-bit resolution into an image with 4-bit resolution. The processor includes three image buffers, and a controller that directs the processing of one image while a second image is being loaded into memory, and a third (converted) image is being sent through the I/O ports, which accommodate 1 byte each. The pipelining of the operations is illustrated below in Figure P9-9. Note: the unit is served by a single data bus, making it necessary to interleave the data that is being sent and received.
10. The DFG shown in Figure 9-5 for a halftone pixel image converter can be used to identify alternative cutsets that define a pipeline architecture for the baseline machine. Identify a cutset that will balance a pair of pipeline stages



Time slots																	
t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}
1	2	3	4	5	6	7	8	15	16	23	24	1	2	3	4	5	6
		9	10	11	12	13	14	21	22	29	30	31	32	9	10	11	12
				17	18	19	20	27	28	35	36	37	38	39	40	17	18
						25	26	33	34	41	42	43	44	45	46	47	48

Time slots											
t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
7	8	15	16	23	24						
13	14	21	22	29	30	31	32				
19	20	27	28	35	36	37	38	39	40		
25	26	33	34	41	42	43	44	45	46	47	48

FIGURE P9-8

t	Buffer_1	Buffer_2	Buffer_3
	Processing	Loading	Sending
	Sending	Processing	Loading
	Loading	Sending	Processing
↓	Processing	Loading	Sending

⋮

FIGURE P9-9

for an 8 × 6 array of pixels. Specify an architecture, and develop and verify the Verilog model *Image_Converter_1_pipe* that implements the pipeline specified by the cutset. Include an ASMD chart for the control unit of the machine (see Figure 9-13).

11. The Verilog model *FIR_Gaussian_Lowpass* (see Example 9.2) is limited to a seventh-order filter. Develop and verify a re-usable model with (1) parameters and memory for up to 16 filter tap coefficients, and (2) a loop-based algorithm that forms *Data_out* for an allowed order of the filter. When *reset* is asserted, the state of the filter is to return to *S_idle*, where it remains until a signal *Load* is asserted. With the state in *S_idle*, an assertion of *Load* will cause the machine to read a byte of data specifying the order of the filter, and move to state *S_loading*. On subsequent clock edges, the machine remains in *S_loading* and sequentially reads the parameters of the filter. After reading the parameters the state enters *S_running*. While in *S_running*, the machine generates the filtered output (*D_out*) from the input signal (*D_in*) until *reset* is asserted again. Synthesize the model and verify the functionality of the gate-level circuit.
12. Develop, verify, and synthesize parameterized Verilog models of the DF-II and TDF-II IIR filters (see Figure 9-27). The filters must import their tap coefficients from the test bench environment.
13. Compare the results of synthesizing *Circular_Buffer_1* and *Circular_Buffer_2* (see Example 9.9).
14. Verify that the 32-bit serial-to-parallel converter *Ser_Par_Conv_32* operates correctly when the data bits are streamed to the machine continuously (i.e., the state does not return to *S_idle* between successive words).
15. Modify the eight-tap Gaussian FIR in Example 9.2 to have a 4-bit representation of the filter's coefficients, but keeping an 8-bit datapath. Compare this modified filter to the filter modeled by *FIR_Gaussian_Lowpass*. Consider their use of physical resources (e.g., configurable logic blocks in a Xilinx FPGA), their accuracy, and their performance.
16. Implement and compare two different architectures for an eight-tap FIR with a 16-bit datapath. The first is to use the architecture shown in Figure 9-23, which has a shift-register structure that stores and shifts the samples of the input sequence. The second is to implement the FIR with a circular buffer controlled by a state machine.

17. Using MATLAB, design a lowpass FIR filter with passband frequency = 1600 Hz, stopband frequency = 2400 Hz, passband gain = 0 dB, stopband attenuation = 20 dB, and sampling rate = 8000 Hz. The datapath at the input is 32 bits wide, and the tap coefficients are to be stored as 16-bit words. Verify that the filter satisfactorily attenuates an input at 2.5 kHz and 3 kHz. Determine the highest sampling rate that can be achieved in the technology that was used to synthesize the filter. (*Optional*: Implement the filter in an FPGA and demonstrate its operation.)
18. Adaptive digital filters are commonly used to (1) filter noise from data whose statistics are either unknown or time-varying and (2) to extract a model of an unknown system from data describing its input–output response. The parameters of an adaptive digital filter are adjusted dynamically, as the statistics of the data evolve during processing. A feedback loop drives an adaptation process that compares the output of the unknown system with the output of the filter and uses the derived information to adjust the filter's weights. In the structure shown in Figure P9-18 [7], the time-sequence response of the adaptive FIR is to approximate the time sequence of the unknown system, and the tap coefficients of the FIR are to adapt dynamically to reduce the error signal. A least mean square algorithm [18] is used to update the tap coefficients of the FIR by adjusting their values according to

$$b_{k\ \text{new}} = b_{k\ \text{old}} + \delta^* y_{\text{error}}$$

The stepsize adjustment parameter δ is chosen to cause the error sequence to go to 0. If δ is too big the LMS (least mean square) algorithm might not converge; if it is too small, it might converge very slowly. Consider values of δ between 10^{-2} and 10^{-4}. An adaptive FIR has been designed elsewhere [7] to filter the output of an unknown system modeled by a seventh order bandpass IIR, with a sampling rate of 8 kHz, and having $M = N = 7$. The passband of the IIR filter is from $\pi/3$ to $2\pi/3$ rad. The stopband attenuation of the filter is 20 dB. The filter's coefficients that were presented in reference 7 are given in Table P9-17, normalized to make $a_0 = 1$.

(a) Develop and compare two implementations of the IIR filter. One is to use a pair of circular buffers, one to hold samples of the output, and one to hold samples of the input. An eight-cell circular buffer will hold the current output and a window of the last seven outputs; a seven-cell circular buffer will hold seven samples of the input. The other implementation is to use shift registers to

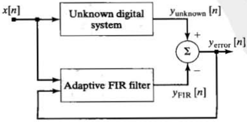


FIGURE P9-18

TABLE P9-17 Coefficients for a seventh-order IIR bandpass filter.

j, k	a_j	b_k
0	1.0000	0.1191
1	0.0179	0.0123
2	0.9409	−0.1813
3	0.0104	−0.0251
4	0.6601	−0.1815
5	0.0342	0.0307
6	0.1129	−0.1194
7	0.0058	−0.0178

hold the samples of data. (*Note:* The implementation will require prescaling and postscaling operations to support the arithmetic operations of the datapath with the finite wordlength of the machine.) (b) Develop a testbench to verify that the IIR acts as a bandpass filter.

19. The nodes of the DFG shown in Figure P9-19 have been annotated with propagation delays. Find the optimal placement of pipeline registers in the circuit.
20. Modify the pipelined FIR filter in Figure 9-41(b) to have coherent datapaths.
21. The DFG in Figure P9-21(a) describes a systolic array processor that implements the matrix product $C = A \times B$, with $c_{ij} = \sum a_{ik}\, b_{kj}$. Each FU of a fully parallel implementation of the processor would require 8 channels of data, 4 multipliers, and 3 adders. Given a shared distribution of data among the cells in a given row and column, the entire array would require 32 channels of data, 64 multipliers, and 48 adders, for only a 4×4 matrix multiplier. As an alternative, consider an array in which the datapaths are pipelined through functional units that are chained together as shift registers. Each FU has the structure shown in Figure P9-21(b), which registers both of its input datapaths and passes the registered values to the adjacent cell in the array. Note that the datapaths between cells are short and that the clock must be distributed to each cell. Compare the throughputs (for a complete set of data), latency, and resources of the implementations. Develop, verify, and synthesize Verilog models of each.
22. Develop and verify a Verilog model of a FIR filter whose architecture exploits symmetry in the tap coefficients.

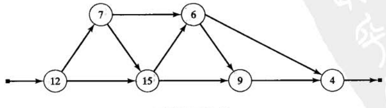


FIGURE P9-19

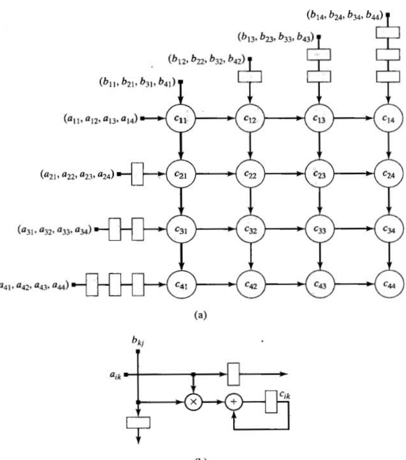


FIGURE P9-21

23. Synthesize *Integrator_Par* (see Example 9.4), and then resynthesize the model after replacing the statement clauses

```
else if (hold) data_out <= data_out;
 else data_out <= data_out + data_in;
```

by the statement clause:

```
else if (!hold) data_out <= data_out + data_in;
```

Compare the two implementations.

24. Develop an ASMD chart for a controller that will control *decimator_3* datapath unit to (1) shift four successive bytes into *Shft_Reg*, (2) transfer the contents of *Shft_Reg* into *Int_Reg*, (3) transfer the contents of *Int_Reg* into *Decim_Reg*, and (4) wait two cycles before repeating the sequence, while *Go* is asserted. Otherwise, the machine returns to and remains in its reset state until *Go* is re-asserted. Note: the port list of *decimator_3* is not to be changed. Embed the datapath and the controller in *decimator_unit_3* and, using the testbench below, verify your model. Synthesize your model and verify that the functionality of the synthesized circuit matches that of your behavioral model.

```
module t_decimator_3_unit # (parameter word_length = 8, latency = 4)( );
  reg [word_length -1: 0]              data_in;
  reg                                  Go;
  reg                                  clock, reset;
  wire [word_length*latency -1: 0]     data_out;

  decimator_3_unit M0 (data_out, data_in, Go, clock, reset);

  initial #1000 $finish;
  initial begin clock = 0; forever #10 clock = ~clock; end
  initial fork
   reset = 1;
   #20 reset = 0;

   #40 data_in = 8'hab;
   #160 data_in = 8'hcd;
   #280 data_in = 8'hef;
   #400 data_in = 8'hab;
   #520 data_in = 8'hcd;
   #640 data_in = 8'hef;

   #50 Go = 1;
   //#160 Go = 0;

  join
endmodule
```

25. The buffer registers in the FIFO implementation in Example 9.10 maintain coherency of the data path by compensating for the inherent latency of the two-stage synchronizer. Consider whether an alternative design can eliminate the buffer registers by modifying the serial-to-parallel converter to anticipate the signal *write* by two clock cycles.
26. Using *t_FIFO_Buffer*, develop additional tests to verify *FIFO_Buffer* for all conditions of the stack and assertions of read and/or write operations.
27. Synthesize *FIFO_Buffer* and verify a simulation match between the behavioral and gate level models.

CHAPTER 10

Architectures for Arithmetic Processors

This chapter presents alternative architectures and algorithms for the arithmetic operations in a digital machine. Many algorithms in digital signal processing require repeated execution of arithmetic operations, so it is important that they be implemented efficiently. How these operations are implemented depends on how numbers are represented in a machine. So we will briefly examine the commonly used schemes for representing positive and negative numbers and fractions. Then we will examine algorithms and architectures for implementing addition, subtraction, multiplication, and division of fixed-point numbers.

10.1 Number Representation

Numbers are represented by a string of characters in a positional notation system having a given radix, or base. A binary number system has two symbols and a radix of 2. An n-bit unsigned binary number is represented in positional notation as $B = b_{n-1}\, b_{n-2} \ldots b_1\, b_0$, with $b_i \in \{0, 1\}$. All digital machines encode numbers in a word of bits. The word has a fixed length, and the interpretation of the pattern of bits depends on the encoding format used by the machine.

The decimal value, B_{10}, of an n-bit unsigned binary number B is formed as a weighted sum of ascending powers of 2, with the most significant bit (MSB) having the greatest weight (2^{n-1}), and the least significant bit (LSB) having the lowest weight (2^0):

$$B_{10} = b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \cdots + b_1 2^1 + b_0 2^0 = \sum_{i=0}^{i=n-1} b_i 2^i$$

An n-bit word can represent 2^n distinct numbers, but the dynamic range of numbers that are realized is dependent on the encoding format. An n-bit unsigned binary format

can represent decimal numbers from 0 to $2^n - 1$. For example, the decimal value of an 8-bit unsigned binary number can range from 0_{10} (0000_0000_2) to 127_{10}(1111_1111_2).

In general, a binary number can be expressed as a weighted sum of ascending and descending powers of 2:

$$B = b_{n-1}b_{n-2}\ldots, b_1 b_0 b_{-1}\, b_{-2}\ldots b_{-m+1} b_{-m}$$

with decimal value

$$B_{10} = b_{n-1}2^{n-1}b_{n-2}2^{n-2} + \cdots + b_1 2^1 + b_0 2^0 + b_{-1}\, 2^{-1} + b_{-2}\, 2^{-2} + \cdots$$
$$+ b_{-m+1}\, 2^{-m+1}\, \mathrm{b}_{-m}\, 2^{-m}$$

and

$$B_{10} = \sum_{i=-m}^{i=n-1} b_i\, 2^i$$

The weights having a negative power of 2 form the fractional part of the number, and the radix point (.) separates the integer part of the number from its fractional part. The radix point for an n-bit integer is located immediately to the right of its LSB; the radix point of an m-bit fraction is located immediately to the left of its MSB. Fixed-point numbers have their radix point in a specific position in a computer word [1]. The radix point is not implemented physically in the machine; the designer must keep track of its location, which may vary as arithmetic operations are performed.

The arithmetic sign of a number in a digital machine must be encoded within the bits of a word. There are three common formats for signed numbers: signed magnitude, 1s complement, and 2s complement. Of these, the 1s complement and 2s complement play a significant role in arithmetic units.

10.1.1 Signed Magnitude Representation of Negative Integers

In signed magnitude representation of positive and negative numbers, the MSB of a word is the encoded sign bit, with 0 representing a positive value, and 1 representing a negative value. The remaining bits of the word represent the magnitude of the number. For example, 0111_2 represents +7, and 1111_2 represents -7_{10}. Eight-bit signed-magnitude numbers can be represented by the number wheel shown in Figure 10-1. The dynamic range of numbers in signed magnitude representation is from $-2^{n-1} - 1$ to $+2^{n-1} - 1$.

If the signs of two signed-magnitude numbers match, addition is executed directly by adding the magnitudes (not the sign bits) and setting the common sign of the result to match the sign of the operands (e.g., $-2_{10} + -3_{10} = 1010_2 + 1011_2 = 1101_2 = -5_{10}$). If the sign bits of the numbers do not match, the signs and relative magnitudes of the words must be used to determine whether to add or subtract the numbers and to determine the sign of the outcome (see Katz [2] for examples). Having two representations for 0 complicates arithmetic operations on signed binary numbers. Hardware units do not directly implement addition and subtraction of signed magnitude numbers.

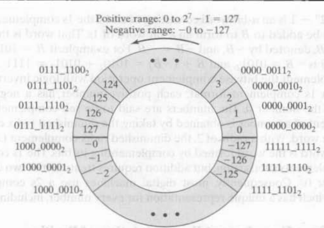


FIGURE 10-1 Number wheel representing 8-bit signed-magnitude numbers.

10.1.2 Ones Complement Representation of Negative Integers

Positive numbers are represented in a 1s complement system in the same way they are represented in a signed-magnitude system, but negative 1s complement numbers are represented differently, in a 1s complement format. The number wheel for 1s complement numbers is shown in Figure 10-2. Note that 0_{10} has two representations: 0000_0000_2 and 1111_1111_2. The dynamic range of numbers in an n-bit 1s complement system is from $-2^{n-1} - 1$ to $+2^{n-1} - 1$, the same as for signed magnitude numbers, but negative numbers are encoded differently in the two formats.

The 1s complement of an n-bit binary number B, denoted by $-B$, is defined by

$$B + (-B) = 2^n - 1$$

FIGURE 10-2 Number wheel representing 8-bit 1s complement numbers.

Because $2^n - 1$ is an n-bit word consisting of all 1s, the 1s complement of B is the word that must be added to B to form an n-bit word of 1s. That word is the bitwise complement[1] of B, denoted by $\sim B$, and $-B = \sim B$. For example, if $B = 1010_2$, the 1s complement of B is $\sim B = 0101_2$, and $B + (\sim B) = 1010_2 + 0101_2 = 1111_2$. Digital machines easily implement the bitwise complement operation with logic inverters.

In a 1s complement format, each positive number has a negative counterpart, including the number 0. The numbers are said to be self-complementing, because the complement of a number is obtained by taking the diminished radix complement of the bits of the word. With a radix of 2, the diminished radix complement (1s complement) of a binary word is the word formed by complementing its bits. The 1s complement format easily implements subtraction, but addition requires treatment of two different representations for 0.[2] Consequently, most digital machines use a 2s complement encoding scheme, which has a unique representation for every number, including 0 [2].

10.1.3 Twos Complement Representation of Positive and Negative Integers

The 2s complement of an n-bit binary integer is defined by $B^* = 2^n - B$, so $B + (B^*) = 2^n = 0$ modulo n. Adding 1 to the 1s complement of a word forms its 2s complement. The range of decimal numbers that can be represented in a 2s complement format is from -2^{n-1} to $+(2^{n-1} - 1)$. Twos complement numbers can be represented by the number wheel shown in Figure 10-3. For $n = 8$, numbers are

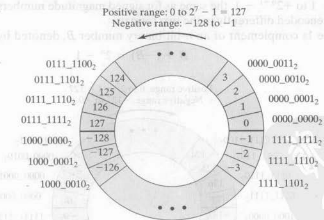


FIGURE 10-3 Number wheel representing 8-bit 2s complement numbers.

[1] The bitwise complement operator (~) is included in the Verilog language to support arithmetic operations.
[2] The signed magnitude format also has two representations for 0.

incremented counterclockwise by 1, from 0 to $2^7 - 1 = 127_{10}$. In the clockwise direction, the count descends from 0 to $-2^7 = -128_{10}$. Figure 10-4(a) demonstrates that the 2s complement of the 2s complement of a number is the number itself. One full rotation around the wheel returns the pointer to the same location as the pointer to 0, and increments the count from $2^{n-1} - 1$ to 2^n.

The 2s complement system is important in logic circuit design because subtraction of 2s complement numbers has a very simple hardware realization, and the arithmetic

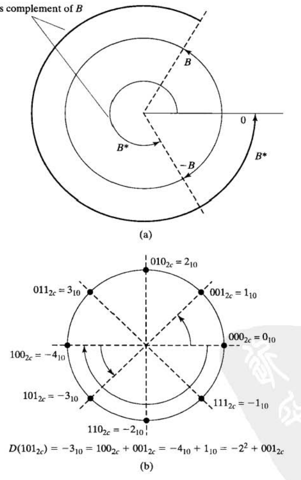


FIGURE 10-4 (a) The 2s complement of the 2s complement of a number B is B. (b) Signed decimal value of (101_{2c}).

operations of addition, subtraction, multiplication, and division of 2s complement numbers can all be performed in hardware with a unit that can carry out binary operations of addition and bitwise complement. Subtraction involves the operations of bitwise complement and addition. Multiplication involves repeated addition, and division involves repeated subtraction. Arithmetic operations on 2s complement numbers use the same hardware for addition and subtraction but have reduced dynamic range compared to an unsigned binary format.

The signed decimal value of a number in 2s complement format can be obtained directly, as

$$D(B) = -b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \cdots + b_1 2^1 + b_0 2^0.$$

Figure 10-4(b) illustrates use of this expression to find $D(101_{2c})$, where the subscript $2c$ denotes a binary word in 2s complement format.

10.1.4 Representation of Fractions

The 2s complement of a fraction is defined by

$$B^* = 2 - B \text{ and } B + B^* = 2.$$

The 2s complement of a fraction can found by starting at the LSB and complementing all the digits to the left of the least significant 1 in the word, which is the same procedure that is used to form the 2s complement of an integer. The 2s complement fraction $1.00000\ldots$ is a special case. It actually represents the number -1, since the sign bit is negative, and the 2s complement of $1.0000\ldots$ is $2 - 1 = 1$. The integer $+1$ cannot be represented in the 2s complement fraction system, since $0.111\ldots$ is the largest positive fraction.

10.2 Functional Units for Addition and Subtraction

Addition and subtraction are implemented in all arithmetic processors. There are several alternatives that provide a trade-off between hardware cost and performance.

10.2.1 Ripple-Carry Adder

The ripple-carry adder presented in Chapter 4 is limited by the time required to propagate a signal transition from the carry-in bit to the carry-out bit of the unit. If the word length of the processor using the adder is large, it might be necessary to use an alternative architecture to form the outputs quickly enough to satisfy timing constraints. Pipelining the dataflow is one alternative (see Chapter 9), but it introduces latency and requires hardware to implement the pipeline registers. Another alternative is to consider other algorithms for addition. Among those that are used are the carry look-ahead algorithm, the carry select algorithm, and the carry-save algorithm [3]. Of these, we will consider the look-ahead algorithm.

10.2.2 Carry Look-Ahead Adder

The algorithm for a carry look-ahead adder is based on the observation that the value of the carry into any stage of a multicell adder depends on only the data bits of the previous stages and the carry into the first stage. This relationship can be exploited to improve the speed of the adder by using additional logic to implement the carry, rather than waiting for the value to propagate through the cells of the adder.

A given cell is said to generate a carry if both of the cell's data bits are 1. A cell is said to propagate a carry if either of the cell's data bits could combine with the carry into the cell to cause a carry out to the next stage of the adder. Let a_i and b_i be the data bits at the ith cell of the adder, let c_i be the carry into the ith cell, let s_i be the sum bit *out* of the ith cell, and let c_{i+1} be the carry out of the cell. We define *generate* and *propagate* bits g_i and p_i using the bitwise-and operator (&)[3] and the exclusive-or operator (^) as follows:

$$g_i = a_i \,\&\, b_i$$
$$p_i = a_i \,^\wedge\, b_I$$

The Venn diagram in Figure 10-5 shows where p_i and g_i are asserted, depending on a_i and b_i. Note that p_i and g_i are mutually exclusive.

The logical expressions forming the sum and carry bits at each stage of the adder can be written in terms of the Verilog bitwise operators as follows:

$$s_i = (a_i \,^\wedge\, b_i) \,^\wedge\, c_i = p_i \,^\wedge\, c_i$$
$$c_{i+1} = ((a_i \,^\wedge\, b_i) \,\&\, c_i) | (a_i \,\&\, b_i) = (p_i \,\&\, c_i) | g_i$$

Note that because p_i and g_i are mutually exclusive the algorithm can also be expressed in arithmetic terms as

$$s_i = (a_i \,^\wedge\, b_i) \,^\wedge\, c_i = p_i \,^\wedge\, c_i$$
$$c_{i+1} = (a_i \,^\wedge\, b_i) \,\&\, c_i + a_i \,\&\, b_i = p_i \,\&\, c_i + g_i$$

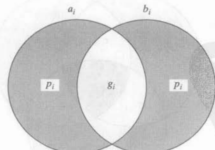


FIGURE 10-5 Venn diagrams for assertion of p_i (propagate) and g_i (generate) for an adder cell.

[3]& and && have the same effect on scalar quantities.

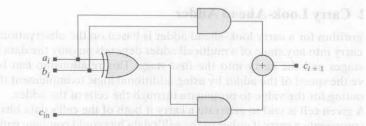


FIGURE 10-6 Schematic for arithmetic implementation of carry bit.

The carry bit can be formed using either the bitwise OR operator or the arithmetic sum (modulo 2 addition), with identical results. The schematic for the subcircuit implementing the algorithmic version of the carry is shown in Figure 10-6. Obviously, the simpler realization is to replace the adder by an OR gate.

The carry out of the ith cell is formed by adding (OR-ing) the bit propagated by the cell with the bit generated by the cell. Only one of the terms will be 1, because p_i and g_i are mutually exclusive. This second form of the equation for c_{i+1} produces the same result as the first one.

The dependencies of s_i and c_{i+1} on a_i, b_i, and c_i are depicted in the Venn diagram of Figure 10-7, where the three circular regions represent the data inputs to a cell of the adder, a_i, b_i, and c_i; each of the subregions denote where the data outputs s_i and c_{i+1} are asserted. The presence of a variable's label within a subregion of the diagram indicates that the variable is asserted for the combination of the data inputs associated with the subregion. For example, the sum bit is asserted in four subregions of the diagram where $a_i = 1, b_i = 0$, and $c_i = 0$; where $a_i = 0, b_i = 1$, and $c_i = 0$; where $a_i = 1, b_i = 1$, and $c_i = 1$; or where $a_i = 0, b_i = 0$ and $c_i = 1$.

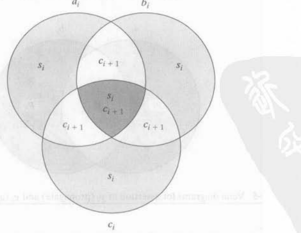


FIGURE 10-7 Data input–output relationships of an adder cell.

In terms of Verilog bitwise logic (&) and arithmetic (+) operators, the cells of the adder have[4]

$$s_0 = p_0 \,^\wedge\, c_0$$
$$c_1 = (p_0 \,\&\, c_0) + g_0$$

$$s_1 = p_1 \,^\wedge\, c_1 = p_1 \,^\wedge\, [(p_0 \,\&\, c_0) + g_0] = p_1 \,^\wedge\, (p_0 \,\&\, c_0) + p_1 \,^\wedge\, g_0$$
$$c_2 = (p_1 \,\&\, c_1) + g_1 = p_1 \,\&\, p_0 \,\&\, c_0 + p_1 \,\&\, g_0 + g_1$$

$$s_2 = p_2 \,^\wedge\, c_2 = p_2 \,^\wedge\, [p_1 \,\&\, p_0 \,\&\, c_0 + p_1 \,\&\, g_0 + g_1]$$
$$c_3 = p_2 \,\&\, c_2 + g_2 = p_2 \,\&\, [p_1 \,\&\, p_0 \,\&\, c_0 + p_1 \,\&\, g_0 + g_1] + g_2$$
$$= p_2 \,\&\, p_1 \,\&\, p_0 \,\&\, c_0 + g_2 = p_2 \,\&\, p_1 \,\&\, g_0 \,\&\, c_0 + p_2 \,\&\, g_1 + g_2$$

$$s_3 = p_3 \,^\wedge\, c_3 = p_3 \,^\wedge\, [p_2 \,\&\, p_1 \,\&\, p_0 \,\&\, c_0 + p_2 \,\&\, p_1 \,\&\, g_0 + p_2 \,\&\, g_1 + g_2]$$
$$c_4 = p_3 \,\&\, c_3 + g_3 = p_3 \,\&\, [p_2 \,\&\, p_1 \,\&\, p_0 \,\&\, c_0 + p_2 \,\&\, p_1 \,\&\, g_0 + p_2 \,\&\, g_1 + g_2] + g_3$$

These expressions expose the fact that the sum and the carry-out bits of each cell can be expressed in terms of the data bits of that cell and of the previous cells, and the carry into only the first cell of the adder chain (i.e., the logic can be effectively flattened). All of these data are available simultaneously, so there is no need to wait for a carry bit to propagate through the adder to a particular cell. This allows the adder to operate faster, but the cost of this improvement is the extra logic needed to compute the sum and carry-out of each stage. The gate-level implementation of the look-ahead adder requires considerably more silicon area than the ripple-carry adder implemented in the same technology and requires more testing for process-induced faults. (Look-ahead carry is usually implemented on a bit slice of a word.)

Another important observation is that the sum and carry bits at each cell can be computed recursively. To expose this, we write

$$s_0 = p_0 \,^\wedge\, c_0$$
$$c_1 = p_0 \,\&\, c_0 + g_0$$
$$s_1 = p_1 \,^\wedge\, c_1$$
$$c_2 = p_1 \,\&\, c_1 + g_1$$
$$s_2 = p_2 \,^\wedge\, c_2$$
$$c_3 = p_2 \,\&\, c_2 + g_2$$
$$\ldots$$

The algorithm to implement the addition of words will take as many steps as there are cells in the adder. The computation at each step of the recursion depends on the data bits of the corresponding cell and on the carry that was calculated at the immediately previous step. If the propagate bits of an n-bit adder are arranged in a vector $p = (p_{n-1}, \ldots, p_2, p_1, p_0)$, the results of the recursion can be used to form an $n + 1$

[4]The operators ^ and & have higher precedence than +.

dimensional vector: $(c_n, \ldots, c_2, c_1, c_0)$ such that the output word and the *c_out* bit are obtained as

$$sum = p \wedge (c_{n-1}, \cdots, c_2, c_1, c_0).$$
$$c_out = c_n$$

The gate-level circuit implementing the 4-bit carry look-ahead logic is shown in Figure 10-8(a), and, for comparison, the schematic of the circuit implementing the

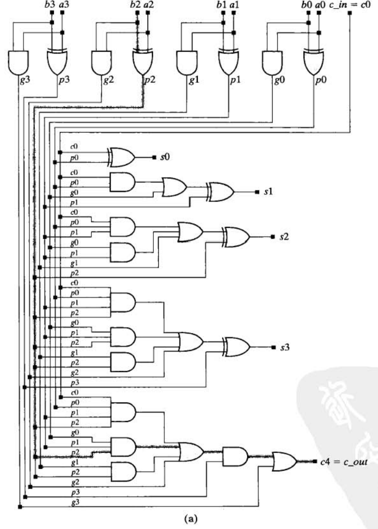
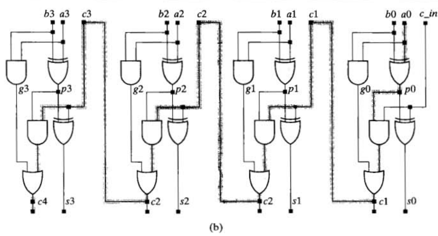


FIGURE 10-8 Gate-level circuits of 4-bit-slice adders: (a) carry look-ahead adder and (b) ripple-carry adder. The longest paths are highlighted.



(b)

FIGURE 10-8 Continued

recursive algorithm is shown in Figure 10-8(b). The longest path is highlighted in both circuits. Note that the recursive algorithm implements the same circuit as a 4-bit ripple carry adder.

A Verilog description of the propagate and generate recursive algorithm [13] for a four-bit adder is given below.

```
module Add_prop_gen (output [3: 0] sum, output c_out, input [3:0] a, b, input c_in);
  reg          [3: 0]    carrychain;
  integer                i;
  wire         [3: 0]    g = a & b; // carry generate, continuous assignment,
                                    bitwise and
  wire         [3: 0]    p = a ^ b; // carry propagate, continuous assignment,
                                    bitwise xor
  wire [4:0] shiftedcarry = {carrychain, c_in} ;          // concatenation
  assign [3:0] sum = p ^ shiftedcarry;                    // summation
  assign c_out = shiftedcarry[4];                         // carry out, usage:
                                                            bit select
  always @ (a, b, c_in, p, g)                             // event "or"
   begin: carry_generation                                // usage: block
                                                            name
   integer i;                                             // local variable
   carrychain[0] = g[0] + (p[0] & c_in);                  // needed for
                                                            simulation
   for(i = 1; i <= 3; i = i + 1)
    begin
     carrychain[i] = g[i] | (p[i] & carrychain[i-1]);
    end
  end
endmodule
```

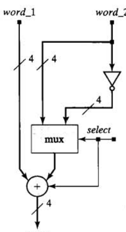


FIGURE 10-9 Architecture for a combined 4-bit datapath adder and subtractor unit.

Hardware units usually implement subtraction by adding the 1s complement of the subtrahend to the minuend, and then adding 1 to the result. This can be implemented with the architecture shown in Figure 10-9. One adder unit can be used for addition or subtraction, depending on the value of the signal *select*.

10.2.3 Overflow and Underflow

Overflow occurs under two conditions: (1) when adding two positive numbers produces a sum that exceeds the largest positive number that can be represented in the word length of the unit (i.e. the result is negative) and (2) when adding two negative numbers produces a sum that is positive (i.e. the sum exceeds the smallest negative number that can be represented in the word length of the machine). Arithmetic units include logic for underflow and overflow detection.

10.3 Functional Units for Multiplication

Multipliers are important functional elements of arithmetic units, digital signal processors, and other circuits that execute arithmetic operations. Multiplication can be implemented with a combinational circuit or by a sequential circuit. A combinational circuit that multiplies two numbers will require more silicon area, but will operate faster than a sequential multiplier. Sequential multipliers are attractive because they require less area, but a complete multiplication takes several clock cycles to form the product. We will investigate various designs of multipliers for signed and unsigned numbers, beginning with binary multipliers, which form the product of a pair of binary words (i.e., unsigned numbers). We will also consider circuits for multiplying fractions. Our approach will be to first present a basic architecture for a multiplier and then present alternative architectures with additional features that enhance the design.

10.3.1 Combinational (Parallel) Binary Multiplier

Consider two binary (unsigned) numbers with the following representation:

$$A = (A_{m-1}, A_{m-2}, \ldots, A_1, A_0)_2 = \sum_{i=0}^{m-1} A_i 2^i$$

$$B = (B_{n-1}, B_{n-2}, \ldots, B_1, B_0)_2 = \sum_{j=0}^{n-1} B_j 2^j$$

Their product can be written as

$$A \times B = \sum_{i=0}^{m-1} A_i 2^i \sum_{j=0}^{n-1} B_j 2^j$$

$$A \times B = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} A_i B_j 2^{i+j}$$

The product has $m \times n$ terms, and their summation may produce one more term, so the final result can be written as a sum of weighted powers of 2:

$$A \times B = \sum_{k=0}^{m+n-1} P_k 2^k$$

where $P_0 = A_0B_0$, $P_1 = A_1B_0 + A_0B_1$, etc. The term $P_{m+n-1} 2^{m+n-1}$ accounts for a possible carry.

Figure 10-10 illustrates the basic process for forming the product of two 8-bit binary words for the product $215_{10} \times 23_{10} = 4945_{10}$. Shifted copies of the multiplicand are successively aligned with the locations of the bits of the multiplier word and then the columns are added. If a multiplier bit is 0, the corresponding copy of the multiplicand is skipped, and the location that will be occupied by the next copy is shifted by one position toward the MSB of the multiplier. For example, a double shift is shown in Figure 10-10 at the shaded location where a multiplier bit is 0.

The process shown in Figure 10-10 works manually, and a combinational circuit can be developed to implement the product, with binary multiplication formed by an

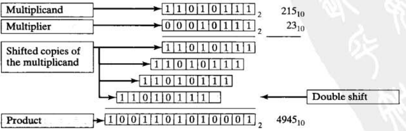


FIGURE 10-10 Formation of a product by columnwise adding shifted copies of the multiplicand.

AND gate. Note, however, that the process uses columnwise addition of the terms that form the partial products, and would require a hardware scheme that uses multiple adders for each column [2]. An ordinary adder operates on only two words at a time, so a more attractive scheme that forms a sequence of row sums by adding a single copy of the multiplicand to an accumulated product will be presented here.

Figure 10-11 shows how the multiplication evolves. First, a pair of appropriately shifted copies of the multiplicand are added to form a sum, then another shifted copy of the multiplicand is added, and so forth, to accumulate the sums that ultimately form the product. This scheme is attractive, because it adds only two words at a time, has a direct hardware counterpart, and can be described by a sequential behavior in a hardware description language (HDL) model.

The structure of a combinational circuit that multiplies two unsigned binary numbers (words) can be derived from the manual operations for multiplying the numbers in a radix 2 system. For simplicity, we illustrate the process for two 4-bit words, A and B (the multiplicand and multiplier, respectively), and we form their product, $A \times B$, as shown in Figure 10-12. Beginning with the LSB of the multiplier, each bit is multiplied by the bits of the multiplicand to form a so-called partial product. In a radix 2 system, the operation of multiplication that forms a partial product is equivalent to AND-ing the multiplier bit with each bit of the multiplicand. Each partial product is shifted toward the MSB to the position of the corresponding multiplier bit. Then the partial products are added. As we noted earlier, a manual method would add the terms in the aligned columns of the partial products, which, in general, requires addition of several terms, but hardware adders are designed to add only two words. Consequently, the result of adding the rows of the partial products is accumulated, with attention to any carries that are generated by the addition of the terms in a given column. The structure will require full adders when carries are involved and half adders otherwise. In general, the resulting final product may contain as many as $2 * L_word$ significant bits, where *L_word* is the length of the multiplier and multiplicand words.

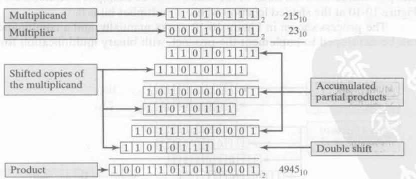


FIGURE 10-11 Alternative sequential process forming the product of a pair of binary words by accumulating partial sums.

				$A3$ $B3$	$A2$ $B2$	$A1$ $B1$	$A0$ $B0$	Multiplicand Multiplier
			A_3B_1 C_{12}	A_3B_0 A_2B_1 C_{11}	A_2B_0 A_1B_1 C_{10}	A_1B_0 A_0B_1	A_0B_0 S_{00}	Partial product 0 Partial product 1 1st row carries
		C_{13} A_3B_2 C_{22}	S_{13} A_2B_2 C_{21}	S_{12} A_1B_2 C_{20}	S_{11} A_0B_2	S_{10}		1st row sums Partial product 2 2nd row carries
	C_{23} A_3B_3 C_{32}	S_{23} A_2B_3 C_{31}	S_{22} A_1B_3 C_{30}	S_{21} A_0B_3	S_{20}			2nd row rums Partial product 3 3rd row carries
C_{33}	S_{33}	S_{32}	S_{31}	S_{30}				3rd row sums
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	Final product
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	Weight

FIGURE 10-12 Steps in the multiplication of unsigned 4-bit binary words.

The combinational logic structure consisting of AND-gates, full adders, and half-adders shown in Figure 10-13 implements a parallel multiplier for a 4-bit wide datapath. This method of multiplication is called *partial product accumulation* [2], because rows of linked adders generate the accumulated partial products that would evolve from a manual multiplication of the data words, as we showed in Figure 10-11. Most of the array is formed of linked copies of the basic cell shown in Figure 10-13. An alternative structure, composed entirely of basic cells, is shown in Figure 10-14, implementing half-adders as full adders with their carry-in line hard wired to 0. The resulting regular array of identical objects is called a systolic array (i.e., copies of a basic cell) and is ideally suited for fabrication as an integrated circuit [5]. In practice, the boundary cells can be replaced by their counterparts from Figure 10-13. A 4-bit combinational multiplier has 16 AND gates, 8 full adders, and 4 half adders. An 8-bit multiplier would extend the array structure to accommodate 8-bit input datapaths, producing a 16-bit output datapath for the product. The systolic array in Figure 10-14 is attractive because it has a regular structure of identical cells and easily accommodates expansion to longer word length by direct abutment of cells, which leads to an area-efficient physical layout of the cell on a die, with short interconnect paths between cells. Because the structure is isomorphic to a dataflow graph, it can be used to identify cutsets for pipelining the structure to obtain greater throughput (see Problem 40 at the end of this chapter).

In synchronous operation, the clock cycle that governs the presentation of data to the multiplier must accommodate the longest path through the circuit, which is the path from the LSBs, through the adders, to the MSB of the product (see the shaded path in Figure 10-13). Both the carry and sum paths of the adders affect the longest

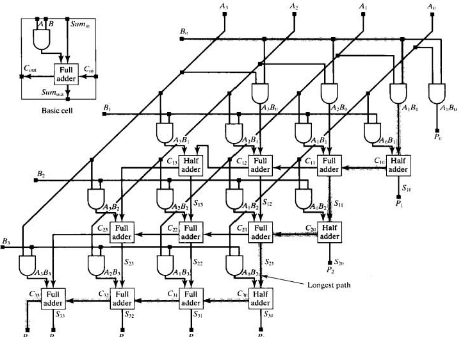


FIGURE 10-13 An array structure of glue logic, half adders, and full adders for a 4-bit binary multiplier.

path, and balanced delays through them are desirable [1]. The area of the device is relatively large, compared to other realizations, such as the sequential multipliers that will be considered below, but the extra area is the price of the superior performance of the combinational multiplier.

10.3.2 Sequential Binary Multiplier

Combinational (array) multipliers operate fast, but require a significant amount of silicon area. If area is an important consideration, it can be reduced at the expense of performance by scheduling the suboperations of the multiplier to execute in successive clock cycles. Sequential multipliers are compact, require fewer adders, and are amenable to pipelining. The area required by combinational multipliers grows geometrically with the word length, but we will see that the area of a sequential multiplier does not grow significantly with word length and that the number of clock cycles required to complete a multiplication also grows in a linear manner, rather than exponentially, with the word length. The behavioral description of a sequential multiplier is parameterizable, which makes the model portable, amenable to synthesis, and re-usable.

The sequence of operations forming the product of binary numbers by adding shifted copies of the multiplicand to an accumulated product were implemented by a combinational circuit in Figure 10-13. The schematic of the circuit suggests how to form

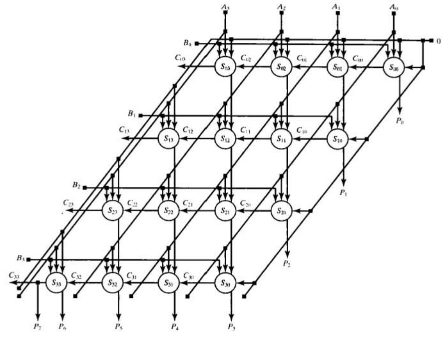


FIGURE 10-14 A systolic array structure for a 4-bit binary multiplier.

a sequential behavioral model of the multiplier, one that eliminates the spatial distribution of adders in exchange for a temporal distribution of computation that uses storage registers and a single adder.

The interface signals and the block diagram symbol of a 4-bit sequential binary multiplier are shown in Figure 10-15, where [−: 0] denotes a parameterizable bit range

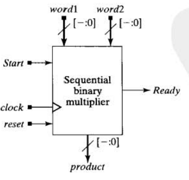


FIGURE 10-15 Interface signals and block diagram symbol for a sequential binary multiplier.

for a datapath (e.g., $[L_word - 1:0]$ for *word1* and *word2*, and $[2 * L_word - 1:0]$ for *product*). The datapaths for *word1, word2*, and *product* hold the multiplicand, multiplier, and product words, respectively. The signal *Ready* asserts when the unit is ready to execute a multiplication sequence, and when a valid product has been formed at the end of an execution sequence. *Ready* asserts until *Start* initiates a multiplication sequence, and re-asserts when a valid product is formed. We will now consider architectural alternatives and methodologies for designing a sequential binary multiplier of unsigned words.

10.3.3 Sequential Multiplier Design: Hierarchical Decomposition

The method for designing a sequential binary multiplier has two main steps: (1) choose a datapath architecture and (2) design a state machine to control the datapath. For a given datapath architecture, the state machine must generate the appropriate sequence of control signals to direct the movement of data and execute the operations that produce the desired product.

The first datapath architecture we will consider is shown in Figure 10-16 for an 8-bit datapath. Along with a single adder, shift registers are allocated for the multiplicand and multiplier, and a fixed register is allocated for the product. The multiplicand register is sized to twice the size of the multiplicand data word to accommodate shifting. The length of the *Product* register is the same as the size of the multiplicand register—the range of values formed by the product of two 8-bit numbers does not require an extra bit for a possible carry. The controller must assert *Ready* and then wait for an external agent to signal *Start*. When *Start* is asserted, the controller must de-assert *Ready*, load the registers, direct the shifting and adding of data to form the product, and finally re-assert the signal *Ready*. The controller uses the bits of the multiplier register to determine whether to execute the shifting and adding operations to form the accumulated product.

The architecture in Figure 10-16 uses only one adder and separate registers to hold *multiplier, multiplicand*, and *product*. In contrast, the combinational multiplier in Figure 10-13 does not use storage registers, but it requires that the data words be held externally until *product* is formed. This has implications for external storage or bus utilization. In contrast, the sequential adder can release its external datapaths as soon its

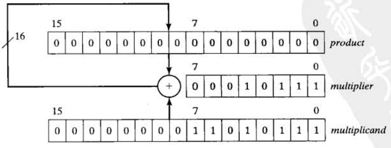


FIGURE 10-16 Datapath architecture of a sequential 8-bit binary multiplier.

registers are loaded. Note that long words do not require more area for logic, other than that needed to properly size the registers and the adder.

The controller of the sequential multiplier architecture in Figure 10-16 loads the multiplicand and multiplier words into their registers, then shifts *multiplicand* to the left, relative to the *product* register. At each step, the *multiplier* register is also shifted, but to the right, and the value of *m0*, *multiplier[0]*, determines whether *multiplicand* is added to *product*. The synchronous movement of data is depicted in Figure 10-17 for the product $215_{10} \times 23_{10} = 4945_{10}$, where the addition operation is executed as parallel addition of the entire words. A single adder supports all of the cycles of addition.

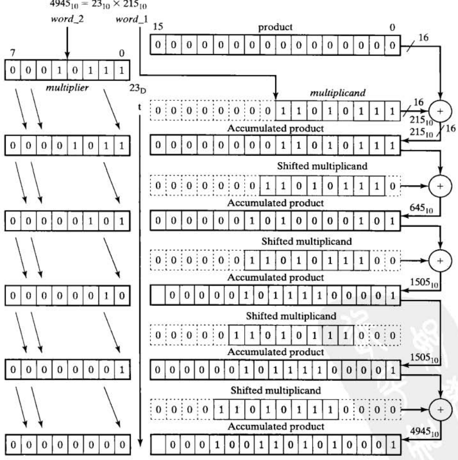


FIGURE 10-17 Register transfers in an 8-bit sequential binary multiplier.

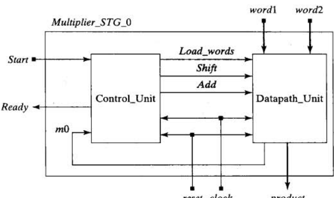


FIGURE 10-18 Structural units of a partitioned sequential binary multiplier.

Partitioning the design into a datapath and controller gives the Verilog structural decomposition shown in Figure 10-18, where *Control_UnitContorl_Init* and *Datapath_Unit* are separately encapsulated Verilog modules instantiated within a top-level module, *Multiplier_STG_0*. In this structure, the status signal *m0* is the LSB of *multiplier*, and is passed to the controller and used to control its state transitions. A fully structural approach to the design would be to write and link structural descriptions of shift registers, adders, and ordinary registers, and then develop (e.g., from timing charts and Boolean expressions) a state machine to control the structure. In contrast, our approach will start with the top-level structural partition and then write behavioral descriptions of the functional units, with reliance on a synthesis tool to determine their actual physical structure. This approach can maximize a designer's productivity, by leveraging the Verilog language and readily available synthesis tools. We will use it to examine alternative datapath architectures, controllers, and design methods and to explore tradeoffs. We will also expose some of the subtle, but serious, traps that await the unwary designer.

10.3.4 STG-Based Controller Design

The first design method will use state-transition graphs (STGs) to specify the state transitions of the controller. Figure 10-19 shows two versions of a STG for the controller, with different behavior in *S_8*. State transitions occur at the active edge of a clock, and are governed by the conditions annotated on the arcs of the graph (i.e., the machine will remain in a given state until the condition is satisfied). An arc without annotation denotes an unconditional transition; – denotes a don't-care condition. Signals that are not explicitly asserted are de-asserted. Under the action of *reset*, the machines enter *S_idle* from any state and remain there, with *Ready* asserted, until *Start* is asserted with *reset* de-asserted. (An alternative design could assert *Ready* as a Mealy output only while *reset* is de-asserted in *S_idle*.) Thereafter, the state transitions depend on *m0*, the LSB of the

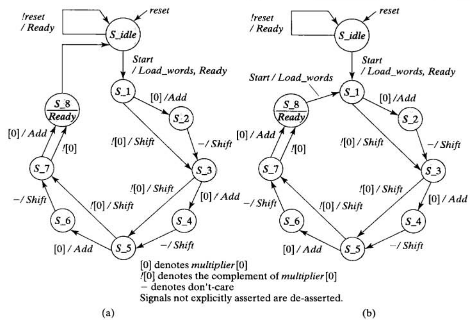


FIGURE 10-19 Alternative STGs for a 4-bit sequential binary multiplier: (a) machine returns to *S_idle* upon completion of multiplication, and (b) machine resides in *S_8* after completion of multiplication.

shifted multiplier. If the LSB is a 1, the signal *Add* is asserted and a transition is made to a state from which *Shift* will be asserted at the next active edge of *clock*. If the LSB of the multiplier is 0, *Shift* is asserted. When *S_8* is entered, *Ready* is asserted as a Moore-type output. At the next active edge of the clock, the machine in Figure 10-19(a) transitions to *S_idle* to await an assertion of *Start*. The version in Figure 10-19(b) remains in *S_8*, with *Ready* asserted, until *Start* is asserted. Then a transition is made to *S_1*, rather than to *S_idle*. Note that once either machine has entered *S_1* it ignores activity on the input datapath until the multiplication is complete.

The Verilog description of *Control_Unit*, given in Example 10.1, is based on the STG in Figure 10-19(b). The machine in Figure 10-19(b) is preferred to the machine in Figure 10-19(a). The former can reach a result in one less state after *Ready* is asserted in continuous operation, because the state does not pass through *S_idle*. In either machine, a complete multiplication can require passage through as few as five states and as many as eight states after *Start* is asserted. Note that the size of the STGs in Figure 10-19 expands linearly with the size of the datapath. This has implications for the utility of an STG-based method in which the sequence of states depends on the size of the datapath, because the model would have to be edited significantly to accommodate a different word size.

This example provides an opportunity to discuss the benefit of using status signals to communicate between the control unit and the datapath unit. An alternative to having the status signal *m0* is to pass the entire multiplier word to the control unit. This would require that the sensitivity list for the level-sensitive behavior modeling the combinational logic of the control unit contain the multiplier word. Consequently, the cyclic behavior would be triggered by all changes to the multiplier word, not just by a change to the LSB. This wastes simulation cycles, and in hardware would result in a higher level of electrical activity distributed over the circuit. Both consequences are undesirable.

The Verilog behavioral description of *Multiplier_STG_0* is given in Example 10.1, along with a testbench that contains an exhaustive pattern generator for *word1* and *word2*, and a comparator that samples *product* and checks, while *Done* is asserted, whether *product* matches the expected value.

Example 10.1

```
module Multiplier_STG_0 #(parameter L_word = 4)(
  output      [2*L_word -1: 0]          product,
  output                                Ready,
  input       [L_word -1: 0]            word1, word2,
  input                                 Start, clock, reset
);
  wire                                  m0, Load_words, Shift;
  Control_Unit M0_Controller (Load_words, Shift, Add, Ready, m0, Start, clock, reset);

  Datapath_Unit M1_Datapath (product, m0, word1, word2, Load_words, Shift, Add,
    clock, reset);

endmodule

module Control_Unit #(parameter L_word = 4, L_state = 4)(          // Datapath
                                                                     size, state
                                                                     size

  output reg                            Load_words, Shift, Add,
  output                                Ready,
  input                                 m0, Start, clock, reset
);
  reg         [L_state -1: 0]           state, next_state;
  parameter                             S_idle = 0, S_1 = 1, S_2 = 2,
                                        S_3 = 3, S_4 = 4, S_5 = 5, S_6 = 6,
                                        S_7 = 7, S_8 = 8;
  assign                                Ready = ((state == S_idle) && !reset)
                                          || (state == S_8);
  always @ (posedge clock, posedge reset)          // State transitions
   if (reset == 1'b1) state <= S_idle; else state <= next_state;

  always @ (state, Start, m0) begin                // Next state and control logic
    Load_words = 0; Shift = 0; Add = 0;            // Default values
    case (state)
```

```
      S_idle:    if (Start) begin Load_words = 1; next_state = S_1; end
                 else next_state = S_idle;
      S_1:       if (m0)        begin Add = 1; next_state = S_2; end
                 else           begin Shift = 1; next_state = S_3; end
      S_2:                      begin Shift = 1; next_state = S_3; end
      S_3:       if (m0)        begin Add = 1; next_state = S_4; end
                 else           begin Shift = 1; next_state = S_5; end
      S_4:                      begin Shift = 1; next_state = S_5; end
      S_5:       if (m0)        begin Add = 1; next_state = S_6; end
                 else           begin Shift = 1; next_state = S_7; end
      S_6:                      begin Shift = 1; next_state = S_7; end
      S_7:       if (m0)        begin Add = 1; next_state = S_8; end
                 else           begin next_state = S_8; end
      S_8:       if (Start) begin Load_words = 1; next_state = S_1; end
                 else next_state = S_8;
      default:   next_state = S_idle;
    endcase
  end
endmodule

module Datapath_Unit (L_word = 4)(
  output reg    [2*L_word -1: 0]      product,
  output                              m0,
  input         [L_word -1: 0]        word1, word2;
  input                               Load_words, Shift, Add, clock, reset
);
  reg           [2*L_word -1: 0]      multiplicand;
  reg           [L_word -1: 0]        multiplier;
  assign                              m0 = multiplier[0];
  // Register/Datapath Operations
  always @ (posedge clock, posedge reset) begin
    if (reset == 1'b1) begin multiplier <= 0; multiplicand <= 0; product <= 0; end
    else if (Load_words == 1'b1) begin
      multiplicand <= word1;
      multiplier <= word2; product <= 0;
    end
  else if (Shift == 1'b1e== 1'b1) product <= product + multiplicand;
 end
endmodule

module test_Multiplier_STG_0 ();
  parameter                      L_word = 4;
  wire        [2*L_word -1: 0]   product;
  wire                           Ready;
  integer                        word1, word2;     // multiplicand, multiplier
  reg                            Start, clock, reset;
  Multiplier_STG_0 M1_Multiplier (product, Ready, word1, word2, Start, clock, reset);
  // Exhaustive Testbench
  reg         [2*L_word -1: 0]   expected_value;
  reg                            code_error;
```

```
initial #80000 finish;              // Timeout
always @ (posedge clock) // Compare product with expected value
  if (Start) begin
    #5 expected_value = 0;
    expected_value = word2 * word1;
    // expected_value = word2 * word1 + 1; // Use to check error detection
    code_error = 0;
  end
  else begin
    code_error = (M1.M2.state == M1.M2.S_8) ? |(expected_value ^ product) : 0;
end
initial begin clock = 0; forever #10 clock = ~clock; end
initial begin
   #2 reset = 1;
   #15 reset = 0;
end
initial begin #5 Start = 1; #10 Start = 15; end          // Test for reset override
initial begin   // Exhaustive patterns
   for (word1 = 0; word1 <= 15; word1 = word1 +1) begin
   for (word2 = 0; word2 <= 15; word2 = word2 +1) begin
     Start = 0; #40 Start = 1;
     #20 Start = 0;
     #200;
   end // word2
   #140;
   end //word1
 end // initial
endmodule
```

End of Example 10.1

The controller in *Multiplier_STG_0* consists of two behaviors, one is synchronous (edge-sensitive) and the other is combinational (level-sensitive). The synchronous behavior models the state transitions at the active edges of the clock, subject to asynchronous reset action. The reset action, being part of the functionality of a flip-flop, is described in the edge-sensitive behavior of the controller—not in the level-sensitive behavior describing combinational logic. The combinational logic (level-sensitive behavior) forms the next state and the output signals that control the datapath. The Verilog description of the outputs of the controller can be written directly from the annotation on the branches of the STG. *Each output variable is initialized to a de-asserted condition at the beginning of the behavior to establish default values.* This implements an "assignment by exception" scheme that reduces the amount of code that must be written and assures that every output is assigned a value whenever the behavior is triggered, which reduces the possibility of accidentally creating latches. Only values that are to be asserted are described in the ***case*** statement branch, for a given state. Note that the level-sensitive behavior uses the blocking procedural assignment operator (=) and its sensitivity list includes all of the variables that are read within the behavior. Using a

blocking procedural assignment operator in the combinational logic behavior, and using a nonblocking assignment operator (<=) to make all register transfers in an edge-sensitive behavior, will avoid race-induced discrepancies between the behavioral and synthesized models of a sequential machine that has been partitioned into a datapath and a controller; the completeness of the level-sensitive event-control expression of the combinational behavior prevents unwanted latches from being synthesized. Also note that the decoding of *S_8* includes an explicit assignment to *next_state* if *Start* is 0, to prevent synthesis of a latch.

The datapath unit is modeled by an edge-sensitive synchronous behavior that handles all of the register operations, under the direction of the signals generated by the controller. The behaviors of the shift registers are compactly described by the shift operator of the Verilog language. The behavior uses the nonblocking procedural assignment operator—the register operations are concurrent, with the values held by the registers after the clock edge being determined by the values held at their inputs immediately before the clock edge.

The testbench, *test_Multiplier_STG_0* combines an exhaustive pattern generator with a self-checker to detect whether the model is correct. Figure 10-20 shows simulation results for all possible patterns of *word1* and *word2*. The displayed resolution obscures the actual data, but the signal *code_error* indicates that the model is correct (provided that the testbench itself is indeed correct). Figure 10-21 shows simulation results for a representative case ($4_{10} \times 8_{10} = 32_{10}$), with a detailed view of state transitions and control signal assertions leading to the formed product. (*Note: word1*, *word2*, *state*, *expected_value*, and *product* are displayed with a decimal radix; *multiplicand* and

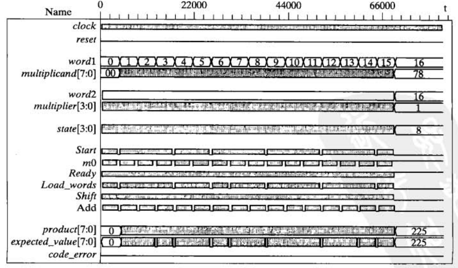


FIGURE 10-20 Results of exhaustive simulation of *Multiplier_STG_0* with a self-checking testbench.

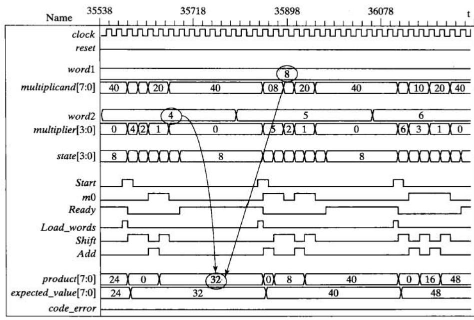


FIGURE 10-21 Sample of a multiplication sequence of *Multiplier_STG_0*.

multiplier are displayed in hexadecimal format.) The exhaustive test results demonstrate that the multiplication operation is correct for the patterns applied. In general, it is also necessary to verify that the machine operates correctly under the action of the asynchronous inputs. For example, a more robust testing scheme would verify that the behavior is correct if *Start, reset, word1*, or *word2* changes randomly. The ability of the testbench to detect an error correctly was also verified, but the results are not displayed.

10.3.5 Efficient STG-Based Sequential Binary Multiplier

The sequential multiplier, *Multiplier_STG_0*, presented in Example 10.1 is inefficient because it executes the add and shift operations in separate clock cycles. If the architecture is modified to direct the output of the adder to the appropriate bits of the product register, the operations can execute in the same cycle.

The modified controller described by the STG in Figure 10-22 can shorten the execution sequence by almost a factor of 2. The signal *Add* is replaced by the signal *Add_shift* to indicate the combined activity. The datapath modification can be achieved in hardware by moving wires by one bit position, but we will include this change in our behavioral model and leave the actual details of the wiring to a synthesis tool. Also, this new design has a more intelligent controller—it includes logic to abort the multiplication sequence if either data word presented to the multiplier is 0, in which case there is

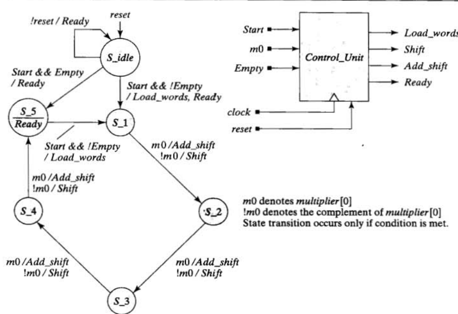


FIGURE 10-22 STG and block diagram symbol for the controller of an efficient sequential multiplier.

no need to multiply *word1* by *word2*. The datapath unit is modified to produce a status signal, *Empty*, to indicate the condition that an input datapath is all 0s. The port structure of the binary multiplier remains unchanged, but *Control_Unit* is modified to include another input port, *Empty*. *Datapath_Unit* is modified to include an input port for *Start* and an output port for *Empty*.

The Verilog behavioral description of *Multiplier_STG_1* is presented in Example 10-2. Like *Multiplier_STG_0*, this machine's controller ignores *Start* if it is asserted during a multiplication sequence, but additional logic flushes *product* if *Start* is asserted with an empty data word while *Ready* is asserted. This removes the residual value of *product* that resulted from a previous multiplication sequence. Figure 10-23 shows waveforms from an exhaustive simulation to verify that *code_error* is 0 for all of the patterns[5]; the waveforms also demonstrate that the machine transitions from *S_idle* to *S_5* if *Empty* is asserted at the first clock at which *Start* is asserted after *Reset* is deasserted. Figure 10-24 shows that *Start* is ignored if either data word is 0. Figure 10-25 shows that *product* is flushed if *Start* is asserted while *Ready* and *Empty* are asserted. The simulation results in Figure 10-26 show the signal assertions forming the product $9_{10} \times 11_{10}$[6].

[5]The shaded areas indicate regions where the resolution of the display does not permit graphing the result.
[6]The exercise of showing that *Start* is ignored during multiplication is left to the reader.

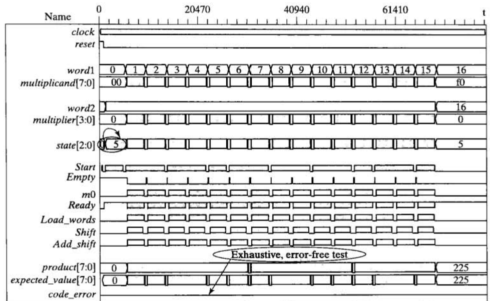


FIGURE 10-23 Simulation results for *Multiplier_STG_1*, an efficient sequential multiplier.

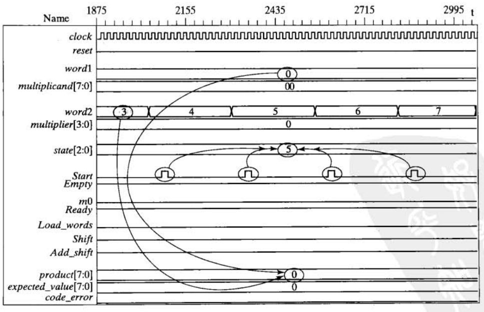


FIGURE 10-24 Simulation results for *Multiplier_STG_1*, demonstrating immediate termination if *Ready* and *Start* are asserted and a data word is 0.

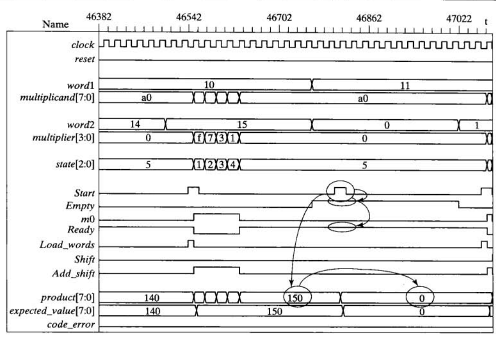


FIGURE 10-25 Simulation results for *Multiplier_STG_1*, demonstrating that *product* is flushed if *Start* is asserted while *Ready* and *Empty* are asserted.

Example 10.2

```
module Multiplier_STG_1 #(parameter L_word = 4)(
  output    [2*L_word -1: 0]          product,
  output                              Ready,
  input     [L_word -1: 0]            word1, word2,
  input                               Start, clock, reset
  );
  wire                                Ready, m0, Empty, Load_words, Shift,
                                        Add_shift;

  Control_Unit M0_Controller (Load_words, Shift, Add_shift, Ready, m0, Empty,
    Start, clock, reset);

  Datapath_Unit M1_Datapath
  (product, m0, Empty, word1, word2, Ready, Start, Load_words, Shift, Add_shift,
    clock, reset);

endmodule
```

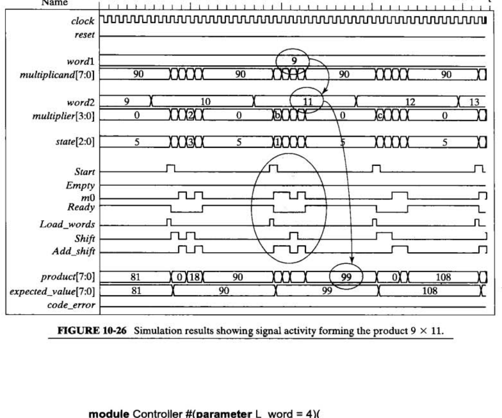


FIGURE 10-26 Simulation results showing signal activity forming the product 9 × 11.

```
module Controller #(parameter L_word = 4)(
  output                              Load_words, Shift, Add_shift, Ready,
  input                               m0, Empty, Start, clock, reset
);
  parameter                           L_state = 3;
  reg           [L_state -1: 0]       state, next_state;
  parameter                           S_idle = 0, S_1 = 1, S_2 = 2, S_3 = 3,
                                        S_4 = 4, S_5 = 5;
  reg                                 Load_words, Shift, Add_shift;
  assign                              Ready = ((state == S_idle) && !reset) ||
                                        (state == S_5);

  always @ (posedge clock or posedge reset)         // State transitions
   if (reset) state <= S_idle; else state <= next_state;

  always @ (state, Start, m0, Empty) begin          // Next state and control
                                                       logic
   Load_words = 0; Shift = 0; Add_shift = 0;
   case (state)
```

```
      S_idle:  if (Start && Empty) next_state = S_5;
               else if (Start) begin Load_words = 1; next_state = S_1; end
               else next_state = S_idle;
      S_1:     begin if (m0) Add_shift = 1; else Shift = 1; next_state = S_2; end
      S_2:     begin if (m0) Add_shift = 1; else Shift = 1; next_state = S_3; end
      S_3:     begin if (m0) Add_shift = 1; else Shift = 1; next_state = S_4; end
      S_4:     begin if (m0) Add_shift = 1; else Shift = 1; next_state = S_5; end
      S_5:     if (Empty) next_state = S_5;
               else if (Start) begin Load_words = 1; next_state = S_1; end
               else next_state = S_5;
      default: next_state = S_idle;
    endcase
  end
endmodule

module Datapath #(parameter L_word = 4)(
  output     [2*L_word -1: 0]     product,
  output                          m0, Empty,
  input      [L_word -1: 0]       word1, word2,
  input                           Ready, Start, Load_words, Shift,
                                  Add_shift, clock, reset
);
  reg        [2*L_word -1: 0]     multiplicand;
  reg        [L_word -1: 0]       multiplier;
  assign                          m0 = multiplier[0];
  assign                          Empty = (~|word1)|| (~|word2);

  // Register/Datapath Operations
  always @ (posedge clock, posedge reset) begin
    if (reset == 1'b1) begin multiplier <= 0; multiplicand <= 0; product <= 0; end
    else if (Start && Empty && Ready) product <= 0;
    else if (Load_words) begin
      multiplicand <= word1;
      multiplier <= word2;
      product <= 0;
    end
    else if (Shift) begin
      multiplier <= multiplier >> 1;
      multiplicand <= multiplicand << 1;
    end
    else if (Add_shift) begin
      product <= product + multiplicand;
      multiplier <= multiplier >> 1;
      multiplicand <= multiplicand << 1;
    end
  end
endmodule
```

End of Example 10.2

10.3.6 ASMD-Based Sequential Binary Multiplier

STGs are convenient tools for designs that have only a few states, but are cumbersome when the number of states is large. For example, the STG-based designs of the 4-bit binary multiplier in Examples 10.1 and 10.2 do not scale with *L_word*, the size of the datapath. For longer words, additional code must be inserted for each new bit to handle additional state transitions. The growth in code is linear in the size of *L_word*. Portable, re-usable HDL models need to be written in a style that does not require the body of a description to be modified by a third party. Either a scalable STG representation or an alternative method, such as an algorithmic state machine chart and datapath (ASMD), must be developed. ASMD charts facilitate scalable, portable, and re-usable designs.

ASMD charts link a datapath to its controller, and display the evolution of a digital machine's activity as an algorithm executes under the influence of inputs. As an alternative method for the design of a sequential multiplier, we will partition the system into a datapath and controller and encapsulate them in separate Verilog modules. The datapath operations will be the same as for *Multipier_STG_0* in Figure 10-19, but the ASMD-based version of the machine will have a slightly enhanced controller. Recall that the machine *Multiplier_STG_1* avoids needless activity if *word_1* or *word_2* are 0 when *Start* is asserted in state *S_idle*. Signal *Empty* will have the same role in this version, but the machine will be further enhanced by terminating activity and asserting *Ready* as soon as the value of the shifted multiplier is 1. (Recall that the versions in *Multiplier_STG_0* and *Multiplier_STG_1* traverse the entire chain of states independently of the content of *multiplier*.) If the value of *multiplier* is 0, *product* is completely formed, so the sequence can terminate when *multiplier* is detected to be 0. Thus, termination of the algorithm is data-dependent. On the other hand, the previously considered models based on STGs consider all of the bits of *multiplier*. The ASMD chart of the datapath controller is shown in Figure 10-27. Note that the ordering of the decision diamonds implies that *Start* is decoded with higher priority than *Empty*.

The ASMD chart for the machine's controller specifies its state transitions and the output signals that are to be asserted during its operation. These output signals will control the datapath unit. In this design, the controller's input signals are the primary input *Start* and the internal status signals *Empty* and *multiplier*. Its output signals are *Ready, Flush, Load_words, Shift*, and *Add. Ready* signals that the unit is ready to accept a command to multiply. It must not assert while *reset* is asserted and must not assert after *Start* has been asserted, until the machine has formed *product*. If a data word is empty while *Start* is asserted in *S_idle* or in *S_done*, the machine enters *S_done*, bypassing any further execution. If *Start* is asserted while the state is in *S_done* and a data word is empty, *Flush* is asserted to empty *product* of any residual value from a previous multiplication.

We saw in Chapter 5 that an ASMD chart is an annotated ASM chart linking the controller to the datapath it controls, by specifying the datapath operations that are to occur synchronously with the state transitions, under the control of the indicated assertions. This additional information can be an aid in verifying that the functionality is correct, and simplifies the task of designing the overall machine.

The machine here has four states: *S_idle, S_shifting, S_adding*, and *S_done*. It enters *S_idle* when *reset* is asserted (asynchronously), and remains there, with *reset* de-asserted

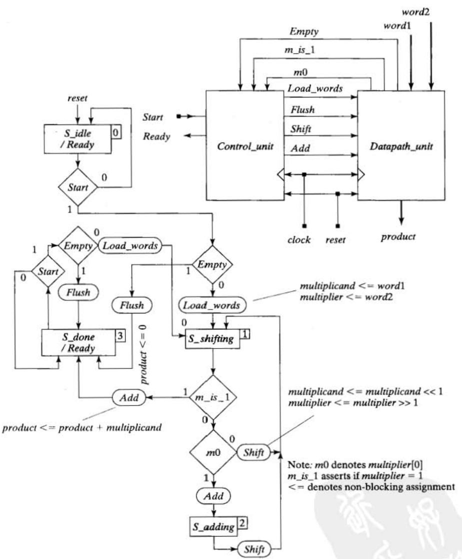


FIGURE 10-27 Block diagram and ASMD chart for *Multiplier_ASM_0*, a sequential binary multiplier, annotated with datapath operations.

and *Ready* asserted, until *Start* is asserted. If *Start* is asserted and a data word is empty, *Flush* is asserted and the state enters *S_done*. Otherwise, the machine asserts *Load_words* and will enter *S_shifting* at the next active edge of the clock. At the clock edge, with *Load_words* asserted, the *multiplicand* and *multiplier* registers will load *word1* and *word2*,

respectively. In *S_shifting*, if *multiplier* has a value of 1, the machine will transition to *S_done*. If not, *m0* (*multiplier[0]*) is tested. If it is 1, *Add* is asserted and the machine will transition to *S_adding* at the next active edge of *clock*. In conjunction with this transition, the contents of register *multiplicand* will be added to the register *product*. In *S_shifting*, if the LSB of *multiplier* is 0, the controller asserts *Shift*, and will transition to *S_shifting*. At the same edge of *clock*, with *Shift* asserted, the contents of *multiplier* will shift by 1 bit toward the register's LSB, and the contents of *multiplicand* will shift by 1 bit toward the register's MSB. In *S_adding, Shift* is asserted, and at the next edge of *clock* the machine will transition back to *S_shifting*. The register operations induced by *Shift* are shown on the ASMD chart as nonblocking assignments with Verilog operator notation.[7] Note that the machine enters *S_done* immediately if a data word is 0, and terminates as soon as the multiplier has a value of 1. This enhanced functionality can be implemented at minor cost of physical hardware (silicon area).

The Verilog code for *Multiplier_ASM_0* is given in Example 10.3. The controller resides in *Control_Unit* and the datapath resides in *Datapath_Unit*. In general, all of the signals that appear in the decision diamonds of the ASM chart must be included in the event-control expression of the combinational behavior that describes the next-state and output logic of the controller. All of the variables that are assigned by the behavior are given an initial (default) value of 0, to avoid the synthesis of an inadvertent latch. Note, *reset* is omitted in the next-state function because it is properly accounted for in the synchronous behavior. The decoding at *S_idle* and *S_3* handles situations in which the machine is directed to multiply with a data word that is 0.

The description of *Multiplier_ASM_0* is parameterizable. A single parameter, *L_word*, can be change to accommodate an arbitrary word length, without other modifications to the model.

Example 10.3

```
module Multiplier_ASM_0 #(parameter L_word = 4) (
  output [2*L_word -1: 0]        product,
  output                         Ready,
  input                          [L_word -1: 0] word1, word2
  );
                                  wire Empty, Load_words, Flush, Add, Shift;

  Control_Unit
   M0_Controller (Ready, Load_words, Flush, Add, Shift, Start, Empty, m0,
      m_is_1, clock, reset);

  Datapath_Unit
   M1_Datapath (product, Empty, m0, m_is_1, word1, word2, Load_words, Flush,
      Add, Shift, clock, reset);

endmodule
```

[7]The annotation associating datapath operations with the arcs of an ASMD chart is similar to register transfer notation used in textbooks on computer architecture.

```
module Control_Unit #(parameter _word = 4) (
  output          Ready,
  output reg      Load_words, Flush, Add, Shift,
  input           Start, Empty, m0, m_is_1, clock, reset
);

  reg [1: 0]      state, next_state;
  parameter       S_idle = 0, S_shifting = 1, S_adding = 2, S_done = 3;
  assign          Ready = ((state == S_idle) && !reset) || (state == S_done);

  always @ (posedge clock, posedge reset)
    if (reset == 1'b1) state <= S_idle; else state <= next_state;

  always @ (state, Start, Empty, m_is_1, m0) begin
    Load_words = 0; Flush = 0; Add = 0; Shift = 0;
    case (state)
      S_idle:      if (!Start) next_state = S_idle;
                   else if (Start && !Empty) begin Load_words = 1; next_state
                    = S_shifting; end
                   else if (Start && Empty) begin Flush = 1; next_state =
                    S_done; end

      S_shifting:  if (m_is_1) begin Add = 1; next_state = S_done; end
                   else if (m0) begin Add = 1; next_state = S_adding; end
                   else begin Shift = 1; next_state = S_shifting; end

      S_adding:    begin Shift = 1; next_state = S_shifting; end

      S_done:      if (Start == 0) next_state = S_done;
                   else if (Empty) begin Flush = 1; next_state = S_done; end
                   else begin Load_words = 1; next_state = S_shifting; end

      default:     next_state = S_idle;
    endcase
  end
endmodule

module Datapath_Unit #(parameter L_word = 4) (
  output reg [2*L_word -1: 0]  product,
  output                       Empty, m0, m_is_1,
  input      [L_word -1: 0]    word1, word2,
  input                        Load_words, Flush, Add, Shift,
                               clock, reset
);

  reg     [2*L_word -1: 0]     multiplicand;
  reg     [L_word -1: 0]       multiplier;
  assign assignEmpty = ((word1 == 0) || (word2 == 0));
  assign m_is_1 = (multiplier == 1'b1);
  assign m0 = multiplier [0];

  always @ (posedge clock, posedge reset) begin
    if (reset == 1'b1) begin multiplier <= 0; multiplicand <= 0; product <= 0; end
    else if (Flush)
```

```
      product <= 0;
    else if (Load_words == 1) begin
      multiplicand <= word1;
      multiplier <= word2;
      product <= 0;
    end
    else if (Shift) begin
      multiplicand <= multiplicand << 1;
      multiplier <= multiplier >> 1;
    end
    else if (Add) product <= product + multiplicand;
  end
endmodule
```

End of Example 10.3

The simulation results shown in Figures 10-28 to 10-30 show the state transitions specified by the ASMD chart. In Figure 10-28, assertion of *reset* drives the

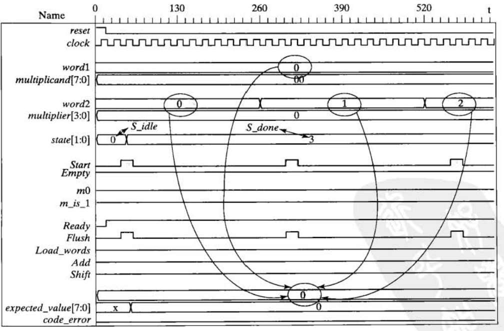


FIGURE 10-28 Simulation results for *Multiplier_ASM_0*, showing activity after initial reset with data words of 0.

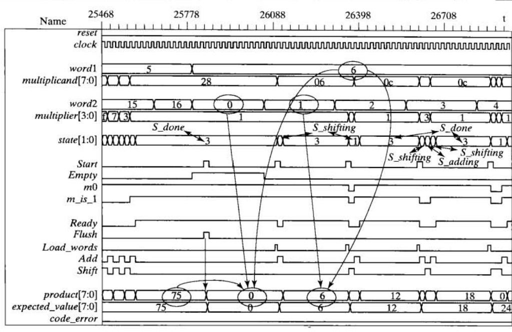


FIGURE 10-29 Simulation results for *Multiplier_ASM_0*, showing proper handling of an empty word.

state to *S_idle* and flushes *product*. With *multiplicand* having a value of 0, assertion of *Start* drives the state to *S_done*. The state remains in *S_done* with subsequent assertions of *Start* because *multiplicand* is 0, even though *multiplier* is not 0. *Ready* is asserted in *S_idle* after *reset* is de-asserted, and remains asserted in *S_done*. Figure 10-29 shows simulation activity in which *product* has a residual value of 75_{10}, which resulted from the product of 5 and 15[8] With *word2* = 0, assertion of *Start* asserts *Flush*, causing *product* to become 0. This case corresponds to a data word transitioning to 0 from the value that led to the residual content of *product*. In this case, if *Start* is asserted *product* is flushed, *state* remains in *S_idle*, and *Ready* is re-asserted. Later, with *word2* = 1 and *word1* = 6, the product becomes 6, and so forth. The graph is annotated to highlight the state transitions. Notice how the assertions of

[8]The displayed value of *word2* = 16 is displayed at the end of a **for** loop. It is formed in the testbench as an integer and the input port of the multiplier passes only the least significant four bits. The value *word2* = 16 is not processed by the multiplier because *Start* is not asserted by the testbench until *word2* begins a new sequence with *word2* = 0...

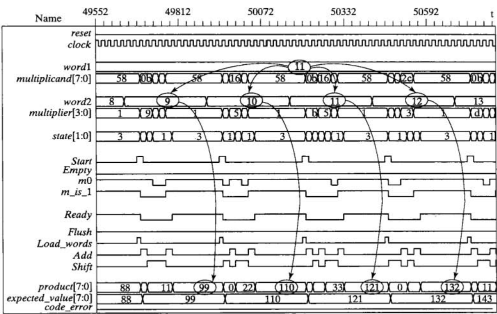


FIGURE 10-30 Simulation results showing four cycles of multiplication for *Multiplier_ASM_0*, an ASMD-based sequential multiplier.

Shift and *Add* correspond to *S_shifting* and *S_adding*, and that *Ready* is de-asserted until the multiplication is complete. Figure 10-30 shows four cycles of multiplication activity extracted from an exhaustive simulation. The shaded waveforms illustrate the sequential assertions of *Ready, Start, Load_words, Add,*, and *Shift*.

10.3.7 Efficient ASMD-Based Sequential Binary Multiplier

The multiplier described by *Multiplier_STG_0* is inefficient because it performs the shift and add operations in separate clock cycles. Likewise, *Multiplier_ASM_0* executes more cycles than are needed to form *product*. Figure 10-31 shows the ASMD chart of a more efficient machine, having only two states, whose datapath operations add and shift in the same cycle while *Add_shift* is asserted. The Verilog

PDG

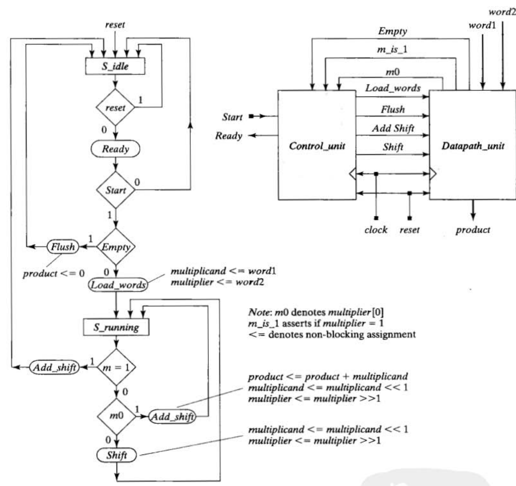


FIGURE 10-31 Block diagram and ASMD chart for *Multiplier_ASM_1*, an efficient ASMD-based multiplier.

description of *Multiplier_ASM_1* is presented in Example 10.4, and simulation results are shown in Figures 10-32 to 10-34. Note that *Multiplier_ASM_1* requires fewer clock cycles to compute *product*, and that *Ready* asserts when *multiplier* becomes empty.

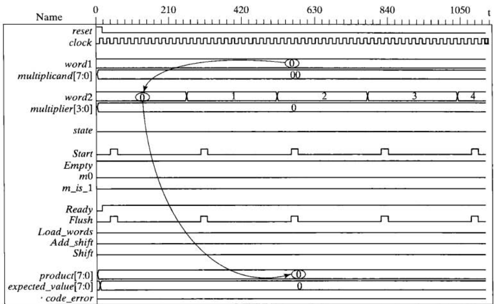


FIGURE 10-32 Simulation results for *Multiplier_ASM_1*, showing activity after assertion of *reset*.

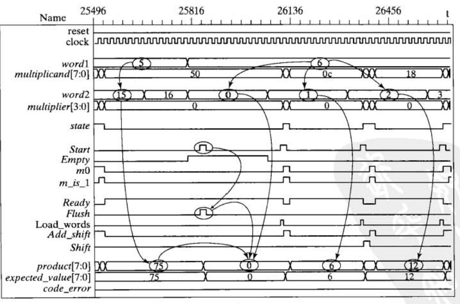


FIGURE 10-33 Simulation results for *Multiplier_ASM_1*, showing correct flushing action when multiplication is attempted with an empty data word.

PDG

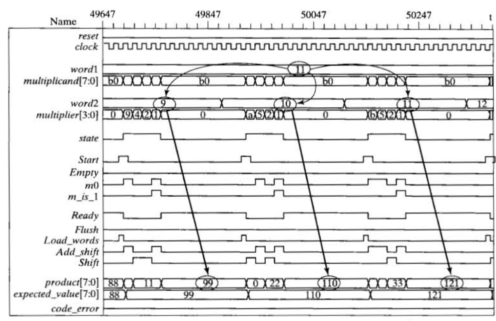


FIGURE 10-34 Simulation results for *Multiplier_ASM_1*, showing a sample of three cycles of multiplication.

Example 10.4

```
module Multiplier_ASM_1 #(parameter L_word = 4)(
  output       [2*L_word -1: 0]      product,
  output                              Ready,
  input        [L_word -1: 0]         word1, word2,
  input                               Start, clock, reset
);
  wire                               Empty, Load_words, Flush,
                                      Add_shift, Shift;

  Control_Unit M0_Controller
   (Ready, Load_words, Flush, Add_shift, Shift, Start, Empty, m0, m_is_1, clock,
     reset);

  Datapath_Unit M1_Datapath
   (product, Empty, m0, m_is_1, word1, word2, Load_words, Flush, Add_shift,
     Shift, clock, reset);
endmodule
```

```
module Control_Unit (
  output Ready, output reg Load_words, Flush, Add_shift, Shift,
  input Start, Empty, p0, c_is_ws, clock, reset
);
  reg                                         state, next_state;
  parameter                                   S_idle = 0, S_running = 1;
  assign                                      Ready = (state == S_idle) &&
                                                (!reset);

  always @ (posedge clock, posedge reset)           // State transitions
   if (reset == 1'b1) state <= S_idle; else state <= next_state;

  always @ (state, Start, Empty, p0, c_is_ws) begin     // Combinational logic
                                                          for ASM-based
                                                          controller
    next_state = S_idle; Flush = 0; Load_words = 0; Shift = 0; Add_shift = 0;
    case (state)
      S_idle:       if (!Start) next_state = S_idle;
                    else if (Empty) begin next_state = S_idle; Flush = 1; end
                    else begin Load_words = 1; next_state = S_running; end

      S_running:    if (c_is_ws) begin next_state = S_idle; Add_shift = 1; end
                    else if (p0) begin Add_shift = 1; next_state = S_running; end
                    else begin Shift = 1; next_state = S_running; end
      default:      next_state = S_idle;
    endcase
  end
endmodule

module Datapath_Unit #( parameter L_word = 4)(
  output reg [2*L_word -1: 0] product,
  output Empty, m0, m_is_1,
  input [L_word -1: 0]  word1, word2,
  input Load_words, Flush, Add_shift, Shift, clock, reset
);

  reg          [2*L_word -1: 0]       multiplicand;
  reg          [L_word -1: 0]         multiplier;
  assign                              Empty = (word1 == 0) ||
                                        (word2 == 0);
  assign                              m0 = multiplier[0];
  assign                              m_is_1 = (multiplier == 1);

  always @ (posedge clock, posedge reset)
   if (reset) begin multiplier <= 0; multiplicand <= 0; product <= 0; end
   else begin
   if (Flush) product <= 0;
```

```
      else if (Load_words == 1) begin
        multiplicand <= word1;
        multiplier <= word2;
        product <= 0;
      end
      else if (Shift) begin
        multiplicand <= multiplicand << 1;
        multiplier <= multiplier >> 1; end
      else if (Add_shift) begin product <= product + multiplicand;
        multiplicand <= multiplicand << 1;
        multiplier <= multiplier >> 1;
      end
    end
endmodule
```

End of Example 10.4

10.3.8 Summary of ASMD-Based Datapath and Controller Design

The previous examples have illustrated how STGs and ASMD charts can be used with an HDL to describe and design a state machine controlling a datapath. Because it leads naturally to portable descriptions of behavior, we summarize below the basic elements of an ASMD chart-based design method for datapaths and their controllers:

1. Partition the design effort into (a) a (edge-sensitive) synchronous behavior controlling state transitions, (b) one or more (level-sensitive) combinational behaviors and/or continuous assignments specifying the next-state and output logic of the controller, and (c) a (edge-sensitive) synchronous behavior describing the datapath operations that are controlled by the logic developed in 1(b) above.
2. In a level-sensitive behavioral description of the combinational logic of the controller, make assignments "by exception", to ensure that accidental latches will not be synthesized, by (a) initializing all output variables to 0 to and (b) assigning a default next state.
3. Use blocking assignments in a level-sensitive behavior describing the combinational logic for next state and output of the datapath controller.
4. Describe the reset action of the machine in the edge-sensitive behaviors, not in the level-sensitive behaviors that model combinational logic.
5. Do not mix datapath operations with the next-state and output logic. Write a separate synchronous (edge-sensitive) behavior describing the datapath operations supporting the architecture.

6. In the behavior describing datapath operations, use nonblocking assignments, and condition the activity flow on the output signals generated by the datapath controller.
7. Be sure that the sensitivity list of the level-sensitive behavior describing the combinational logic for the next state and the outputs is complete or, as an alternative, use the wildcard operator pair @* or @(*).

10.3.9 Reduced-Register Sequential Multiplier

The previously considered architectures for the binary multiplier use separate registers to hold *multiplicand*, *multiplier*, and *product*. A shift register sized at *L_word2*L_word* initially holds *multiplicand* and accommodates the shifting operations that occur at each step of the multiplication sequence. An alternative, and more efficient, architecture is shown in Figure 10-35, where the register for *multiplicand* is hardwired to the adder, as are the leftmost (*L_word* + 1) bits of *product*. The value of *multiplier* is initially stored in the leftmost *L_word* bits of *product*. The row sums are placed in the leftmost bits of *product* as they are formed, and the contents of *product* are shifted to the right (i.e., *product* moves relative to a fixed register holding *multiplicand*). At each step, the LSB of *product* determines whether *multiplicand* is to be added to *product*. The sequence of operations continues until all the bits of *multiplier* have been shifted out of *product*, leaving only the result of the multiplication. This scheme eliminates a separate register for *multiplier*, and reduces the size of the register for *multiplicand* by a factor of 2. Also, the register for *multiplicand* is fixed (i.e., not a shift register). The adder required to generate the sums is also reduced in size by a factor of 2, saving silicon area and improving speed. Figure 10-35 also illustrates the movement of data forming the product $215_{10} \times 23_{10} = 4945_{10}$.

The interface signals and the controller for this datapath architecture are based on the ASMD chart in Figure 10-36. This machine is efficient, like *Multiplier_ASM_1*, having only two states, *S_idle* and *S_running*. Given that the multiplier is stored in the product register and is shifted out of the register as the process evolves, a counter is used to determine when the process is complete. (Problem 6 at the end of the chapter specifies a design that terminates the multiplication process as soon as the shifted multiplier subword is empty of 1s.) The controller generates a signal, *Increment*, which controls the counter.

The Verilog code for *Multiplier_RR_ASM* is given in Example 10.5. Note that the register holding *product* is 1 bit larger than in the previous models, because the addition and concatenation operations induced by *Add_shift* occur before the shift operation and might generate a carry that would otherwise be dropped, even though the final result will fit in a register having *L_word* bits. This condition does not arise in the previous designs, because their multiplication process builds sums from right to left, and the product register is large enough to accommodate any intermediate carry. If the product register were sized to only 2 * *L_word* here, an intermediate carry will overflow and be lost. This condition might not have been detected by a sporadic testing scheme, but it was revealed by the exhaustive testbench's error detection signal.

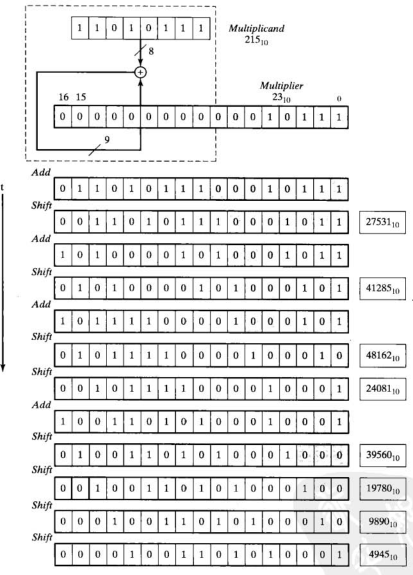


FIGURE 10-35 Architecture and data movement for a binary multiplier with reduced registers.

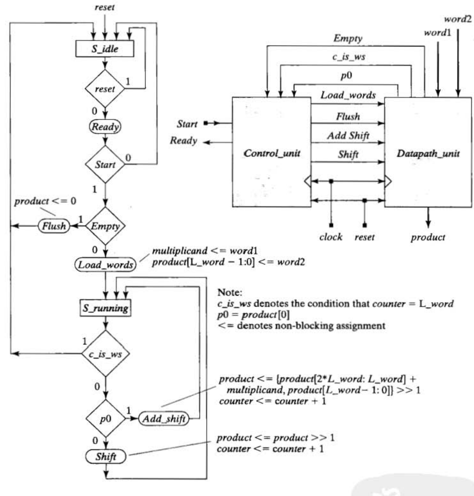


FIGURE 10-36 ASMD chart for the controller of *Multiplier_RR_ASM*, a reduced-register ASM-based sequential multiplier, annotated with datapath operations.

Example 10.5

```
module Multiplier_RR_ASM #(parameter L_word = 4)(
 output      [2*L_word: 0]     product,
 output                        Ready,
 input       [L_word -1: 0]    word1, word2,
 input                         Start, clock, reset
);
```

```
  Control_Unit M0_Controller (
   .Ready(Ready), .Load_words(Load_words), .Flush(Flush), .Add_shift(Add_shift),
    .Shift(Shift),
   .Start(Start), .Empty(Empty), .p0(p0), .c_is_ws(c_is_ws), .clock(clock),
    .reset(reset));

   Datapath_Unit M1_Datapath (
    .product(product), .Empty(Empty), .p0(p0), .c_is_ws(c_is_ws), .word1(word1),
     .word2(word2),
    .Load_words(Load_words), .Flush(Flush), .Add_shift(Add_shift), .Shift(Shift),
    .clock(clock), .reset(reset));

endmodule

module Control_Unit (
 output Ready, output reg Load_words, Flush, Add_shift, Shift,
 input Start, Empty, p0, c_is_ws, clock, reset
);
 reg              state, next_state;
 parameter        S_idle = 0, S_running = 1;
 assign           Ready = (state == S_idle) && (!reset );
 always @ (posedge clock, posedge reset)          // State transitions
  if (reset == 1'b1) state <= S_idle; else state <= next_state;

  always @ (state, Start, Empty, p0, c_is_ws) begin  // Comb logic for next state
                                                        and outputs
   Flush = 0; Load_words = 0; Shift = 0; Add_shift = 0;
   case (state)
    S_idle:          if (!Start) next_state = S_idle;
                     else if (Empty) begin next_state = S_idle; Flush = 1; end
                     else begin Load_words = 1; next_state = S_running; end

    S_running:       if (c_is_ws) next_state = S_idle;
                     else if (p0) begin Add_shift = 1; next_state = S_running;
                      end
                     else begin Shift = 1; next_state = S_running; end

    default:         next_state = S_idle;
   endcase
 end
endmodule

module Datapath_Unit #(parameter L_word = 4, L_count = 3)(
  output reg [2*L_word: 0] product,
  input [L_word -1: 0] word1, word2,
  input Load_words, Flush, Add_shift, Shift, clock, reset
);

  reg          [L_word -1: 0]       multiplicand;
  reg          [L_count -1 : 0]     counter;
  assign                            Empty = (word1 == 0) || (word2 == 0);
  assign                            p0 = product[0];
  assign                            c_is_ws = (counter == L_word);
```

```
always @ (posedge clock, posedge reset)
 if (reset == 1'b1) begin multiplicand <= 0; product <= 0; counter <= 0; end
 else begin
  if (Flush) product <= 0;
  if (Load_words == 1)
    begin multiplicand <= word1; product <= word2; counter <= 0; end
  if (Shift) begin product <= product >> 1; counter <= counter + 1; end
  if (Add_shift) begin
   product <= {product[2*L_word: L_word] + multiplicand, product[L_word -1: 0]}
     >> 1;
   counter <= counter + 1;
  end
 end
endmodule
```

End of Example 10.5

A sample of waveforms produced by *Multiplier_RR_ASM* is shown in Figure 10-37. (The signal *multiplier* is formed from *word2* in the test bench and is not part of the

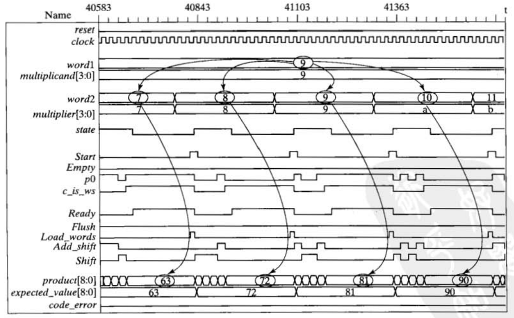


FIGURE 10-37 Register transfers in *Multiplier_RR_ASM* demonstrating multiple cycles of activity.

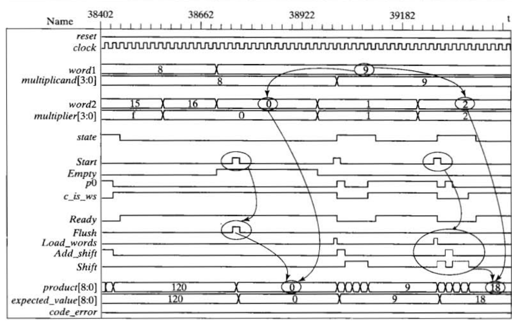


FIGURE 10-38 Simulation results for *Multiplier_RR_ASM* demonstrating correct register flushing action.

machine.) The simulation results in Figure 10-38 demonstrate the machine's ability to immediately terminate execution and re-assert *Ready* if a data word is 0 when *start* is asserted in *S_idle*. Note that the waveform display has been organized to form a group composed of the primary input and the status signals of the controller, and a second group composed of the output signals of the controller. This arrangement can be helpful in verifying and understanding the signal activity and overall behavior of the machine.

The design based on the ASMD chart in Figure 10-36 specifies that *state* return to *S_running* after *product* is finally formed, *before* returning to *S_idle* and asserting *Ready*. Thus, *one cycle of execution is wasted*. As an alternative design, the ASMD chart in Figure 10-39 moves the test of the counter in the datapath to occur *after* the datapath signals have been formed, and completes the multiplication sequence by directing *state* to *S_idle* without returning to *S_running*. This eliminates the wasted cycle and asserts *Ready* as soon as *product* is valid. The modified code is shown in Figure 10-40.

10.3.10 Implicit-State-Machine Binary Multiplier

An implicit-state machine [6–8] consists of a single cyclic behavior with multiple embedded, edge-sensitive event-control expressions that specify an evolution of clock cycles and implicitly define the states of a machine (see Chapter 6). Unlike the machines in the previous examples, an implicit-state machine does not have an explicitly

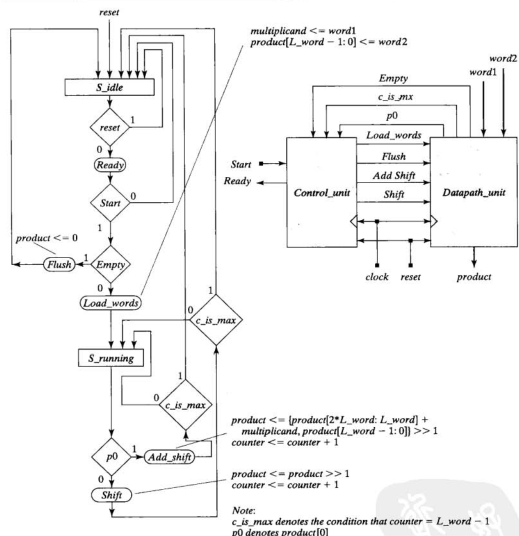


FIGURE 10-39 Alternative ASMD chart to recover the wasted cycle in *Multiplier_RR_ASM*.

```
/*Modified version to recover final cycle*/
  S_running:    begin
                  if (p0) Add_shift = 1; else Shift = 1;
                  if (c_is_max) next_state = S_idle; else next_state = S_running;
                end

// Note: p0 = product[0], c_is_max asserts if counter = L_ word −1.
```

FIGURE 10-40 Code to modify *Multiplier_RR_ASM* to recover a wasted cycle of operation.

defined state, nor does it have an explicit state-transition behavior. Some designers prefer the simplicity and clarity of the descriptive style offered by an implicit-state machine, but its utility is limited. Be aware that the structure of the STG or the ASM chart of an implicit-state machine has the restriction that a given state may be entered from only one other state. Many systems do not satisfy this constraint. Verilog models of implicit-state machines require careful consideration of reset signals to ensure that an asserted reset signal returns the machine to the beginning of the sequence of clock cycles, regardless of the clock cycle in which it is asserted. For synthesis, the active edge of the synchronizing signal (clock) of an implicit-state machine must have the same polarity (e.g., rising) at all state transitions.

An implicit-state machine controller for the synchronous binary multiplier whose architecture was shown in Figure 10-35 is presented in Example 10.6. The controller in *Multiplier_IMP_1* was designed to satisfy the following requirements:

1. A signal, *Ready*, is to be asserted at the first active edge of *clock* after *reset* has been cycled through assertion and de-assertion. *Ready* indicates that the multiplier is ready to respond to a *Start* signal, with *reset* de-asserted. *Ready* is to be de-asserted while *reset* is asserted, and, after *Ready* has been asserted, it is to remain asserted until the first active edge of the clock occurs with *Start* asserted.
2. The machine must begin a multiplication sequence at the first active edge of *clock* at which *Start* and *Ready* are both asserted.
3. Once the multiplication sequence is initiated, the machine is to assert a signal, *Ready*, at the first active edge of *clock* at which *multiplier* is 0. The assertion of *Ready* is to coincide with the formation of the value of *product*. *Ready* indicates that *product* can be read by an external processor.
4. Once the multiplication sequence is initiated, *Start* is to be ignored until *Ready* is asserted.
5. The datapaths for *word1* and *word2* are to be ignored after the multiplicand and multiplier registers have been loaded (i.e., external datapaths are free during the multiplication sequence).
6. Once asserted, *Ready* is to be held asserted until *Start* (or *reset*) is re-asserted.
7. The machine must recover from a random assertion of *reset* during a multiplication sequence (i.e., must recover from a running reset).
8. The machine must operate correctly if *Start* is asserted randomly, and if *Start* has a duration of more than one cycle of *clock*.

The controller in *Multiplier_IMP_1* is an implicit finite-state machine with the features specified above. The additional code that results from the requirement that an implicit-state machine recover correctly from an assertion of *reset* in any clock cycle is reduced in this example by use of a task, *Clear_and_Set_Regs*, to reset all registers affected by *reset*. Note that the datapath operations are relatively simple, but the controller is more elaborate and complex.

Controller_IMP_1, has a single cyclic behavior with the following threads of activity: (1) a single-cycle thread in which the machine idles until *Start* is asserted, (2) a

two-cycle thread that detects whether a data word is 0 and, if so, terminates immediately, (3) a multicycle thread in which data words are loaded, followed by a sequence of assertions of *Add_shift* or *Shift*, depending on the running content of the bit *multiplier[0]*. Each cycle of activity is initiated by the rising edge of *clock*. If *reset* is asserted, the activity aborts the named block, *Main_Block*, and returns to monitor the first event-control expression.

For convenience and clarity, the three threads of activity within the *Main_Block* of the behavior are labeled as named blocks. In *Idling*, the machine waits for assertion of *Start* (with *Ready* asserted), and asserts *Flush* to remove the residual value of *product*. *Early_Terminate* aborts an attempt to multiply with a zero. In *Load_and_Multiply*, the controller asserts *Load_words* in the first cycle, then issues commands to shift or shift and add in successive clock cycles until the multiplication process is complete. In the model shown here, an additional signal, *Done*, is used in the controller to expose a key detail of the operation of the machine. *Done* asserts at the end of the loop that computes *product*. The loop is not data-dependent, and executes through all bits of *multiplier*. If an external agent uses *Ready* to initiate a multiplication sequence after *Ready* is asserted, and while the machine is executing the loop, *Start* will be ignored. This could lead to ambiguity in associating *product* with *word1* and *word2*. For unambiguous operation, *Start* should not be re-asserted until *Done* is asserted. The control signals generated by the controller are registered, so the signals are formed in the clock cycle before the cycle in which they are used.

Example 10.6

```
module Multiplier_IMP_1 #(parameter L_word = 4)(
  output [2*L_word -1: 0]                product,
  output                                 Ready, Done,
  input [L_word -1: 0] word1, word2,
  input                                  Start, clock, reset
);
  wire                                   Empty, w2_0, m_is_1, m1;  // status
                                                                     signals
  wire                                   Flush, Load_words, Shift, Add_shift;

  Control_Unit M0_Controller
   (Ready, Flush, Load_words, Shift, Add_shift, Done, Empty, w2_0, m_is_1, m1,
     Start, clock, reset);

  Datapath_Unit M1_Datapath
   (product, Empty, w2_0, m_is_1, m1, word1, word2, Flush, Load_words, Shift,
     Add_shift, clock, reset);

endmodule
```

```
module Control_Unit #(parameter L_word = 4)(
  output reg       Ready, Flush, Load_words, Shift, Add_shift, Done,
  input            Empty, w2_0, m_is_1, m1,
  input            Start, clock, reset
);
  reg [L_word: 0]  k;

  always
    @ (posedge clock, posedge reset) begin: Main_Block
    if (reset == 1'b1) begin Clear_and_Set_Regs; disable Main_Block; end
    else if (Start != 1) begin: Idling
      Flush <= 0; Ready <= 1;
    end // Idling
    else if (Start && Empty) begin: Early_Terminate
      Flush <= 1; Ready <= 0; Done <= 0;
    @ (posedge clock or posedge reset)
    if (reset == 1'b1) begin Clear_and_Set_Regs; disable Main_Block; end
    else begin
      Flush <= 0; Ready <= 1; Done <= 1;
    end
  end // Early_Terminate

  else if (Start) begin: Load_and_Multiply
   Ready <= 0; Flush <= 0; Load_words <= 1; Done <= 0;Shift <= 0;
   Add_shift <= 0;
   @ (posedge clock, posedge reset)
   if (reset == 1'b1) begin Clear_and_Set_Regs; disable Main_Block; end
   else begin // not reset
     Load_words <= 0;
     if (w2_0) Add_shift <= 1; else Shift <= 1;
     for (k = 0; k <= L_word -1; k = k +1)
       @ (posedge clock, posedge reset)
       if (reset == 1'b1) begin Clear_and_Set_Regs; disable Main_Block; end
       else begin // multiple cycles
         Shift <= 0;
         Add_shift <= 0;
         if (m_is_1) Ready <= 1;
         else if (m1) Add_shift <= 1;
         else Shift <= 1; // Notice use of multiplier[1]
       end // multiple cycles
        Done <=1;
     end // not reset
   end // Load_and_Multiply
  end // Main_Block

  task Clear_and_Set_Regs;
   begin
    Ready <= 1; Flush <= 0; Load_words <= 0; Done <= 1; Shift <= 0; Add_shift <= 0;
   end
```

```
  endtask
endmodule

module Datapath_Unit #(parameter L_word = 4)(
  output reg        [2*L_word -1: 0]    product,
  output                                Empty, w2_0, m_is_1, m1,
  input             [L_word -1: 0]      word1, word2,
  input                                 Flush, Load_words, Shift,
                                          Add_shift, clock, reset
);
  reg               [2*L_word -1: 0]    multiplicand;
  reg               [L_word -1: 0]      multiplier;
  assign                                Empty = (word1 == 0) ||
                                          (word2 == 0);
  assign                                w2_0 = (word2[0] == 1);
  assign                                m_is_1 = (multiplier == 1);
  assign                                m1 = (multiplier[1] == 1);

always @ (posedge clock, posedge reset)
  if (reset == 1'b1) begin multiplier <= 0; multiplicand <= 0; product <= 0; end
  else begin
   if (Flush) product <= 0;
   else if (Load_words == 1) begin
     multiplicand <= word1;
     multiplier <= word2;
     product <= 0; end
   else if (Shift) begin
     multiplier <= multiplier >> 1;
     multiplicand <= multiplicand << 1; end
   else if (Add_shift) begin
     multiplier <= multiplier >> 1;
     multiplicand <= multiplicand << 1;
     product <= product + multiplicand; end
  end
endmodule
```

End of Example 10.6

The simulation results shown in Figure 10-41 for *Multiplier_IMP_1* demonstrate that the controller (1) "wakes up" correctly after an initial assertion of *reset* and immediately flushes *product*, (2_{10}) ignores *Start* while *reset* is asserted, and (3_{10}) responds to the second assertion of *Start* and handles a zero multiplicand correctly, leaving product at its residual value of 0. The waveforms in Figure 10-42 show that the machine handles a zero multiplier correctly—the multiplication process is aborted and the residual value of *product* (60_{10}) is flushed. The results in Figure 10-43

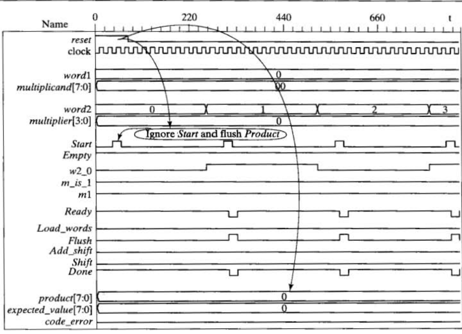


FIGURE 10-41 Simulation results for verification of *Multiplier_IMP_1*, a binary multiplier controlled by an implicit state machine, demonstrating a startup sequence after assertion of power-up *reset*.

show that the machine recovers correctly when *reset* is asserted during a multiplication sequence (an important consideration for an implicit machine). The machine is in the process of multiplying 6_{10} by 11_{10} when *reset* is asserted. The machine flushes *product* and idles to await the next assertion of *Start*, which forms the product of 6_{10} by 12_{10}, and then asserts *Ready* and *Done*. In Figure 10-44, the waveforms show the machine correctly multiplying a sample of data words. The machine was verified exhaustively for 4- and 8-bit words.

The controller in *Multiplier_IMP_1* wastes cycles of execution by cycling through all of the bits of the multiplier, even after *Ready* has been asserted. A more efficient design, *Multiplier_IMP_2*, is shown in Example 10.7,[9] where *Controller_IMP_2* terminates activity when *Ready* is asserted. The controller determines when the content of the multiplier has the value 1 and exits the multiplication loop at the next cycle. This machine also has a more robust controller that detects whether the multiplicand or the multiplier have a value of 1 and issues a command to the datapath to load either *word2* or *word1* directly into the register holding *product*.

[9]A compiler directive is placed at the top of the file to define *word_size* so that a single change in the file customizes *L_word* throughout, but does not require editing of the internal modules.

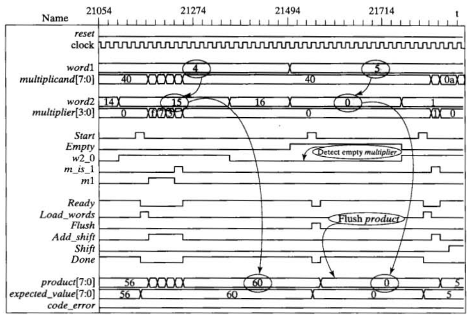


FIGURE 10-42 Simulation results for verification of *Multiplier_IMP_1*, demonstrating correct multiplication with an empty multiplier word (i.e., *word2* = 0).

Example 10.7

```
'define                                                    word_size          4
module Multiplier_IMP_2 #(parameter L_word = 'word_size)(
   output              [2*L_word –1: 0]                    product,
   output                                                  Ready, Done,
   input               [L_word –1: 0]                      word1, word2,
   input                                                   Start, clock, reset
);
   wire                                                    Flush, Load_words,
                                                            Load_multiplier;
   wire                                                    Load_multiplicand;
   wire                                                    Shift, Add_shift;

   Control_Unit M0_Controller
   (
    .Ready(Ready), .Flush(Flush), .Load_words(Load_words),
    .Load_multiplier(Load_multiplier), .Load_multiplicand(Load_multiplicand),
    .Shift(Shift), .Add_shift(Add_shift), .Done(Done),
    .Start(Start), .Empty(Empty), .w1_is_1(w1_is_1), .w2_is_1(w2_is_1),
    .w2_bit0(w2_bit0), .mp_is_1(mp_is_1), .mp_bit1(mp_bit1), .clock(clock),
    .reset(reset)
   );
```

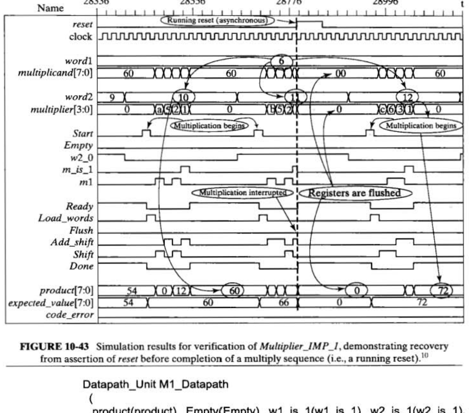


FIGURE 10-43 Simulation results for verification of *Multiplier_IMP_1*, demonstrating recovery from assertion of *reset* before completion of a multiply sequence (i.e., a running reset).[10]

```
Datapath_Unit M1_Datapath
 (
 .product(product), .Empty(Empty), .w1_is_1(w1_is_1), .w2_is_1(w2_is_1),
   .w2_bit0(w2_bit0),
 .mp_is_1(mp_is_1), .mp_bit1(mp_bit1), .word1(word1), .word2(word2),
   .Flush(Flush),
 .Load_words(Load_words), .Load_multiplier(Load_multiplier),
 .Load_multiplicand(Load_multiplicand),
 .Shift(Shift), .Add_shift(Add_shift), .clock(clock), .reset(reset)
 );
endmodule

module Control_Unit #(parameter L_word = 'word_size)(
  output reg          Ready, Flush,
                      Load_words, Load_multiplier,
                      Load_multiplicand,
                      Shift, Add_shift, Done,
  input               Start, Empty, w1_is_1, w2_is_1, w2_bit0, mp_is_1,
                      mp_bit1, clock, reset
);
```

[10]Note that *Done* does not assert until completion of a subsequent multiplication sequence.

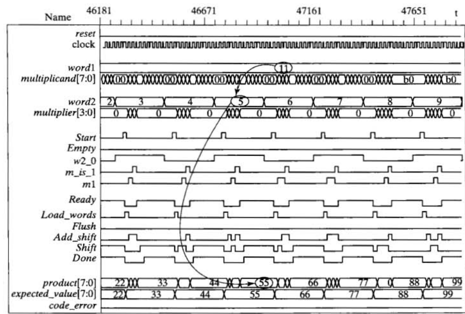


FIGURE 10-44 Simulation results for verification of *Multiplier_IMP_1*, showing multiplication sequences.

```
integer                    k;

always @ (posedge clock, posedge reset) begin: Main_Block
  if (reset == 1'b1) begin Clear_and_Set_Regs; disable Main_Block; end
  else if (Start != 1) begin: Idling
    Flush <= 0; Ready <= 1;
    Load_words <= 0; Load_multiplier <= 0; Load_multiplicand <= 0;
    Shift <= 0; Add_shift <= 0;
  end // Idling

  else if (Start && Empty) begin: Early_Terminate
    Flush <= 1; Ready <= 0; Done <= 0;
    @ (posedge clock, posedge reset)
    if (reset == 1'b1) begin Clear_and_Set_Regs; disable Main_Block; end
    else begin
      Flush <= 0; Ready <= 1; Done <= 1;
    end
    end // Early_Terminate
  else if (Start && w1_is_1) begin: Load_Multiplier_Direct
    Ready <= 0; Done <= 0;
    Load_multiplier <= 1;
    @ (posedge clock, posedge reset)
    if (reset == 1'b1) begin Clear_and_Set_Regs; disable Main_Block; end
```

```
      else begin Ready <= 1; Done <= 1; end
    end     // Load_Multiplier_Direct
    else if (Start && w2_is_1) begin: Load_Multiplicand_Direct
      Ready <= 0; Done <= 0;
      Load_multiplicand <= 1;
      @ (posedge clock, posedge reset)
      if (reset == 1'b1) begin Clear_and_Set_Regs; disable Main_Block; end
      else begin Ready <= 1; Done <= 1; end
    end     // Load_Multiplicand_Direct
    else if (Start ) begin: Load_and_Multiply
      Ready <= 0; Done <= 0; Flush <= 0; Load_words <= 1;
      @ (posedge clock, posedge reset)
      if (reset == 1'b1eClear_and_Set_Regs; disable Main_Block; end
      else begin: Not_Reset
        Load_words <= 0;
        if (w2_bit0) Add_shift <= 1; else Shift <= 1;
        begin: Wrapper
          forever begin: Multiplier_Loop
            @ (posedge clock, posedge reset)
            if (reset == 1'b1) begin Clear_and_Set_Regs; disable Main_Block; end
            else begin: Multiple_Cycles
              Shift <= 0;
              Add_shift <= 0;
              if (mp_is_1) begin Done <= 1;
                @ (posedge clock, posedge reset)
                if (reset == 1'b1) begin Clear_and_Set_Regs; disable Main_Block; end
                else disable Wrapper;
              end           // Done <= 1
              else if (mp_bit1) Add_shift <= 1;
              else Shift <= 1; // Notice use of multiplier[1]
            end // multiple cycles
          end // Multiplier_Loop
        end // Wrapper
        Ready <= 1;
      end // Not_Reset
    end // Load_and_Multiply
  end // Main_Block
  task Clear_and_Set_Regs;
  begin
    Flush <= 0; Ready <= 1; Done <= 1;
    Load_words <= 0; Load_multiplier <= 0; Load_multiplicand <= 0;
    Shift <= 0; Add_shift <= 0;
  end
 endtask
endmodule

module Datapath_Unit #(parameter L_word = 'word_size)(
  output reg          [2*L_word -1: 0]     product,
  output                                   Empty, w1_is_1, w2_is_1, w2_bit0,
                                             mp_is_1, mp_bit1,
```

```
    input          [L_word -1: 0]      word1, word2,
    input                              Flush, Load_words, Load_multiplier,
    input                              Load_multiplicand, Shift, Add_shift,
                                         clock, reset
  );
    reg            [2*L_word -1: 0]    multiplicand;
    reg            [L_word -1: 0]      multiplier;

    assign    Empty = (word1 == 0) || (word2 == 0);
    assign    w1_is_1 = (word1 == 1'b1);
    assign    w2_is_1 = (word2 == 1'b1);
    assign    w2_bit0 = (word2[0] == 1'b1);
    assign    mp_is_1 = (multiplier == 1'b1);
    assign    mp_bit1 = multiplier[1];

    always @ (posedge clock, posedge reset)
      if (reset == 1'b1) begin multiplier <= 0; multiplicand <= 0; product <= 0; end
      else begin
        if (Flush) product <= 0;
        else if (Load_words == 1) begin
          multiplicand <= word1;
          multiplier <= word2;
          product <= 0; end
        else if (Load_multiplicand) begin
          product <= word1; end
        else if (Load_multiplier) begin
          product <= word2; end
        else if (Shift) begin
          multiplier <= multiplier >> 1;
          multiplicand <= multiplicand << 1; end
        else if (Add_shift) begin
          multiplier <= multiplier >> 1;
          multiplicand <= multiplicand << 1;
          product <= product + multiplicand; end
    end
  endmodule
```

End of Example 10.7

Several cycles of simulation activity are shown in Figure 10-45. Note that *Done* asserts as soon as *product* is formed, and that *Ready* asserts when the machine is prepared to initiate another cycle of multiplication. In Figure 10-46, *Start* launches the multiplication of 10_{10} by 2_{10}, and is re-asserted before the computation is complete. The machine ignores the re-assertion of *Start*, completes the multiplication, asserts *Done*, and then asserts *Ready* in the next clock cycle. Then, with *Start* and *Ready* both asserted, the machine multiplies 10_{10} by 9_{10}. The simulation results in Figure 10-47 demonstrates the machine's recovery from a running reset. A multiplication sequence begins at the first active edge after the de-assertion of *reset*.

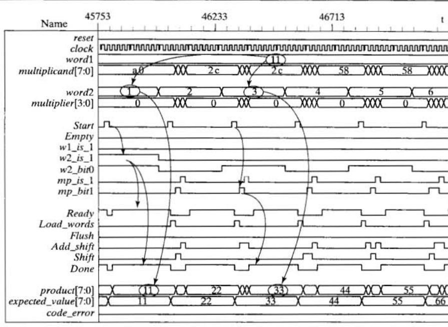


FIGURE 10-45 Simulation results for verification of *Multiplier_IMP_2*, an efficient multiplier controlled by an implicit-state machine, showing multiplication sequences.

10.3.11 Booth's Algorithm Sequential Multiplier

Various algorithms have been developed to improve the performance of sequential multipliers and to simplify their circuitry. Booth's recoding algorithm is widely used because it has a simple hardware realization, requires less silicon area, and can speed up sequential multiplication significantly [1, 3–5, 9–11].

Multipliers that use Booth's algorithm recode the bits of the multiplier to reduce the number of additions required to complete a cycle of multiplication. Only the multiplier is recoded; the multiplicand is left unchanged. A derivative form, called Radix-4 recoding or *bit-pair encoding*, can reduce the number of partial products by a factor of 2 [3] (see Section 10.3.12).

Booth's algorithm is applicable to positive numbers and to negative numbers in 2s complement representation (i.e., signed and unsigned numbers). Thus, a hardware multiplier using Booth recoding does not require modification to accommodate negative numbers. In contrast, multipliers that use signed magnitude representation must extract the magnitudes of the inputs, examine the signs of the data words, and then possibly convert the result to a 2s complement representation (i.e., radix-2 complement form). Multipliers that use Booth's recoding can multiply 2s complement numbers directly. The design given here can multiply two positive numbers, two negative numbers, and a mixture of positive and negative numbers (in 2s complement form).

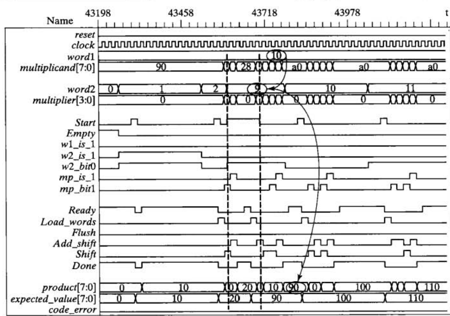


FIGURE 10-46 Simulation results for *Multiplier_IMP_2*, showing simulation activity with *Start* asserted coincidentally with *Done*.

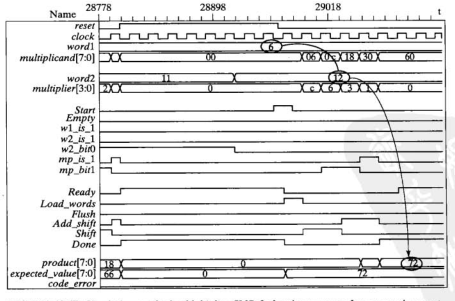


FIGURE 10-47 Simulation results for *Multiplier_IMP_2*, showing recovery from a running reset.

To gain insight into Booth's algorithm, note that the decimal value of a number in an n-bit 2s complement format can be gotten by (1) multiplying the leftmost bit by -2^{n-1}, (2) multiplying the remaining bits by 2^i, where i is the bit position, and (3) adding the results [12]. For example, the 2s complement representation of -7 is 1001_2. The decimal value, with $n = 4$, is obtained as shown in Figure 10-48.

The negative weight of the leftmost bit is expressed in signed digit notation [4] as an underscore; for example, the signed digit representation of -7 is given as $\underline{1}001$. Ordinarily, the bits of a binary number can have only positive weights, but in Booth's recoding algorithm, the bits of a number can have positive or negative weights expressed in signed digit notation.

The key to Booth's algorithm is that it skips over strings of 1s in the multiplier and replaces a series of additions by one addition and one subtraction. For example, the word 1111_0000 is equivalent to $2^8 - 1 - (2^4 - 1) = 2^8 - 2^4 = 256 - 16 = 240$. An arithmetic unit that must multiply negative numbers can exploit this relationship to possibly reduce the number of additions in multiplying two numbers. *A Booth recoding scheme recodes the multiplier by detecting strings of 1s and replacing them by signed digits that result in the same decimal value when the indicated addition and subtraction operations are performed.*

Table 10-1 summarizes the recoding rules. The algorithm reads bits from the LSB to the MSB, and the value of two successive bits (m_i, m_{i-1}) determines the Booth recoded multiplier bit, BRC_i. As the algorithm reads two successive bits, the present and the immediate past, it forms and uses BRC_i to determine whether to add or subtract before skipping to the next bit. The first step of the algorithm is seeded with a value of 0 to the right of the LSB of the word. If the signed digit $\underline{1}$ is encountered, a subtraction operation is performed (i.e., an appropriately shifted copy of the 2s complement of the multiplicand is added to the product). The process encodes the first encountered 1 as a $\underline{1}$, skips over any successive 1s until a 0 is encountered. That 0 is

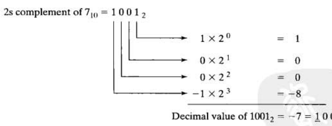


FIGURE 10-48 Extraction of the decimal value from a 2s complement number.

TABLE 10-1 Rules for Booth recoding of a 2s complement number.

m_i	m_{i-1}	BRC_i	Value	Status
0	0	0	0	String of 0s
0	1	1	+1	End of string of 1s
1	0	$\underline{1}$	−1	Begin string of 1s
1	1	0	0	Midstring of 1s

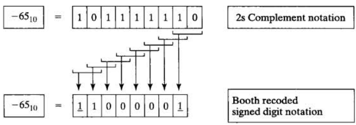


FIGURE 10-49 Booth recoding of -65_{10}.

encoded as a 1 to signify the end of a string of 1s, and then the process continues. The algorithm is valid for the entire range of 2s complement numbers (i.e., those with a 0 in the MSB, and those with a 1 in the MSB [4]).

As an example of Booth recoding, the encoding of $-65_{10} = 1011_1111_2$ is shown in Figure 10-49. Note that ordinary multiplication by this number would require seven additions, but the Booth-recoded multiplier requires only one addition and two subtractions.

An STG for the controller of a 4-bit Booth multiplier is shown in Figure 10-50. The annotation on the branches of the graph indicates that the Booth recoding bits (denoted by *BRC*) control the state transitions. The structural units for *Multiplier_Booth_STG_0* are shown in Figure 10-51, and the Verilog source code of *Multiplier_Booth_STG_0* is given in Example 10.8.

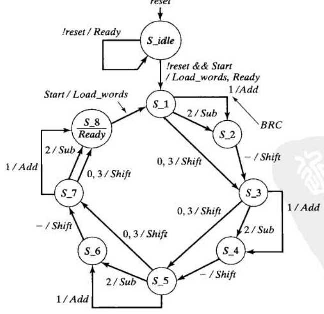


FIGURE 10-50 STG for a 4-bit Booth sequential multiplier.

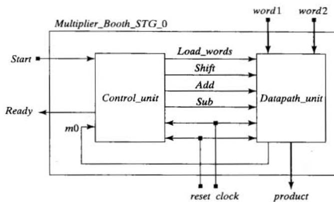


FIGURE 10-51 Structural units of a multiplier with Booth recoding.

The controller in *Multiplier_Booth_STG_0* generates the signals *Add* and *Sub* to control the addition and subtraction operations implied by the Booth algorithm. An alternative design could use one signal, *Add_sub*, to control these operations, but would need to generate and use a signal *Done* in the datapath unit to ensure that addition and subtraction operations are suspended while *Done* is asserted. Otherwise, the final value of *product* would be overwritten in state *S_8*. Note that the datapath unit uses a priority decoding scheme that decodes *Shift* before *Add* and *Sub*. The controller requires a flip-flop to store the LSB of *multiplier* for use in forming the Booth recoding bits (*BRC[1: 0]*), and the datapath operations require an adder/subtractor unit. Also, note how the 2s complement representation is formed when the multiplicand is negative. The left half of *multiplicand* must be filled with 1s to form the 2s complement correctly (sign extension). Similar actions are taken within the testbench to predict the expected value of *product*. For convenience, a pair of nested ***for*** loops generate integer values of *word1* and *word2*. Additional code converts the patterns to the wordlength of the parameterized machine. The testbench for *Multiplier_STG_0* is also presented because it contains some noteworthy features that are used when the pattern generator and comparator must form sign extensions of 2s complement negative numbers.

Example 10.8

```
module Multiplier_Booth_STG_0 #(parameter
  L_word = 4,
  L_BRC = 2,
  All_Ones = 4'b1111,
  All_Zeros = 4'b0000)(
  output        [2*L_word -1: 0]        product,
  output                                Ready,
  input         [L_word -1: 0]          word1, word2,
  input                                 Start, clock, reset
);
```

```
  wire                                            m0, Load_words, Shift, Add,
                                                    Sub, Ready;
  Controller_Booth_STG_0 M0_Controller
   (.Load_words(Load_words), .Shift(Shift), .Add(Add), .Sub(Sub), .Ready(Read),
    .m0(m0),
   .Start(Start), .clock(clock), .reset(reset));
  Datapath_Booth_STG_0 M1_Datapath
   (.product(product), .m0(m0), .word1(word1), .word2(word2),
    .Load_words(Load_words),
   .Shift(Shift), .Add(Add), .Sub(Sub), .clock(clock), .reset(reset));
endmodule

module Controller_Booth_STG_0 #(parameter L_word = 4, L_state = 4, L_BRC = 2)(
  output reg                          Load_words, Shift, Add, Sub,
  output                              Ready,
  input                               m0, Start, clock, reset
);
  reg        [L_state -1: 0]          state, next_state;
  parameter                           S_idle = 0, S_1 = 1, S_2 = 2, S_3 = 3,
                                      S_4 = 4, S_5 = 5, S_6 = 6, S_7 = 7, S_8 = 8;
  assign                              Ready = ((state == S_idle ) && !reset) ||
                                        (state == S_8);
  reg                                 m0_del;
  wire       [L_BRC -1: 0]            BRC = {m0, m0_del};       // Booth
                                                                  recoding bits
  // Necessary to reset m0_del when Load_words is asserted, otherwise it would
    start with residual value
  always @ (posedge clock, posedge reset)
   if (reset) m0_del <= 0; else if (Load_words) m0_del <= 0; else m0_del <= m0;
  always @ (posedge clock, posedge reset)
   if (reset) state <= S_idle; else state <= next_state;
  always @ (state, Start, BRC) begin          // Next state and control logic
   Load_words = 0; Shift = 0; Add = 0; Sub = 0;
  case (state)
   S_idle:if (Start)                               begin Load_words = 1;
                                                     next_state = S_1; end
         else                                      next_state = S_idle;
   S_1:  if ((BRC == 0) || (BRC == 3))             begin Shift = 1; next_state
                                                     = S_3; end
         else if (BRC == 1)                        begin Add = 1; next_state
                                                     = S_2; end
         else if (BRC == 2)                        begin Sub = 1; next_state
                                                     = S_2; end
   S_3:  if ((BRC == 0) || (BRC == 3))             begin Shift = 1; next_state
                                                     = S_5; end
         else if (BRC == 1)                        begin Add = 1; next_state
                                                     = S_4; end
         else if (BRC == 2)                        begin Sub = 1; next_state
                                                     = S_4; end
```

```
    S_5:  if ((BRC == 0) || (BRC == 3))      begin Shift = 1; next_state
                                              = S_7; end
          else if (BRC == 1)                 begin Add = 1; next_state
                                              = S_6; end
          else if (BRC == 2)                 begin Sub = 1; next_state
                                              = S_6; end
    S_7:  if ((BRC == 0) || (BRC == 3))      begin Shift = 1; next_state
                                              = S_8; end
          else if (BRC == 1)                 begin Add = 1; next_state
                                              = S_8; end
          else if (BRC == 2)                 begin Sub = 1; next_state
                                              = S_8; end
    S_2:                                     begin Shift = 1; next_state
                                              = S_3; end
    S_4:                                     begin Shift = 1; next_state
                                              = S_5; end
    S_6:                                     begin Shift = 1; next_state
                                              = S_7; end
    S_8:  if (Start)                         begin Load_words = 1;
                                              next_state = S_1; end
          else                               next_state = S_8;
    default:                                 next_state = S_idle;
   endcase
  end
endmodule

module Datapath_Booth_STG_0 #(parameter L_word = 4)(
  output reg    [2*L_word -1: 0]   product,
  output                           m0,
  input         [L_word -1: 0]     word1, word2,
  input                            Load_words, Shift, Add, Sub,
                                    clock, reset
);
  reg           [2*L_word -1: 0]   multiplicand;
  reg           [L_word -1: 0]     multiplier;
  assign                           m0 = multiplier[0];
  parameter                        All_Ones = 4'b1111;
  parameter                        All_Zeros = 4'b0000;

  // Register/Datapath Operations
  always @ (posedge clock, posedge reset) begin
   if (reset) begin multiplier <= 0; multiplicand <= 0; product <= 0; end
   else if (Load_words) begin
   if (word1[L_word -1] == 0) multiplicand <= word1;
   else multiplicand <= {All_Ones, word1[L_word -1: 0]};
   multiplier <= word2;
   product <= 0;
  end
  else if (Shift) begin
   multiplier <= multiplier >> 1;
   multiplicand <= multiplicand << 1;
```

```
    end
    else if (Add) begin product <= product + multiplicand; end
    else if (Sub) begin product <= product - multiplicand; end
  end
endmodule

module test_Multiplier_STG_0 ();
  parameter                                       L_word = 4;
  wire         [2*L_word -1: 0]                   product;
  wire                                            Ready;
  integer                                         word1, word2; // multiplicand,
                                                    multiplier
  reg                                             Start, clock, reset;
  reg          [3: 0]                             mag_1, mag_2;

  Multiplier_Booth_STG_0 M1 (.product(product), .Ready(Ready), .word1(word1),
  .word2(word2), .Start(Start), .clock(clock), .reset(reset));
  // Exhaustive Testbench
  reg          [2*L_word -1: 0]                   expected_value, expected_mag;
  reg                                             code_error;
  parameter                                       All_Ones = 4'b1111;
  parameter                                       All_Zeros = 4'b0000;
  initial #80000 $finish;                         // Timeout
 // Error detection
  always @ (posedge clock) // Compare product with expected value
    if (Start) begin
      expected_value = 0;
      case({word1[L_word -1], word2[L_word -1]})
        0: begin       expected_value = word1 * word2;
                       expected_mag = expected_value; end
        1: begin       expected_value = word1* {All_Ones,word2[L_word -1: 0]};
                       expected_mag = 1+ ~(expected_value); end
        2: begin       expected_value = {All_Ones,word1[L_word -1: 0]} *word2;
                       expected_mag = 1+ ~(expected_value); end
        3: begin       expected_value = ({All_Zeros, ~word2[L_word -1: 0]}+1)
                         * ({All_Zeros, ~word1[L_word -1: 0]}+1);
                       expected_mag = expected_value; end
      endcase
    code_error = 0;
    end
  else begin
    code_error = Ready ? |(expected_value ^ product) : 0;
  end
 initial begin clock = 0; forever #10 clock = ~clock; end
 initial begin
```

```
  #2 reset = 1;
  #15 reset = 0;
end
initial begin        // Exhaustive patterns
  #100
  for (word1 = All_Zeros; word1 <= 15; word1 = word1 +1) begin
    if (word1[L_word -1] == 0) mag_1 = word1;
    else begin mag_1 = word1[L_word -1: 0];
      mag_1 = 1+ ~mag_1; end
    for (word2 = All_Zeros; word2 <= 15; word2 = word2 +1) begin
      if (word2[L_word -1] == 0) mag_2 = word2;
      else begin mag_2 = word2[L_word -1: 0]; mag_2 = 1+ ~mag_2; end
      Start = 0; #40 Start = 1;
      #20 Start = 0;
      #200;
    end // word2
    #140;
    end // word1
  end
endmodule
```

End of Example 10.8

Figure 10-52 shows the simulation results produced by *Multiplier_STG_0* and *Multiplier_ Booth_STG_0* in multiplying $7_{10} \times 7_{10}$. For these data, the multiplier forms *product* in five cycles using Booth recoding, and in seven cycles without recoding. The waveforms of *product* are shown in decimal format (as 2s complement values), and the waveforms of *expected_value* and *expected_mag* are also in decimal format. The computational efficiency of Booth recoding is more significant for wide datapaths.

The implementation of Booth's algorithm based on the STG in Figure 10-50 is straightforward, but it is limited to a 4-bit multiplier. A block diagram and ASMD chart for an efficient and more flexible controller are shown in Figure 10-53. The machine accommodates a parameterized wordlength, and is efficient because it does not waste time doing needless operations, such as multiplying by 0 or multiplying after the last 1 in the multiplier has been found. We will present the corresponding model here because it reveals a subtlety that is not apparent in the STG-based model. When the value of the multiplier register is 1_{10}, the machine's action depends on whether this last bit of 1 is possibly due to *word2* having been the 2s complement code corresponding to a negative number (i.e., the MSB of *word2* was 1). Moreover, when *m_is_1* is asserted, there are only two possible values of *BRC*: 2_{10} and 3_{10}. In the former case, the usual subtraction must be performed: *Sub* is asserted and the

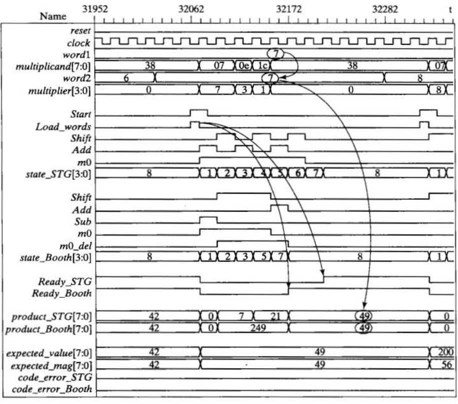


FIGURE 10-52 Comparison of state transitions in a multiplier with and without a Booth recoding scheme.

state moves from *S_running* to *S_shift1*, where *BRC* is now 1_2. If *word2* was negative no further action is needed; otherwise, a final addition must be executed. *Shift* is asserted in *S_shift1* to align the multiplicand for the final addition, then the state moves to *S_shift2*, where *Add* is asserted. The latter case must be handled differently, because the condition that *BRC* is 11_2 in *S_running* with *m_is_1* asserted could be due to an intermediate string of 1s or to a terminating string of 1s in *word2*. A terminating string of 1s corresponds to a negative 2s complement multiplier and dictates that *Add* not be asserted in *S_shift2*. There is no way to distinguish between these two cases without setting a flag in the datapath to indicate that *word2* is negative and passing the result as a status signal to the controller. The machine uses a

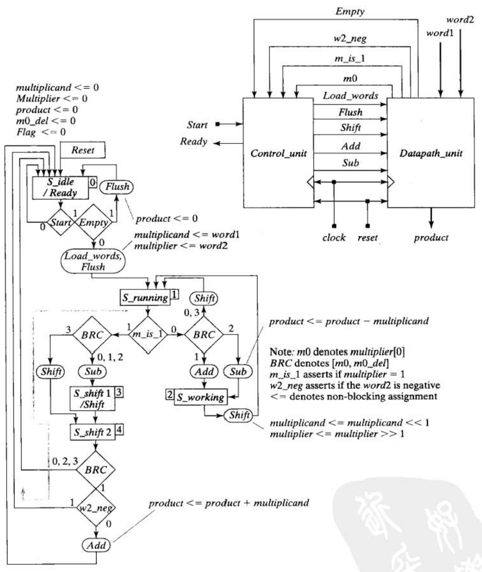
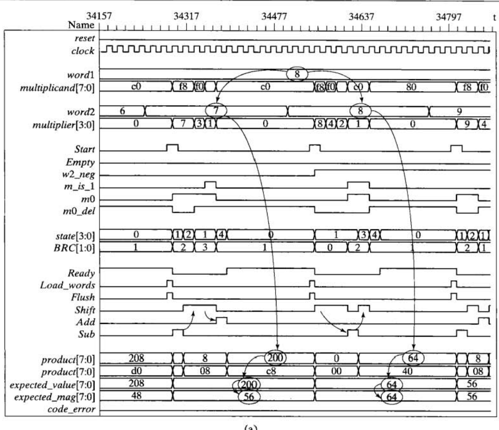


FIGURE 10-53 Block diagram and ASMD chart for a wordsize-flexible Booth sequential multiplier.

flag register in the datapath to form an additional status signal, *w2_neg*, to indicate that the data pattern of *word2* was for a negative value, e.g, *word2* = 1110_2 in the case of a 4-bit multiplier. The sequence of states for the special case is illustrated in Figure 10-53 by the highlighted path.



(a)

FIGURE 10-54 Simulation results for a wordsize-flexible Booth sequential multiplier with (a) multiplicand = −8 and multiplier = +7 and −8 and (b) multiplicand = −5 and multiplier = 0 and 1.

Figure 10-54 shows the product of -8 (encoded in 4-bit, 2s complement as $8_{10} = 1000_2$) by $+7_{10}$ and again by -8_{10} to produce products of 200 (magnitude is 56) and 64, respectively. Figure 10-55 shows detection of *word2* having a value of 0, and correct multiplication of -5_{10} (encoded in 4-bit, 2s complement as $11_{10} = 1011_2$) by 1_{10} to give a product of -5_{10}.

(b)

FIGURE 10-54 Continued

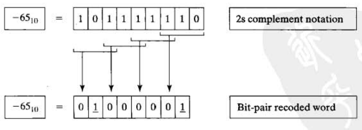


FIGURE 10-55 Bit-pair (radix-4) recoding of -65_{10}.

Example 10.9

```
module Multiplier_Booth_ASMD #(parameter L_word = 4)(
 output [2*L_word - 1: 0]  product,
 output                    Ready,
 input  [L_word - 1: 0]    word1, word2,
 input                     Start, clock, reset
);
 wire Empty, w2_neg, m_is_1, m0, Flush, Load_words, Shift, Add, Sub;

 Control_Unit M0_Controller (.Load_words(Load_words), .Flush(Flush), .Shift(Shift),
   .Add(Add), .Sub(Sub), .Ready(Ready), .Empty(empty), .w2_neg(w2_neg),
   .m_is_1(m_is_1), .m0(m0), .Start(Start), .clock(clock), .reset(reset));
 Datapath_Unit M1_Datapath (.product(product), .Empty(Empty), .w2_neg(w2_neg),
   .m_is_1(m_is_1), .m0(m0), .word1(word1), .word2(word2),
   .Load_words(Load_words), .Flush(Flush), .Shift(Shift), .Add(Add), .Sub(Sub),
   .clock(clock), .reset(reset));

endmodule

module Control_Unit #(parameter L_word = 4, L_state = 3, L_BRC = 2)(
 output reg      Load_words, Flush, Shift, Add, Sub,
 output          Ready,
 input           Empty, w2_neg, m_is_1, m0, Start, clock, reset
);
 parameter       S_idle = 0, S_running = 1, S_working = 2, S_shift1 = 3, S_shift2
                  = 4;

 reg   [L_state - 1: 0]   state, next_state;
 reg                      m0_del;
 wire  [L_BRC - 1: 0]     BRC = {m0, m0_del};        // Booth recoding bits
 assign Ready = (state == S_idle) || (state == S_idle);

// Necessary to reset m0_del when Load_words is asserted, otherwise it would
  start with residual value

 always @ (posedge clock, posedge reset)
  if (reset) m0_del <= 0; else if (Load_words) m0_del <= 0; else if (Shift) m0_del
   <= m0;

 always @ (posedge clock, posedge reset)
  if (reset) state <= S_idle; else state <= next_state;

 always @ (state, Start, BRC, Empty, w2_neg, m_is_1, m0) begin   // Next state
                                                                   and control
                                                                   logic

  Load_words = 0; Flush =0; Shift = 0; Add = 0; Sub = 0;
  next_state = S_idle;
  case (state)

   S_idle:        if (!Start) next_state = S_idle;
                  else if (Empty) begin Flush = 1; next_state = S_idle; end
                  else begin Load_words = 1; Flush = 1; next_state = S_running; end
```

```
    S_running:    if (m_is_1) begin
                   if (BRC == 3) begin Shift = 1; next_state = S_shift2; end
                   else begin Sub = 1; next_state = S_shift1; end  // Only BRC =
                                                                     2 is possible
                  end
                  else begin
                   if (BRC == 1) begin Add = 1; next_state = S_working; end
                   else if (BRC == 2) begin Sub = 1; next_state = S_working; end
                   else begin Shift = 1; next_state = S_running; end
                  end
    S_shift1:     begin Shift = 1; next_state = S_shift2; end
    S_shift2:     begin
                   next_state = S_idle;
                   if ((BRC == 1) && (!w2_neg)) Add = 1;
                  end
    S_working:    begin Shift = 1; next_state = S_running; end
    default:      next_state = S_idle;
  endcase
  end
endmodule
module Datapath_Unit #(parameter L_word = 4)(
 output reg      [2*L_word - 1: 0]   product,
 output Empty, w2_neg, m_is_1, m0,
 input   [L_word - 1: 0]    word1, word2,
 input   Load_words, Flush, Shift, Add, Sub, clock, reset
);
 reg     [2*L_word - 1: 0]  multiplicand;
 reg     [L_word - 1: 0]    multiplier;
 reg                               Flag;
 assign Empty = ((word1 == 0) || (word2 == 0));
 assign w2_neg = Flag;
 assign m_is_1 = (multiplier == 1);
 assign m0 = multiplier[0];
 parameter       All_Ones = {L_word{1'b1}};
 parameter       All_Zeros = {L_word{1'b0}};
// Register/Datapath Operations
 always @ (posedge clock, posedge reset)
  if (reset) begin multiplier <= 0; multiplicand <= 0; product <= 0; Flag <= 0; end
  else begin
   if (Load_words) begin
    Flag = word2[L_word -1];
    if (word1[L_word - 1] == 0) multiplicand <= word1;
    else multiplicand <= {All_Ones, word1[L_word -1: 0]};
     multiplier <= word2;
   end // Load_words
   if (Flush) product <= 0;
```

```
      if (Shift) begin
        multiplier <= multiplier >> 1; multiplicand <= multiplicand << 1;
      end
      if (Add) begin product <= product + multiplicand; end
      if (Sub) begin product <= product - multiplicand; end
    end
  endmodule
```

End of Example 10.9

10.3.12 Bit-Pair Encoding

Booth recoding does not always lead to a reduction in the clock cycles required for multiplication in multipliers whose STG has been modified to perform the operation of shifting in the same cycle as *Add_sub*. Depending on the data pattern, Booth recoding may actually increase the number of clock cycles! Thus, the efficiency of the Booth recoding algorithm depends on the data. An alternative scheme, called bit-pair encoding (BPE), overcomes this limitation by encoding the digits as signed radix-4 digits (also called bit-pair encoding) [4, 10]. BPE (recoding) ensures that the number of additions does not increase. In fact, the number of additions is reduced from n to $n/2$.

BPE of a multiplier examines 3 bits at a time to determine whether to (1) add the multiplicand, (2) shift the multiplicand by 1 bit and then add, (3) subtract the multiplicand (i.e., add the 2s complement of the multiplicand to the product), (4) shift the 2s complement of the multiplicand to the left by 1 bit and then add, or (5) to only shift the multiplicand to the location corresponding to the next bit-pair (i.e., without adding or subtracting at the present location). As in Booth recoding, the first step of the BPE algorithm is seeded with a value of 0 in a register cell to the right of the LSB of the multiplier word. Subsequent actions depend on the value of the recoded bit-pair. The multiplier bit index i increments by 2 until the word is exhausted. If the multiplier word contains an odd number of bits, its sign bit must be extended by 1 bit to accommodate the recoding scheme. Recoding divides the multiplier word by 2, so the number of possible additions is reduced by a factor of 2. The rules for BPE are summarized in Table 10-2.

TABLE 10-2 Rules for bit-pair (radix-4) recoding of a 2s complement number.

m_{i+1}	m_i	m_{i-1}	Code	BRC_{i+1}	BRC_i	Value	Status	Actions
0	0	0	0	0	0	0	String of 0s	Shift by 2
0	0	1	1	0	1	+1	End of string of 1s	Add
0	1	0	2	0	1	+1	Single 1	Add
0	1	1	3	1	0	+2	End of string of 1s	Shift by 1, Add, Shift by 1
1	0	0	4	$\underline{1}$	0	−2	Begin string of 1s	Shift by 1, Subtract, Shift by 1
1	0	1	5	0	$\underline{1}$	−1	Single 0	Subtract
1	1	0	6	0	$\underline{1}$	−1	Begin string of 1s	Subtract
1	1	1	7	0	0	0	Midstring of 1s	Shift by 2

Example 10.10

The bit-pair recoding of $-65_{10} = 1011_1111_2$ is shown in Figure 10-55(e).

End of Example 10.10

Example 10.11

The 2s complement product of 5_{10} by the multiplier -65_{10}, with -65_{10} recoded in a bit-pair format, is illustrated in Figure 10-56. The first bit-pair (shaded) indicates subtraction, so the 2s complement of 5_{10} is formed and aligned with the LSB of the multiplicand. Double shifts result from the next two bit-pairs. The final bit-pair (shaded) specifies subtraction, so the 2s complement of 5_{10} is formed at the proper location. Taking the sum of the shifted multiplicands forms the 2s complement of the product. The magnitude of the result is also shown. Note that it is necessary to sign-extend the copies of the multiplicand to fit the word length of the product register.

End of Example 10.11

Example 10.12

Figure 10-57 shows the 2s complement product of -128_{10} multiplied by the multiplier -128_{10}, with -128_{10} recoded in a bit-pair format. The first three bit-pairs of the multiplier

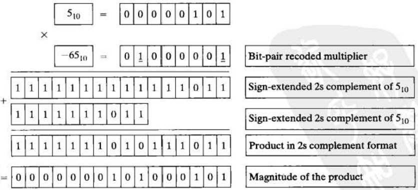


FIGURE 10-56 Multiplication of 5_{10} by bit-pair (radix-4) recoding of -65_{10}.

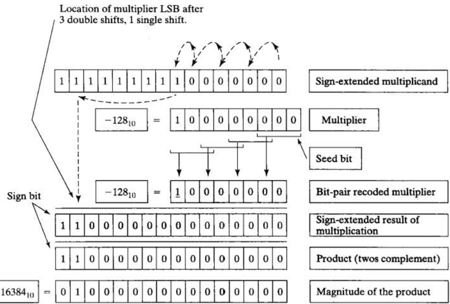


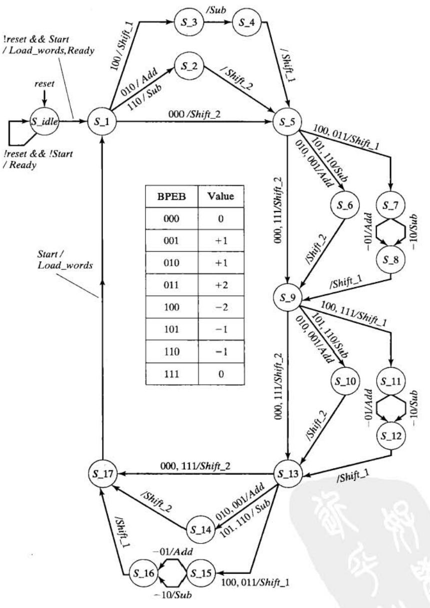
FIGURE 10-57 Multiplication of -128_{10} by bit-pair (radix-4) recoding of -128_{10}.

(beginning at the LSB) cause a copy of the multiplicand to be shifted by 6 bits toward the MSB; the final bit-pair signifies the beginning and end of a string of 1s, so the multiplicand is shifted by 1 bit and added to the product register. The magnitude of the result is also shown. Note that it is necessary for the product register to have a length (16 bits) of twice the word length of the data (8 bits), and for the multiplicand to be sign-extended to fit the word length of the product register.

End of Example 10.12

Example 10.13

The Verilog description of an 8-bit radix-4 multiplier having the STG of the control unit shown in Figure 10-58, with separate cycles for shifting, adding, and subtracting. The recoding rules are also shown. The functionality of the multiplier was verified



BPEB	Value
000	0
001	+1
010	+1
011	+2
100	−2
101	−1
110	−1
111	0

FIGURE 10-58 STG for multiplication with bit-pair recoding of 8-bit multipliers.

exhaustively for all combinations of positive and negative data words. The controller handles the states necessary for BPE, and the datapath is expanded to handle single and double bit shifts.

```
module Multiplier_Radix_4_STG_0 #(parameter L_word = 8) (
 output          [2*L_word -1: 0]  product,
 output                            Ready,
 input           [L_word -1: 0]    word1, word2,
 input                             Start, clock, reset
);
 wire                              Load_words, Shift_1, Shift_2, Add, Sub;
 wire            [2: 0]            BPEB;

 Controller_Radix_4_STG_0 M0
   (.Load_words(Load_words), .Shift_1(Shift_1), .Shift_2(Shift_2), .Add(Add),
   .Sub(Sub), .Ready(Ready), .BPEB(BPEB), .Start(Start), .clock(clock),
   .reset(reset));

 Datapath_Radix_4_STG_0 M1
   (.product(product), .BPEB(BPEB), .word1(word1), .word2(word2),
   .Load_words(Load_words), .Shift_1(Shift_1), .Shift_2(Shift_2), .Add(Add),
   .Sub(Sub), .clock(clock), .reset(reset));

endmodule

module Controller_Radix_4_STG_0 #(parameter L_word = 8)(
 output reg                 Load_words, Shift_1, Shift_2, Add, Sub,
 output                     Ready,
 input           [2: 0]     BPEB,
 input                      Start, clock, reset
);
 reg             [4: 0]     state, next_state;
 parameter                  S_idle = 0, S_1 = 1, S_2 = 2, S_3 = 3;
 parameter                  S_4 = 4, S_5 = 5, S_6 = 6, S_7 = 7, S_8 = 8;
 parameter                  S_9 = 9, S_10 = 10, S_11 = 11, S_12 = 12;
 parameter                  S_13 = 13, S_14 = 14, S_15 = 15;
 parameter                  S_16 = 16, S_17 = 17;
 assign                     Ready = ((state == S_idle) && !reset) ||
                              (next_state == S_17);

 always @ (posedge clock, posedge reset)
   if (reset) state <= S_idle; else state <= next_state;

 always @ (state, Start, BPEB) begin            // Next state and control logic
```

```
Load_words = 0; Shift_1 = 0; Shift_2 = 0; Add = 0; Sub = 0;
case (state)
 S_idle:    if (Start) begin Load_words = 1;     next_state = S_1; end
            else                                 next_state = S_idle;
 S_1:       case (BPEB)
              0:      begin Shift_2 = 1;         next_state = S_5; end
              2:      begin Add = 1;             next_state = S_2; end
              4:      begin Shift_1 = 1;         next_state = S_3; end
              6:      begin Sub = 1;             next_state = S_2; end
              default:                           next_state = S_idle;
            endcase
 S_2:       begin     Shift_2 = 1;               next_state = S_5; end
 S_3:       begin     Sub = 1;                   next_state = S_4; end
 S_4:       begin     Shift_1 = 1;               next_state = S_5; end
 S_5:       case (BPEB)
              0, 7:   begin Shift_2 = 1;         next_state = S_9; end
              1, 2:   begin Add = 1;             next_state = S_6; end
              3, 4:   begin Shift_1 = 1;         next_state = S_7; end
              5, 6:   begin Sub = 1;             next_state = S_6; end
            endcase
 S_6:       begin     Shift_2 = 1;               next_state = S_9; end
 S_7:       begin     if (BPEB[1: 0] == 2'b01) Add = 1;
                      else Sub = 1;              next_state = S_8; end
 S_8:       begin     Shift_1 = 1;               next_state = S_9; end
 S_9:       case (BPEB)
              0, 7:   begin Shift_2 = 1;         next_state = S_13; end
              1, 2:   begin Add = 1;             next_state = S_10; end
              3, 4:   begin Shift_1 = 1;         next_state = S_11; end
              5, 6:   begin Sub = 1;             next_state = S_10; end
            endcase
 S_10:      begin     Shift_2 = 1;               next_state = S_13; end
 S_11:      begin     if (BPEB[1: 0] == 2'b01) Add = 1;
                      else Sub = 1;              next_state = S_12; end
 S_12:      begin     Shift_1 = 1;               next_state = S_13; end
 S_13:      case (BPEB)
              0, 7:   begin Shift_2 = 1;         next_state = S_17; end
              1, 2:   begin Add = 1;             next_state = S_14; end
              3, 4:   begin Shift_1 = 1;         next_state = S_15; end
              5, 6:   begin Sub = 1;             next_state = S_14; end
            endcase
 S_14:      begin     Shift_2 = 1;               next_state = S_17; end
 S_15:      begin     if (BPEB[1: 0] == 2'b01) Add = 1;
            else Sub = 1;                        next_state = S_16; end
 S_16:      begin     Shift_1 = 1;               next_state = S_17; end
 S_17:      if        (Start) begin Load_words = 1; next_state = S_1; end
```

```
                            else                next_state = S_17;
    default:                                    next_state = S_idle;
   endcase
  end
endmodule

module Datapath_Radix_4_STG_0 #(parameter L_word = 8)(
  output reg        [2*L_word -1: 0]    product,
  output            [2: 0]              BPEB,
  input             [L_word -1: 0]      word1, word2,
  input                                 Load_words, Shift_1, Shift_2,
                                        Add, Sub, clock, reset
);
  reg               [2*L_word -1: 0]    multiplicand;
  reg               [L_word -1: 0]      multiplier;
  reg                                   m0_del;
  parameter                             All_Ones = {L_word{1'b1}};

  assign                                BPEB = {multiplier[1: 0], m0_del};

 // Register/Datapath Operations
 always @ (posedge clock, posedge reset)
   if (reset) begin
     multiplier <= 0; m0_del <= 0; multiplicand <= 0; product <= 0;
   end
   else begin
     if (Load_words) begin
      m0_del <= 0;
      if (word1[L_word -1] == 0) multiplicand <= word1;
      else multiplicand <= {'All_Ones, word1[L_word -1: 0]};
      multiplier <= word2; product <= 0;
     end
     if (Shift_1) begin
      {multiplier, m0_del} <= {multiplier, m0_del} >> 1;
      multiplicand <= multiplicand << 1;
     end
     if (Shift_2) begin
      {multiplier, m0_del} <= {multiplier, m0_del} >> 2;
      multiplicand <= multiplicand << 2;
     end
     if (Add) begin product <= product + multiplicand; end
     if (Sub) begin product <= product - multiplicand; end
    end
endmodule
```

Figure 10-59 shows output waveforms in hex and decimal formats verifying the multiplication of -45_{10} by -38_{10}. The values of *word1* and *word2* are also shown in hex and decimal formats (*mag_1* and *mag_2*); the magnitudes of *word1* and *word2* are shown in decimal format. The value of *multiplicand* and the *multiplier* are shown in hex and decimal formats verifying the multiplication of -45_{10} by -38_{10}. The values of *word1* and *word2* are also shown in hex and decimal formats (*mag_1* and *mag_2*); the magnitudes of *word1* and *word2* are shown in decimal format. The value of *multiplicand* and the *multiplier* are shown in hex format. The value of *product* is shown

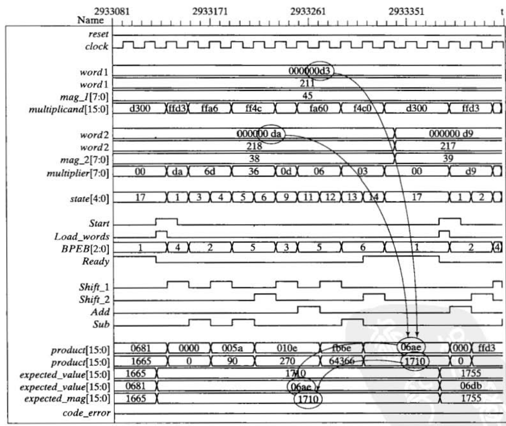


FIGURE 10-59 Simulation of *Multiplier_Radix_4_STG_0* multiplying -45_{10} by -38_{10}.

in hex and decimal formats. The expected value (produced by the testbench) is shown in both formats.

End of Example 10.13

10.4 Multiplication of Signed Binary Numbers

Although signed binary numbers in 2s complement format can be multiplied with Booth's algorithm, we will reconsider their multiplication here, to prepare for multiplication of fractions. There are four cases to consider in multiplying signed numbers in 2s complement format, depending on the signs of the multiplicand and multiplier. We have already seen that the product of unsigned binary numbers is formed by adding shifted copies of the multiplicand. We will consider the remaining three cases, in which one or both words are negative.

10.4.1 Product of Signed Numbers: Negative Multiplicand, Positive Multiplier

The steps to multiply a negative multiplicand by a positive multiplier are the same as the steps taken to multiply unsigned numbers, but the sign bit of the multiplicand must be extended to the word length of the final product before operating on the 2s complement words. The sign-extended multiplicand is used when forming the partial products and accumulating the sums. The result of the multiplication is the 2s complement of the product. Then the magnitude of the product is formed by taking the 2s complement of the result, as illustrated in Figure 10-60 for the product of -3_{10} by 6_{10}. The sign-extended multiplicands are shown, with the carry bits that are generated in each column-wise addition.

10.4.2 Product of Signed Numbers: Positive Multiplicand, Negative Multiplier

To form the product of a positive multiplicand by a negative multiplier, extend the sign of multiplier to the word length of the multiplier. Then add shifted copies of the multiplicand, but instead of adding a copy of the multiplicand at the position corresponding to the extended sign bit of the multiplier, add the 2s complement of the multiplicand. This last step follows from the observation (see Figure 10-4(b)) that the decimal value of the 2s complement of an n-bit multiplier can be written as the sum: $-B_{n-1}2^{n-1} + B_{n-2}2^{n-2} + \cdots + B_1 2^1 + B_0 2^0$. The actions associated with $B_{n-2}, \ldots, B_1, B_0$ are the usual ones of adding shifted copies of the multiplicand. The action of the term associated with $-B_{n-1} \times 2^{n-1}$ is equivalent to adding a shifted copy of the 2s complement of the multiplicand to the sum of the previously accumulated partial sums. The results in Figure 10-61 form the product of 3_{10} by -6_{10}.

10.4.3 Product of Signed Numbers: Negative Multiplicand, Negative Multiplier

When the multiplicand and the multiplier are both negative numbers expressed in 2s complement format, the last sum adds the 2s complement of the multiplicand to the

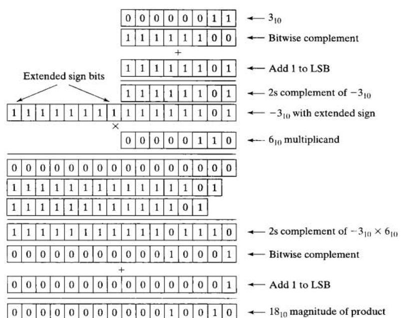


FIGURE 10-60 Multiplication of a negative multiplicand (-3_{10}), in a 2s complement, sign-extended format, by a positive multiplier (6_{10}), forming the product -18_{10}.

accumulated partial sums, but the accumulated sums are formed with shifted copies of the 2s complement sign-extended multiplicand, instead of the multiplicand. For clarity, Figure 10-62 also shows the columnwise carries that are generated in forming the product of -3_{10} by -6_{10}.

10.5 Multiplication of Fractions

Numbers are normalized in digital signal processors to avoid the overflow that would result when the product of two numbers exceeds the dynamic range provided by the word length of the machine [12]. The dynamic range of the decimal value of the numbers that can be represented by N bits in a 2s complement format is $-2^{N-1} \leq D(B) \leq 2^{N-1} - 1$. For example, 2s complement 4-bit words have $N = 4$, and the dynamic range of their decimal value is from -8 to $+7$. Any product whose value exceeds this range causes overflow, because the product cannot be stored accurately as a 4-bit value. For example, the product of 7 by 3 exceeds the dynamic range provided by a 4-bit word format.

Normalization divides an N-bit 2s complement word by 2^{N-1} to convert a fixed-point integer representation of a value to a fixed-point fractional representation. The dynamic range of the magnitude of the fractional value is bounded between -1 and $+1$. Normalization is equivalent to shifting the word toward its LSB by $N - 1$ positions, and

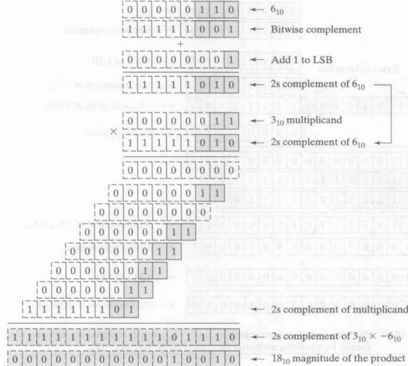


FIGURE 10-61 Multiplication of a positive multiplicand (3_{10}) by a multiplier (-6_{10}), in a 2s complement, sign-extended multiplier, forming the product -18_{10}.

associating its weights with fractions. If a 2s complement word B has the decimal value $D(B) = -b_{N-1}2^{N-1} + b_{N-2}2^{N-2} \cdots + b_1 2^1 + b_0 2^0$, its normalized value is given by $F(B) = -b_{N-1}2^0 + b_{N-2}2^{-1} + \cdots + b_1 2^{-(N-2)} + b_0 2^{-(N-1)}$, the so-called Q-format representation of the number [12]. For example, a Q-5 number format has 5 bits, including a sign bit. The product of two Q-5 numbers has 10 bits, including an extended sign bit and a sign bit. The radix point of Q-format numbers is to the right of the sign bit.

Normalization of integers prevents overflow in multiplication, because the product of two fractions is always a fraction. It also extends the dynamic range of the numbers that can be multiplied, for a given word length, at the expense of precision. The result of multiplying two normalized numbers may have less precision than if the numbers could be multiplied by a machine that has sufficient word length to avoid overflow. For example, the product of $8_{10} = 1000_2$ by $7_{10} = 0011_2$ is 56_{10}, which cannot be stored as a 4-bit value in 2s complement format. Normalization produces the following fractions in Q-5 format: $F(8_{10}) = 2^{-4} = 0.1000_2$, and $F(7_{10}) = 0.0111_2$. Their product is 00.0011_1000_2, in Q-10 format. Storing the product as a Q-5 value gives $F(8_{10} \times 7_{10}) = 0.0011$. The denormalized decimal value is $F(8_{10} \times 7_{10}) \cong 48_{10}$, obtained by scaling the Q-5 value by 2^8, or left-shifting the word by 8 bits.

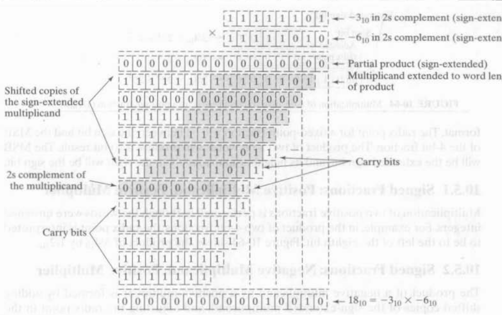


FIGURE 10-62 Multiplication of a negative multiplicand (-3_{10}) by a negative multiplier (-6_{10}), forming the product 18_{10}.

The 2s complement of an n-bit number M is a number M^* such that $M + M^* = 2^n$. The 2s complement of a binary fraction M is given by $M^* = 2 - M$, so that $M + M^* = 2$. An m-bit fraction F is represented as $F = b_{-1}2^{-1} + b_{-2}2^{-2} + b_{-3}2^{-3} + \cdots + b_{-m}2^{-m}$. The 2s complement of a fraction is formed by complementing the bits from the sign bit to the least significant 1, then adding 1 at the position of the least significant 1. This is equivalent to complementing the bits to the left of the rightmost 1 in the word. Both methods are shown in Figure 10-63.

Fractions are multiplied like whole numbers, but overflow is not possible, because the product of two fractions must be a fraction. Care must be taken to adjust the location of the radix point of the result of multiplying fractions. In a fixed-point format, a 4-bit fraction is represented by 5 bits, with the MSB holding the sign of the number in a 2s complement

0 1 1 0 0 ← $3/4_{10} = 2^{-1} + 2^{-2}$
1 0 0 0 0 ← Bitwise complement
0 0 1 ← Add 1
1 0 1 0 0 ← $-3/4_{10} = 2^{0} + 2^{-2}$
Sign bit
Radix point

0 1 1 0 0 ← $3/4_{10} = 2^{-1} + 2^{-2}$
1 0 1 0 0 ← Bitwise complement
1 0 1 0 0 ← $-3/4_{10} = 2^{0} + 2^{-2}$

FIGURE 10-63 Equivalent methods of the forming 2s complement of a fraction ($3/4_{10}$).

Extended sign bit
Sign bit
Adjusted radix point

	0.1100	← $3/4_{10} = 2^{-1} + 2^{-2}$
×	0.1000	← $1/2_{10} = 2^{-1}$
	00.01100000	← $3/8_{10} = 2^{-2} + 2^{-3}$

FIGURE 10-64 Multiplication of a positive fraction ($3/4_{10}$) by a positive fraction ($1/2_{10}$).

format. The radix point for a fixed-point fraction will be between the sign bit and the MSB of the 4-bit fraction. The product of two 5-bit words will produce a 10-bit result. The MSB will be the extended sign bit, and its neighbor immediately to the right will be the sign bit.

10.5.1 Signed Fractions: Positive Multiplicand, Positive Multiplier

Multiplication of two positive fractions is performed as though the words were unsigned integers. For example, in the product of two 4-bit fractions, the radix point is interpreted to be to the left of the eighth bit. Figure 10-64 shows the product of $3/4_{10}$ by $1/2_{10}$.

10.5.2 Signed Fractions: Negative Multiplicand, Positive Multiplier

The product of a negative multiplicand by a positive multiplier is formed by adding shifted copies of the sign-extended multiplicand, and adjusting the radix point in the result. The example in Figure 10-65 forms the product of $-3/4_{10}$ by $3/8_{10}$.

10.5.3 Signed Fractions: Positive Multiplicand, Negative Multiplier

If the multiplicand is positive and the multiplier is negative, we add shifted copies of the sign-extended multiplicand, except at the position of the sign bit of the multiplier. At that position, we add the 2s complement of the multiplicand. In this case, it is convenient to define the 2s complement of a fraction A as $A^* = 2 - A$, or $10_2 - A_2$. This limits the representation of the multiplicand to have only 2 bits to the left of the radix,

	0.1100	← $3/4_{10} = 2^{-1} + 2^{-2}$
	1.0100	← $-3/4_{10} = 2^{0} + 2^{-2}$
	111111.0100	← Sign-extended multiplicand
×	0.0110	← $3/8_{10} = 2^{-2} + 2^{-3}$
	1111101000	← Shifted, extended multiplicand
	11110100	← Shifted, extended multiplicand
	1110111000	← $-9/32_{10}$ = in 2s complement
	11.01001000	← $9/31_{10} = 2^{-2} + 2^{-5}$

Adjusted radix point

FIGURE 10-65 Multiplication of a negative fraction ($3/4_{10}$) by a positive fraction ($3/8_{10}$).

0.1 1 0 0 ← $3/4_{10} = 2^{-1} + 2^{-2}$
× 1.1 0 1 0 ← $-3/8_{10}$
0 1 1 0 0 0
0 1 1 0 0
1 0 1 0 0 ← 2s complement of multiplicand
1 1 1 0 1 1 1 0 0 0 ← $-9/32_{10}$ = in 2s complement
0 0 1 0 0 1 0 0 0 0 ← $9/31_{10} = 2^{-2} + 2^{-5}$ = magnitude of $(3/4)_{10} \times (-3/8)_{10}$

Adjusted radix point

FIGURE 10-66 Multiplication of a positive multiplicand ($3/4_{10}$) by a negative multiplier ($-3/8_{10}$).

thereby reducing the number of products that must be added to form the sum, as shown in Figure 10-66.

10.5.4 Signed Fractions: Negative Multiplicand, Negative Multiplier

To form the product of two negative fractions, add shifted copies of the sign-extended multiplicand, and add the 2s complement of the accumulated sum, as shown in Figure 10.67.

10.6 Functional Units for Division

Sequential multipliers use an *add-and-shift* algorithm to form the product of two words. We will consider various architectures for sequential dividers that use a *subtract-and-shift* algorithm to form a quotient of two numbers.

0.1 1 0 0 ← $3/4_{10} = 2^{-1} + 2^{-2}$
1.0 1 0 0 ← $-3/4_{10} = 2^{0} + 2^{-2}$
1 1 1 1 1 1.0 1 0 0 ← Sign-extended multiplicand
× 1.1 0 1 0 ← $-3/8_{10}$
1 1 1 1 1 0 1 0 0 0 ← Shifted, extended multiplicand
1 1 1 0 1 0 0 ← Shifted, extended multiplicand
0 0 1 1 0 0 ← 2s complement of multiplicand
0 0 0 1 0 0 1 0 0 0 ← $9/31_{10} = 2^{-2} + 2^{-5} = (-3/4)_{10} \times (-3/8)_{10}$

Adjusted radix point

FIGURE 10-67 Multiplication of a negative multiplicand ($-3/4_{10}$) by a negative multiplier ($-3/8_{10}$).

10.6.1 Division of Unsigned Binary Numbers

A sequential algorithm for dividing two unsigned binary numbers (i.e., positive integers) subtracts the divisor from the dividend repeatedly, until the remainder is detected to be smaller than the divisor. The quotient is formed by incrementing a counter each time a subtraction occurs; the final value of the remainder is formed as the residual content of the dividend when the subtraction sequence ends. Other architectures, such as one implementing a subtract-and-shift algorithm, can be more efficient, but we will examine the basic architecture first.

Example 10.14

Figure 10-68 shows the architecture for *Divider_STG_0*, a machine that forms the quotient of unsigned binary numbers by repeatedly subtracting the content of a divisor register from the content of a dividend register until the remainder is less than the divisor. This architecture is effective, but inefficient. It uses more registers than needed, and it can require a very long execution sequence to form the quotient when the divisor is small compared to the dividend.

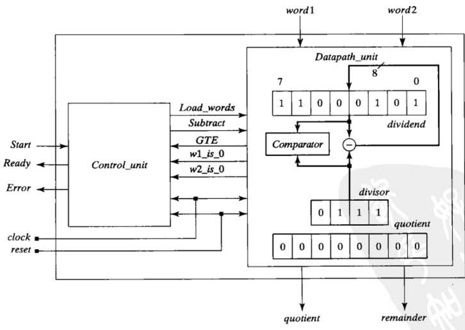


FIGURE 10-68 Architecture of *Divider_STG_0*, a simple, but inefficient, binary divider unit.

Divider_STG_0 will serve to introduce some features that will be included in a more sophisticated machine. Among the machine's features, we want it to detect an attempt to divide by 0, and to terminate without needless computation if the datapath presents a dividend that is 0. The machine should ignore *Start* while a division sequence is in progress. A signal *Ready* should be asserted after a division sequence is completed, and remain asserted until a new sequence begins; *Ready* should also be asserted while the machine is in its idle state, with *reset* not asserted. A signal *Error* should be asserted if a divide-by-zero is attempted, and should remain asserted until *reset* is asserted. An asynchronous reset signal should drive the state to its idle state from any state.

Divider_STG_0 has parameterized word lengths, shown here for a dividend datapath size of 8 bits, and a divisor datapath size of 4 bits. The implementation assumes that the length of the divisor does not exceed the length of the dividend. In Figure 10-69, *dividend* and *quotient* are stored in 8-bit registers and *divisor* is stored in a 4-bit register. To implement the subtraction of *divisor* from *dividend*, the word for *divisor* will be converted to 2s complement value, and extended by concatenating the 4-bit 2s complement with four 1s. A comparator determines whether or not to subtract. For an 8-bit dividend, the worst case will take 255 subtraction steps (*dividend* = 255, *divisor* = 1). The machine's STG is shown in Figure 10.69, with its control logic annotated with symbols for Verilog operators. Unlike an ASMD chart, the STG is not annotated with the register operations of the datapath unit. State transitions along the branches leaving a state are conditioned by the indicated assertions, provided that *reset* is not asserted.

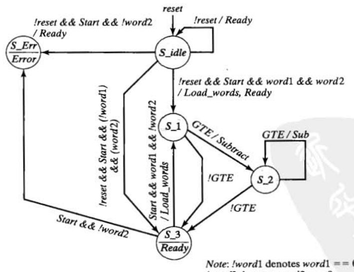


FIGURE 10-69 State-transition diagram of controller for *Divider_STG_0*.

The state of the controller enters *S_idle* on the asynchronous action of *reset* and remains there until *Start* is asserted (with *reset* de-asserted). If *word2* (the datapath value for *divisor*) is 0, the state enters an error state, *S_Err,* when *Start* is asserted, and remains there until *reset* is re-asserted. The signal *Error* is asserted as a Moore-type output in the state *S_Err* (the machine also enters *S_Err* as a fail-safe feature if its state is not one of those specified by the STG, but that detail is not shown on the STG). If *word2* is not 0, *word1* (the datapath value of *dividend*) is checked.[11] If *word1* is 0, the state immediately transfers to *S_3*, where *Ready* is asserted as a Moore-type output. If not, *Load_words* is asserted and the machine enters *S_1*, where *Subtract* is asserted while successive subtractions occur. In state *S_idle, Load_words* and *Ready* are Mealy-type outputs.

At each step, the algorithm compares *divisor* and *dividend*. When *dividend* is found to be less than *divisor*, the state enters *S_3*, where it remains until the next assertion of *Start* and *Ready* is asserted. It is the responsibility of the external agent controlling the machine to know that *Start* will be ignored until the machine is in *S_idle* or *S_3*. So *Start* should not be asserted until after *Ready* is asserted. Otherwise, the contents of *quotient* and *remainder* could be mistakenly associated with the new values of *word1* and *word2*, rather than the values that were present when the division that led to the assertion of *Ready* was initiated.

For the architecture shown in Figure 10-68, three status signals are generated in the datapath unit: *GTE*, *w1_is_0*, and *w2_is_0* to indicate whether the dividend is larger than the divisor, the datapath word that would be loaded in the dividend is 0, and the datapath word that would be loaded into the divisor is 0. These signals are used to implement the logic that skips division if the word that would be loaded into the dividend is 0, and sends the state to *S_Err* and aborts an attempt to divide by 0.

End of Example 10.14

Example 10.15

The Verilog description of *Divider_STG_0* is given below for an 8-bit dividend and a 4-bit divisor. We show two ways to implement subtraction. The first corresponds to the actual hardware supporting the machine's datapath operation of subtraction of 2s complement words and requires sign extension of *divisor*. An alternative uses the statement *dividend <= dividend − divisor*. This form exploits the built-in 2s complement arithmetic of Verilog and automatically accommodates the different word lengths of the operands, leaving the actual hardware up to a synthesis tool. The testbench (see the Web

[11]In Verilog, the Boolean value of *word2* is true if and only if *word2* has the value of a positive integer.

site) for *Divider_STG_0* has a triggered stimulus generator. Note that the number of cycles required to form *quotient* is data-dependent, so stimulus patterns are triggered by the completion of a division sequence. The machine is synthesizable, because the data dependency is handled by the controller, not by a data-dependent loop.

```
module Divider_STG_0 #(parameter L_divn = 8, L_divr = 4)(
  output [L_divn -1: 0]      quotient,
  output [L_divn -1: 0]      remainder,
  output Ready, Error,
  input  [L_divn -1: 0]      word1,   // Datapath for dividend
  input  [L_divr -1: 0]      word2,   // Datapath for divisor
  input  Start, clock, reset
);

/* Includes checks for a divide by zero, subtracts the divisor from the dividend until the
dividend is less than the divisor, and counts the number of subtractions performed.
The length of divisor must not exceed the length of dividend .
*/

  Control_Unit M0_Controller
   (.Ready(Ready), .Error(Error), .Load_words(Load_words),
    .Subtract(Subtract), .Start(Start), .GTE(GTE), .w1_is_0(w1_is_0), .w2_is_0(w2_is_0),
    .clock(clock), .reset(reset));

  Datapath_Unit M1_Datapath
  (.quotient(quotient), .remainder(remainder), .GTE(GTE), .w1_is_0(w1_is_0),
    .w2_is_0(w2_is_0), .word1(word1), .word2(word2), .Load_words(Load_words),
    .Subtract(Subtract), .clock(clock), .reset(reset)
  );
endmodule

module Control_Unit (output Ready, Error, output reg Load_words, Subtract,
  input Start, GTE, w1_is_0, w2_is_0, clock, reset
);
  parameter                 S_idle = 0, S_1 = 1, S_2 = 2, S_3 = 3, S_Err = 4;
  parameter                 L_state = 3;
  reg      [L_state -1: 0]  state, next_state;

  assign Ready = ((state == S_idle) && !reset) || (state == S_3);
  assign Error = (state == S_Err);

  always @ (posedge clock, posedge reset)
   if (reset) state <= S_idle; else state <= next_state;

  always @ (state, Start, GTE, w1_is_0, w2_is_0) begin
   Load_words = 0; Subtract = 0; next_state = S_Err; // Default values
   case (state)
    S_idle:       case (Start)
                   0:        next_state = S_idle;
                   1:        if (w2_is_0) next_state = S_Err;
```

```
                    else if (!w1_is_0) begin next_state = S_1; Load_words
                     = 1; end
                    else next_state = S_3;
                endcase
    S_1:        if (GTE) begin next_state = S_2; Subtract = 1; end
                else next_state = S_3;

    S_2:        if (GTE) begin next_state = S_2; Subtract = 1; end
                else next_state = S_3;

    S_3:        case (Start)
                 0:     next_state = S_3;
                 1:     if (w2_is_0) next_state = S_Err;
                        else if (w1_is_0) next_state = S_3;
                        else begin next_state = S_1; Load_words = 1; end
                endcase
    S_Err:      next_state = S_Err;
    default:    next_state = S_Err;
   endcase
  end
endmodule

module Datapath_Unit #(parameter L_divn = 8, L_divr = 4)(
 output reg       [L_divn -1: 0]    quotient,
 output           [L_divn -1: 0]    remainder,
 output                             GTE, w1_is_0, w2_is_0,
 input  [L_divn -1: 0]    word1,    // Datapath for dividend
 input  [L_divr -1: 0]    word2,    // Datapath for divisor
 input                    Load_words, Subtract, clock, reset
);
 reg    [L_divn -1: 0]    dividend;
 reg    [L_divr -1: 0]    divisor;

 assign GTE = (dividend >= divisor);        // Comparator
 assign w1_is_0 = (word1 == 0);
 assign w2_is_0 = (word2 == 0);
 assign remainder = dividend;

 always @(posedge clock, posedge reset) begin   // Register/Datapath Operations
  if (reset) begin divisor <= 0; dividend <= 0; quotient <= 0; end
  else if (Load_words == 1) begin
   dividend <= word1;
   divisor <= word2;
   quotient <= 0;
  end
  else if (Subtract) begin                  // Note sign extension below
   dividend <= dividend[L_divn -1: 0] + 1'b1 + {{(L_divn -L_divr){1'b1}},
    ~divisor[L_divr -1: 0]};
  // dividend <= dividend - divisor;         // alternative using built-in
                                              2's complement arithmetic
```

```
      quotient <= quotient + 1;
    end                                        // Use quotient +2 to test error detection
  end
endmodule
```

End of Example 10.15

Figure 10-70 shows waveforms obtained by simulating *Divider_STG_0* dividing 100_{10} by 13_{10} to produce a quotient of 8_{10} and a remainder of 3_{10}. The values of *word1, word2, dividend, divisor, quotient*, and *remainder* are shown in decimal format. The signal *code_error* is generated by the testbench on detecting an error in either *quotient* or *remainder*. These waveforms do not demonstrate all of the features of the design; the testbench provided at the companion Web site can be used for additional verification, including recovery from an attempted divide by 0.

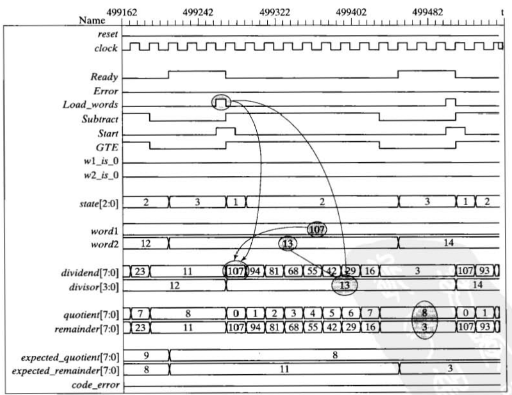


FIGURE 10-70 Simulation results for *Divider_STG_0*, a simple binary divider.

Although a Verilog behavioral description can be written without concern for the details of implementation, leaving them up to a synthesis tool, it might be wise to consider the fact that *a synthesis tool can fail to produce the most efficient implementation, by not recognizing economies in the architecture.* For example, the architecture of *Divider_STG_0* uses a comparator (see Figure 10-68) to determine whether *divisor* should be subtracted from *dividend*, and uses a subtractor to perform the subtraction. An alternative design would exploit the observation that in 2s complement subtraction the carry bit reveals the relative magnitude of the numbers, eliminating the need for a comparator. The subtractor is implemented by an adder with a carry-in, and an inverted datapath for *divisor* (bitwise-complement). The carry-out of the adder produces the sign bit that controls the datapath. An architecture for the alternative machine, *Divider_STG_0_sub*, is shown in Figure 10-71, and the machine's Verilog description is presented in Example 10.16. Note that a single continuous assignment forms the concatenation $\{carry, difference\}$ by adding the 2s complement of *divisor* to *dividend*.

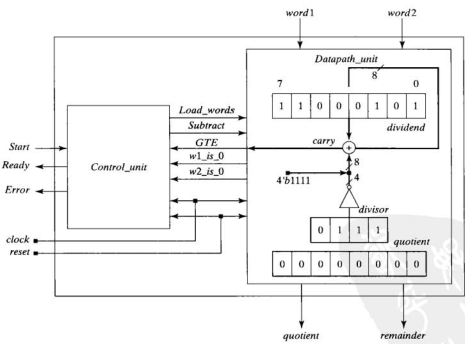


FIGURE 10-71 Architecture for *Divider_STG_0_sub*, a modified architecture of a simple, but inefficient, binary divider unit with an 8-bit dividend. The carry bit formed in 2s complement subtraction replaces the comparator used by *Divider_STG_0*.

Example 10.16

The Verilog description of *Divider_STG_0_sub*[12] uses the carry bit from 2s complement subtraction to replace a comparator and control the datapath of the machine. The control unit from *Divider_STG_0* is re-used; the datapath unit is changed slightly as shown below. It has additional code to form the concatenation $\{carry, difference\}$. The signal *carry* provides *GTE* at the interface between the control unit and the datapath unit. Simulation results (not shown) for *Divider_STG_0* and *Divider_STG_0_sub* *match*.

```
module Datapath_Unit #(parameter L_divn = 8, L_divr = 4)(
  output reg     [L_divn -1: 0]      quotient,
  output         [L_divn -1: 0]      remainder,
  output                             carry, w1_is_0, w2_is_0,
  input  [L_divn -1: 0]   word1,  // Datapath for dividend
  input  [L_divr -1: 0]   word2,  // Datapath for divisor
  input                   Load_words, Subtract, clock, reset
);
  reg    [L_divn -1: 0]   dividend;
  reg    [L_divr -1: 0]   divisor;
  wire   [L_divn -1: 0]   difference;

  assign {carry, difference} = dividend[L_divn-1: 0] + {{(L_divn -L_divr){1'b1}},
    ~divisor[L_divr -1: 0]} + 1'b1;
  assign w1_is_0 = (word1 == 0);
  assign w2_is_0 = (word2 == 0);
  assign remainder = dividend;

  always @(posedge clock, posedge reset) begin   // Register/Datapath Operations

   if (reset) begin divisor <= 0; dividend <= 0; quotient <= 0; end
   else if (Load_words == 1) begin
    dividend <= word1;
    divisor <= word2;
    quotient <= 0; end
   else if (Subtract) begin                   // Note sign extension below
    dividend <= dividend[L_divn -1: 0] + 1'b1 + {{(L_divn -L_divr){1'b1}},
      ~divisor[L_divr -1: 0]};
    // dividend <= dividend - divisor;          // alternative using built-in
                                                  2's complement arithmetic
    quotient <= quotient + 1; end               // Use quotient +2 to test error detection
  end
endmodule
```

End of Example 10.16

[12]The complete file and testbench for *Divider_STG_0*_sub are provided at www.pearsonhighered.com/ciletti.

10.6.2 Efficient Division of Unsigned Binary Numbers

The machines in the previous section divide unsigned binary numbers by repeatedly subtracting the divisor from the dividend. Both machines are very inefficient when dividing by a relatively small divisor, because they must perform several subtractions. A basic architecture for a more efficient divider is shown in Figure 10-72. Its operations parallel the commonly used manual steps that divide two numbers by (1) aligning the divisor with the MSB of the dividend, (2) then repeatedly subtracting the divisor from the dividend, and (3) shifting the *divisor* toward the LSB of the dividend. However, in the hardware implementation, the contents of the *dividend* register will be shifted repeatedly *toward the MSB* of the divisor.

Care must be taken in designing the architecture. In the dividers of the previous section, the registers holding *divisor* and *dividend* are physically aligned, so their LSBs are aligned too. At any stage of subtraction in the next architecture, it might be necessary to align the divisor and the dividend, depending on their relative size and on the relative location of each word's most significant 1 bit. Also, the dividend register must be extended to the left by 1 bit to accommodate the possibility that the initial (decimal) value of the aligned content of the divisor register exceeds the (decimal) value of the

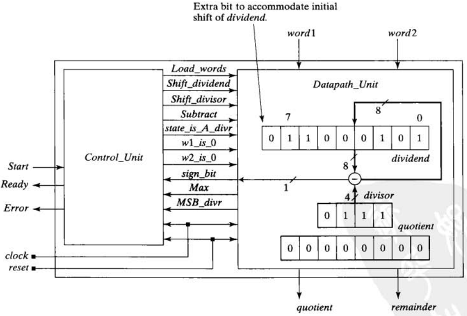


FIGURE 10-72 Architecture for *Divider_STG_1*, a self-aligning divider for unsigned binary words, with an 8-bit dividend and a 4-bit divisor.

associated 4 bits of the dividend register, in which case we must shift a 1 from the MSB of the dividend before subtraction can be performed. For example, to divide 1001_2 by 1010_2, we must first shift the dividend to the left to align the dividend for the next subtraction. Consequently, the controller for the machine will be more complex, and includes signals for shifting both the divisor and the dividend, as shown in Figure 10-72.

The physical architecture of the machine aligns the divisor word with the leftmost four bits of the dividend's 8-bit datapath. In operation, the dividend word is shifted repeatedly from right to left, and the divisor word is subtracted from the dividend bits that it is aligned with at each step, depending on whether the divisor is less than the corresponding part-select of the dividend. However, instead of subtracting the divisor from the dividend, the machine is aligned to subtract the largest possible product of the divisor and a power of 2, thereby eliminating repeated subtractions when the divisor is relatively small.

The machine is said to be *self-aligning*, because it automatically determines whether *divisor* or *dividend* need to be aligned at the beginning of a division sequence, depending on the relative position of their leftmost nonzero bits. An approach that would always *initially* align both words so that their MSB contains a 1 is inefficient because it can require far more shifts than are needed. The approach we will take is to initially shift the divisor toward the leftmost nonzero bit of the dividend (instead of the LSB of the dividend).

There are two cases that require an initial alignment of the datapath words: (1) the value of the leftmost 4-bit subword of *dividend* is less than the value of *divisor* (e.g., 1100_2 divided by 1110_2) and (2) the LSB of divisor is 0 and the divisor word can be shifted to the left and still divide into *dividend* (e.g., 1100_2 divided by 0101_2). In the first case, *dividend* must be shifted repeatedly to the left by 1 bit until the value of the leftmost 5 bits of the 1-bit-extended *dividend* equals or exceeds that of *divisor*, or until no further shifts are allowed; in the second case, *divisor* must be shifted to the left until the word produced by a further shifting cannot divide into the leftmost 4 bits of the *dividend* word (excluding the extra bit). The physical location of the remainder bits at the end of a division sequence depends on whether the dividend has been shifted for alignment. Therefore, the alignment shifts are counted and used to control the state machine and adjust the value of the remainder at the end of the execution sequence.

The STG for a self-aligning divider, *Divider_STG_1*, is shown in Figure 10-73. At a given state, a control label that is used on a branch leaving a state node will be treated as de-asserted on any other exiting branch where it is not used explicitly. A label that does not appear on any branch leaving a state node will be considered to be a don't-care. The reset signal is shown only at state *S_idle*, but is understood to have asynchronous action at all of the other states too.

In state *S_Adivr* the action of *Shift_divisor* aligns the divisor with the most significant 1-bit of the dividend; in state *S_Adivn* the action of *Shift_dividend* aligns the dividend register for subtraction; and in *S_div* the actual subtraction occurs, together with more shift operations. In states *S_Adivn* and *S_Adivr*, the variable *Max* detects when the maximum allowed shifts have occurred.

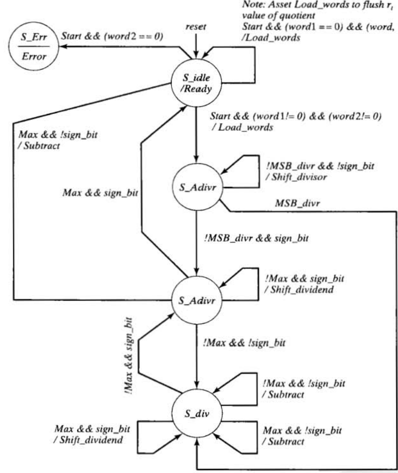


FIGURE 10-73 STG for *Divider_STG_1*, a self-aligning divider.

Example 10.17

The testbench and Verilog description of a self-aligning divider, *Divider_STG_1*, corresponding to the STG in Figure 10-73 are given below for an 8-bit dividend and a 4-bit divisor. Note two features of *Divider_STG_1*: (1) the sign bit produced by the subtractor controls the datapath (an alternative design would rely on a synthesis tool to possibly replace the logic of a comparator with the sign bit of the subtractor),

and (2) the datapath to the subtractor is multiplexed. This feature eliminates the need for a separate comparator to implement the test to determine whether the result of shifting *divisor* by 1 bit will divide into the 4-bit sub-word of *dividend*. In this case, the value of *comparison* depends on whether *divisor* has been shifted before the difference between *dividend* and *divisor* is formed. The sign bit of *comparison* determines *sign_bit*. Alignment of *dividend* with *divisor* occurs in *S_Adivn*. The datapath operations of subtracting and shifting occur in state *S_div*.[13] The test patterns generated by the testbench are triggered by a de-assertion of *Ready*. This feature was included, because the number of cycles needed to complete a division sequence is data-dependent. A fixed-cycle pattern generator would have to accommodate the worst-case sequence for division, and use that delay for all patterns, making the test much longer than necessary. The testbench developed for this model includes error detection for exhaustive verification.

```
module Divider_STG_1 #(parameter L_divn = 8, L_divr = 4)( // Choose L_divr <=
  L_divn
 output         [L_divn -1: 0]      quotient,
 output         [L_divn -1: 0]      remainder,
 output         Ready, Error,
 input          [L_divn -1: 0]      word1,    // Datapath for dividend
 input          [L_divr -1: 0]      word2,    // Datapath for divisor
 input          Start, clock, reset
);
 wire    Load_words, Shift_dividend, Shift_divisor, Subtract, state_is_S_Adivr;
 wire    w1_is_0, w2_is_0, sign_bit, Max, MSB_divr;

 Control_Unit M0_Controller (
   .Ready(Ready), .Error(Error), .Load_words(Load_words),
   .Shift_dividend(Shift_dividend), .Shift_divisor(Shift_divisor),
   .Subtract(Subtract), .state_is_S_Adivr(state_is_Adivr),
   .Start(Start), .w1_is_0(w1_is_0), .w2_is_0(w2_is_0), .sign_bit(sign_bit),
   .Max(Max), .MSB_divr(MSB_divr), .clock(clock), .reset(reset)
 );
 Datapath_Unit M1_Datapath (
   .quotient(quotient), .remainder(remainder), .w1_is_0(w1_is_0), .w2_is_0(w2_is_0),
   .sign_bit(sign_bit), .Max(Max), .MSB_divr(MSB_divr),
   .word1(word1), // Datapath for dividend
   .word2(word2), // Datapath for divisor
   .Load_words(Load_words), .Shift_dividend(Shift_dividend),
   .Shift_divisor(Shift_divisor), .Subtract(Subtract), .state_is_S_Adivr(state_is_Adivr),
   .clock(clock), .reset(reset)
 );
endmodule
```

[13]See Problem 28 at the end of the chapter for a modification to *Divider_STG_1* that reduces the cycles needed for division.

```
module Control_Unit (output Ready, Error,
  output reg Load_words, Shift_dividend, Shift_divisor, Subtract,
  output state_is_S_Adivr,
  input Start, w1_is_0, w2_is_0, sign_bit, Max, MSB_divr, clock, reset
);
parameter S_idle = 0, S_Adivr = 1, S_Adivn = 2, S_div = 3, S_Err = 4, L_state = 3;
  reg    [L_state -1: 0]    state, next_state;
  assign Ready =((state == S_idle) && !reset);
  assign Error = (state == S_Err);
  assign state_is_S_Adivr = (state == S_Adivr);

  always @ (posedge clock, posedge reset)
   if (reset) state <= S_idle; else state <= next_state;

  always @ (state, Start, w1_is_0, w2_is_0, sign_bit, Max, MSB_divr) begin
   Load_words = 0; Shift_dividend = 0; Shift_divisor = 0; Subtract = 0; next_state
    = S_idle;

   case (state)
    S_idle:       case (Start)
                   0:      next_state = S_idle;
                   1:      if (w2_is_0) next_state = S_Err;
                           else if (!w1_is_0) begin next_state = S_Adivr;
                             Load_words = 1; end
                           else next_state = S_idle;
                  endcase

    S_Adivr:      case (MSB_divr)
                   0: if (!sign_bit) begin next_state = S_Adivr; Shift_divisor = 1;
                     end // can shift divisor
                     else if (sign_bit) begin next_state = S_Adivn; end  // cannot
                                                                            shift
                                                                            divisor
                   1: next_state = S_div;
                  endcase

    S_Adivn:      case ({Max, sign_bit})
                   2'b00:  next_state = S_div;
                   2'b01:  begin next_state = S_Adivn; Shift_dividend = 1; end
                   2'b10:  begin next_state = S_idle; Subtract = 1; end
                   2'b11:  next_state = S_idle;
                  endcase

    S_div:        case ({Max, sign_bit})
                   2'b00:  begin next_state = S_div; Subtract = 1; end
                   2'b01:  next_state = S_Adivn;
                   2'b10:  begin next_state = S_div; Subtract = 1; end
                   2'b11:  begin next_state = S_div; Shift_dividend = 1; end
                  endcase
```

```
    default:       next_state = S_Err;
   endcase
  end
endmodule

module Datapath_Unit #(parameter L_divn = 8, L_divr = 4, L_cnt = 4)(
// Choose L_divr <= L_divn
  output reg [L_divn -1: 0] quotient,
  output [L_divn -1: 0]     remainder,
  output                    w1_is_0, w2_is_0, sign_bit, Max, MSB_divr,
  input   [L_divn -1: 0]             word1,   // Datapath for dividend
  input   [L_divr -1: 0]             word2,   // Datapath for divisor
  input                              Load_words, Shift_dividend, Shift_divisor,
                                       Subtract, state_is_S_Adivr,
  input                              clock, reset
);
  parameter Max_cnt = L_divn-L_divr;
  reg     [L_divn: 0]        dividend;          // Extended dividend
  reg     [L_divr -1: 0]     divisor;
  reg     [L_cnt -1: 0]      num_shift_dividend, num_shift_divisor;
  // Logic for status signals
  assign MSB_divr = divisor[L_divr -1];
  assign w1_is_0 = !(|word1);
  assign w2_is_0 = !(|word2);
  assign Max = (num_shift_dividend == Max_cnt + num_shift_divisor );
  // Shift the remainder to compensate for alignment shifts:
  assign remainder = (dividend[L_divn -1: L_divn -L_divr] ) >> num_shift_divisor;

  wire [L_divr: 0] comparison = ((MSB_divr == 0) && (state_is_S_Adivr))?
                   dividend[L_divn: L_divn - L_divr] + {1'b1, ~(divisor << 1)} + 1'b1:
                   dividend[L_divn: L_divn - L_divr] + {1'b1, ~divisor[L_divr -1: 0]}
                    + 1'b1;

  assign sign_bit = comparison[L_divr];

  always @ (posedge clock, posedge reset)          // Register/Datapath operations
    if (reset) begin
      divisor <= 0; dividend <= 0; quotient <= 0; num_shift_dividend <= 0;
        num_shift_divisor <= 0;
    end
   else begin
    if (Load_words) begin
      dividend <= word1;
      divisor <= word2;
      quotient <= 0;
      num_shift_dividend <= 0;
      num_shift_divisor <= 0;
    end
    if (Shift_divisor) begin
```

```
      divisor <= divisor << 1;
      num_shift_divisor <= num_shift_divisor + 1;
    end
    if (Shift_dividend) begin
      dividend <= dividend << 1;
      quotient <= quotient << 1;
      num_shift_dividend <= num_shift_dividend +1;
    end
    if (Subtract) begin
      dividend [L_divn: L_divn -L_divr] <= comparison;
      quotient[0] <= 1;
    end
  end
endmodule

module test_Divider_STG_1 ();
  parameter L_divn = 8;
  parameter L_divr = 4;
  parameter word_1_max = 255;
  parameter word_1_min = 1;
  parameter word_2_max = 15;
  parameter word_2_min = 1;
  parameter max_time = 850000;
  parameter half_cycle = 10;
  parameter start_duration = 20;
  parameter start_offset = 30;
  parameter delay_for_exhaustive_patterns = 490;
  parameter reset_offset = 50;
  parameter reset_toggle = 5;
  parameter reset_duration = 20;

  parameter word_2_delay = 20;
  wire [L_divn -1: 0] quotient;
  wire [L_divn-1: 0] remainder;
  wire Ready, Div_zero;
  integer word1;  // dividend
  integer word2;  // divisor
  reg Start, clock, reset;
  reg [L_divn-1: 0] expected_quotient;
  reg [L_divn-1: 0] expected_remainder;
  wire quotient_error, rem_error;
  integer k, m;
  // probes
  wire [L_divr-1: 0] Left_bits = M0.M1.dividend[L_divn-1: L_divn -L_divr];
```

```
Divider_STG_1 M0 (
  .quotient(quotient), .remainder(remainder), .Ready(Ready), .Error(Error),
  .word1(word1), .word2(word2), .Start(Start), .clock(clock), .reset(reset));

initial #max_time $finish;
initial begin clock = 0; forever #half_cycle clock = ~clock; end
initial begin expected_quotient = 0; expected_remainder
 forever @ (negedge Ready) begin        // Form expected values
  #2 if (word2 != 0) begin expected_quotient = word1 / word2; expected_remainder
    = word1 % word2; end
 end
end

 assign quotient_error = (!reset && Ready) ? |(expected_quotient ^ quotient): 0;
 assign rem_error = (!reset && Ready) ? |(expected_remainder ^ remainder): 0;

 initial begin    // Test for divide by zero detection
  #2 reset = 1;
  #15 reset = 0; Start = 0;
  #10 Start = 1; #5 Start = 0;
 end

 initial begin    // Test for recovery from error state on reset and running reset
  #reset_offset reset = 1; #reset_toggle Start = 1; #reset_toggle reset = 0;
  word1 = 0;
  word2 = 1;
  while (word2 <= word_2_max) #20 word2 = word2 +1;
  #start_duration Start = 0;
 end

 initial begin    // Exhaustive patterns
  #delay_for_exhaustive_patterns
  word1 = word_1_min; while (word1 <= word_1_max) begin
  word2 = 1; while (word2 <= 15) begin
   #0 Start = 0;
   #start_offset Start = 1;
   #start_duration Start = 0;
   @ (posedge Ready) #0;
   word2 = word2 + 1; end              // divisor pattern
   word1 = word1 + 1; end              // dividend pattern
 end
endmodule
```

End of Example 10.17

Figure 10-74 presents simulation results for *Divider_STG_1*, illustrating the initial alignment of *dividend* by the action of *Shift_dividend*. Based on the STG in Figure 10-73, the machine correctly forms the quotient of $28_{10} = 0001_1011_2$ by $8_{10} = 1000_2$, giving a quotient of 3_{10} with a remainder of 4_{10}. When *Start* is asserted, the machine loads *dividend* and *divisor*, and moves to state *S_Adivr*, where it compares *divisor* and the leftmost byte of the dividend word to determine whether alignment is necessary. The machine detects a need for alignment of *dividend*, and with *sign_bit* asserted, moves at the next clock to *S_Adivn* to begin aligning *dividend*. With *shift_dividend* asserted for the next three clocks, *dividend* ($28_{10} = 0001_1100_2$) is shifted three positions to the left to

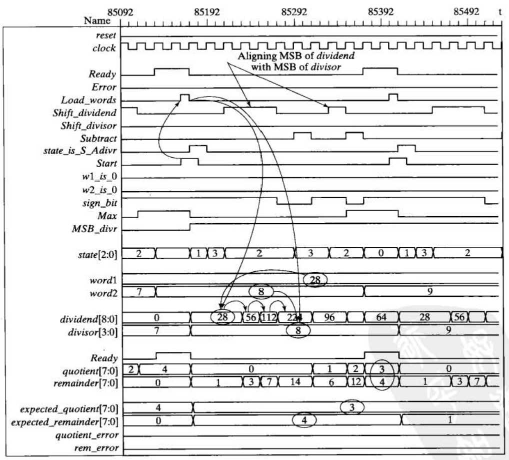


FIGURE 10-74 Simulation results for *Divider_STG_1*, dividing 28_{10} by 8_{10}, with initial alignment of *dividend* by the action of *Shift_dividend*.

align its MSB with the MSB of *divisor* ($8_{10} = 1000_2$), producing for *dividend* a value of $1110_0000_2 = 224_{10}$. The simulation results reveal a feature of the design: it wastes a clock cycle before and after aligning the dividend. Further modification of the machine is left to a problem at the end of this chapter.

Figure 10-75 shows division of $193_{10} = 1100_0001_2$ by $1_{10} = 0001_2$ and the action of *Shift_divisor* in aligning the MSB of *divisor* in three clock cycles. The waveforms have been annotated to show the cycles at which the shifting action occurs. Note that *Divider_STG_1* machine requires more communication between the controller and the datapath than *Divider_STG_0*, but *quotient* and *reminder* are formed in only 18 cycles instead of 193 cycles.

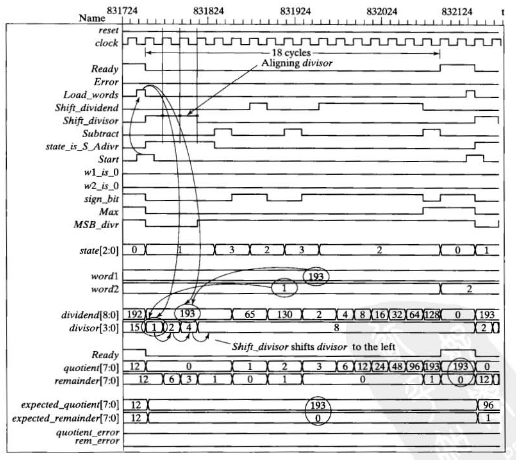


FIGURE 10-75 Simulation results showing division of 193_{10} by 2_{10} and the initial alignment of *divisor* with *dividend* by the action of *Shift_divisor*.

10.6.3 Reduced-Register Sequential Divider

A more efficient architecture for a divider exploits the fact that the contents of the dividend register are shifted toward its MSB as the division sequence unfolds, leaving room in the register for the bits of the quotient. This architecture is more efficient in its use of physical resources because it eliminates the need for a separate register to hold the quotient, as shown in Figure 10-76. The implementation has the following additional features: (1) shifting and subtracting occur in the same clock cycle instead of in separate cycles, (2) the remainder is adjusted to correct for its final location in the register, and (3) an overflow bit detects an invalid result.

The organization of the register shared by dividend and quotient is shown in Figure 10-77. The register includes a 1-bit extension to accommodate an initial shift that might be required to align *dividend* and *divisor*, and to hold the sign bit that is formed by subtracting *divisor* from *dividend*. The register is extended on the right by 1 bit to

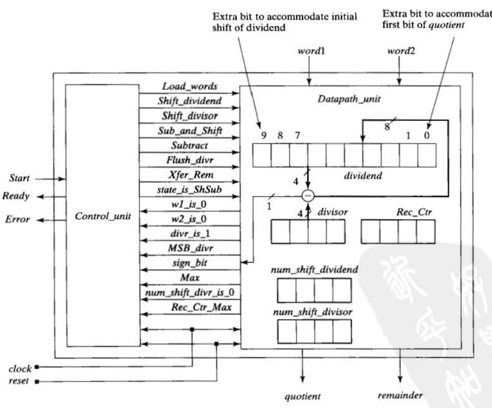


FIGURE 10-76 Architecture for *Divider_STG_RR*, a binary divider with reduced registers.

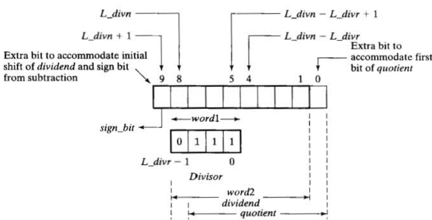


FIGURE 10-77 Register organization for *Divider_STG_RR*, a binary divider with reduced registers.

hold the first bit that is formed for *quotient*. Counters within the datapath unit are used to generate status signals sent to the control unit. Figure 10-78 shows the ASMD chart for the machine.[14]

Example 10.18

```
module Divider_RR_STG #(parameter L_divn = 8, L_divr = 4 )(
 // Choose L_divr <= L_divn
 output [L_divn -1: 0]          quotient,
 output [L_divr -1: 0]          remainder,
 output                 Ready, Error,
 input [L_divn -1: 0]           word1, // Datapath for dividend
 input [L_divr -1: 0]           word2, // Datapath for divisor
 input                          Start, clock, reset
);

 Control_Unit M0_Controller (
  .Ready(Ready), .Error(Error), .Load_words(Load_words),
  .Shift_dividend(Shift_dividend), .Shift_divisor, (Shift_divisor),
  .Sub_and_Shift(Sub_and_Shift), .Subtract(Subtract),
  .Flush_divr(Flush_divr), .Xfer_Rem(Xfer_Rem), .state_is_ShSub(state_is_ShSub),
  .Start(Start), w1_is_0(w1_is_0), .w2_is_0(w2_is_0), .divr_is_1(divr_is_1),
  .MSB_divr(MSB_divr), .sign_bit(sign_bit), .Max(Max),
```

[14]The logic forming the status signals within the datapath is not part of the ASMD chart.

```
    .num_shift_divr_is_0(num_shift_divr_is_0), .Rec_Ctr_Max(Rec_Ctr_Max),
    .clock(clock), .reset(reset));
  Datapath_Unit M1_Datapath (
    .quotient, .remainder, .w1_is_0(w1_is_0), .w2_is_0(w2_is_0), .divr_is_1(divr_is_1),
    .MSB_divr(MSB_divr), .sign_bit(sign_bit), .Max(Max),
    .num_shift_divr_is_0(num_shift_divr_is_0),
    .Rec_Ctr_Max(Rec_Ctr_Max), .word1(word1), .word2(word2),
    .Load_words(Load_words),
    .Shift_dividend(Shift_dividend), .Shift_divisor(Shift_divisor),
    .Sub_and_Shift(Sub_and_Shift),
    .Subtract(Subtract), .Flush_divr(Flush_divr), .Xfer_Rem(Xfer_Rem),
    .state_is_ShSub(state_is_ShSub), .clock(clock), .reset(reset));
endmodule

module Control_Unit (
  output          Ready, Error,
  output reg      Load_words, Shift_dividend, Shift_divisor, Sub_and_Shift, Subtract,
                  Flush_divr, Xfer_Rem,
output    state_is_ShSub,
input             Start, w1_is_0, w2_is_0, divr_is_1, MSB_divr, sign_bit, Max,
                  num_shift_divr_is_0, Rec_Ctr_Max, clock, reset
);
parameter L_state = 3;
parameter S_idle = 0, S_Adivr = 1, S_ShSub = 2, S_Rec = 3, S_Err = 4;
reg [L_state -1: 0] state, next_state;

  assign        Ready =((state == S_idle) && !reset) ;
  assign        Error = (state == S_Err);
  assign        state_is_ShSub = (state == S_ShSub);

  always @ (posedge clock, posedge reset) if (reset) state <= S_idle; else state
    <= next_state;

  always @ (state, Start, w1_is_0, w2_is_0, divr_is_1, MSB_divr, sign_bit, Max,
    num_shift_divr_is_0, Rec_Ctr_Max)
    begin
     Load_words = 0; Shift_dividend = 0; Shift_divisor = 0;
     Sub_and_Shift = 0; Subtract = 0; Flush_divr = 0; Xfer_Rem = 0;
     case (state)
      S_idle:      case (Start)
                     0:     next_state = S_idle;
                     1:     if (w2_is_0) next_state = S_Err;
                            else if (!w1_is_0) begin next_state = S_Adivr;
                              Load_words = 1; end
                            else if (sign_bit) next_state = S_ShSub;
                            else next_state = S_idle;
                    default: next_state = S_Err;
                   endcase
```

```
      S_Adivr:     if (divr_is_1) begin next_state = S_idle; end else
                     case ({MSB_divr, sign_bit})
                       2'b00: begin next_state = S_Adivr; Shift_divisor = 1; end // can
                                                                                  shift
                                                                                  divisor
                       2'b01:next_state = S_ShSub;                               // cannot
                                                                                  shift
                                                                                  divisor
                       2'b10:next_state = S_ShSub;
                       2'b11:next_state = S_ShSub;
                     endcase

      S_ShSub:     case ({Max, sign_bit})
                     2'b00:  begin next_state = S_ShSub; Sub_and_Shift = 1; end
                     2'b01:  begin next_state = S_ShSub; Shift_dividend = 1; end
                     2'b10:  begin Subtract = 1; if ( num_shift_divr_is_0) next_state
                               = S_idle;
                             else next_state = S_ShSub; end
                     2'b11:  if ( num_shift_divr_is_0) next_state = S_idle;
                             else begin next_state = S_Rec; Flush_divr = 1; end
                   endcase

      S_Rec:       if (Rec_Ctr_Max) begin next_state = S_idle; end
                   else  begin next_state = S_Rec; Xfer_Rem = 1; end

     default :     next_state = S_Err;
   endcase
  end
endmodule

module Datapath_Unit #( parameter L_divn = 8, L_divr = 4,
 L_Rec_Ctr = 3, L_cnt = 4, Max_cnt = L_divn - L_divr)(
 output [L_divn -1: 0]      quotient,
 output [L_divr -1: 0]      remainder,
 output                     w1_is_0, w2_is_0, divr_is_1, MSB_divr, sign_bit, Max,
 output                     num_shift_divr_is_0,
 output                     Rec_Ctr_Max,
 input   [L_divn -1: 0]     word1,            // Datapath for dividend
 input   [L_divr -1: 0]     word2,            // Datapath for divisor
 input                      Load_words, Shift_dividend, Shift_divisor,
                              Sub_and_Shift, Subtract,
                            Flush_divr, Xfer_Rem, state_is_ShSub, clock, reset
);
 reg     [L_divn +1: 0]     dividend;         // Doubly extended dividend
 reg     [L_divr -1: 0]     divisor;
 reg     [L_cnt -1: 0]      num_shift_dividend, num_shift_divisor;
 reg     [L_Rec_Ctr -1: 0]  Rec_Ctr;          // Recovery counter
```

```
wire    [L_divr: 0]        comparison;        // includes sign_bit

assign MSB_divr = divisor[L_divr -1];
assign w1_is_0 = !(|word1);
assignw2_is_0 = !(|word2);
assign num_shift_divr_is_0 = !(|num_shift_divisor);
assign quotient = ((divisor == 1) && ( num_shift_divr_is_0))? dividend[L_divn: 1]:
 (num_shift_divr_is_0) ? dividend[L_divn - L_divr : 0]: dividend[L_divn +1: 0];
assign remainder = num_shift_divr_is_0 ? (divisor == 1) ? 0:
 (dividend[L_divn: L_divn - L_divr +1] ): divisor;
assign divr_is_1 = (divisor == 1);
assign Rec_Ctr_Max = (Rec_Ctr == L_divr - num_shift_divisor);
assign Max = (num_shift_dividend == Max_cnt + num_shift_divisor);
assign comparison =
 (state_is_ShSub) ? dividend[L_divn + 1: L_divn - L_divr + 1] + {1'b1,
   ~divisor[L_divr -1: 0]} + 1'b1:
 MSB_divr ? dividend[L_divn + 1: L_divn - L_divr + 1] + {1'b1, ~(divisor << 1)} + 1'b1:
  dividend[L_divn + 1: L_divn - L_divr + 1] + {1'b1, ~divisor[L_divr -1: 0]} + 1'b1;

assign sign_bit = comparison[L_divr];

 always @(posedge clock, posedge reset)          // Register/Datapath operations
 if (reset) begin
  dividend <= 0;
  divisor <= 0;
  num_shift_dividend <= 0;
  num_shift_divisor <= 0;                         // use to down-cnt
  Rec_Ctr <= 0;
end
else begin
 if (Load_words == 1) begin
  dividend <= {1'b0, word1[L_divn -1: 0], 1'b0};
  divisor <= word2;
  num_shift_dividend <= 0;
   num_shift_divisor <= 0;
  Rec_Ctr <= 0;
 end
 if (Shift_divisor) begin
  divisor <= divisor << 1;
  num_shift_divisor <=  num_shift_divisor + 1;
 end
 if (Shift_dividend) begin
  dividend <= dividend << 1;
  num_shift_dividend <= num_shift_dividend + 1;
 end
 if (Sub_and_Shift) begin
  dividend <= {comparison[L_divr -1: 0], dividend [L_divn -L_divr: 1], 2'b10} ;
  num_shift_dividend <= num_shift_dividend + 1;
 end
```

```
    if (Subtract) begin
      dividend[L_divn + 1: 1] <= {comparison[L_divr: 0], dividend [L_divn -L_divr: 1]} ;
      dividend[0] <= 1;
    end
    if (Flush_divr) begin
      Rec_Ctr <= 0;
      divisor <= 0;
    end
    if (Xfer_Rem) begin
      dividend[ L_divn - L_divr + num_shift_divisor + 1 + Rec_Ctr] <= 0;
      divisor[Rec_Ctr] <= dividend[ L_divn - L_divr + num_shift_divisor + 1 + Rec_Ctr];
       Rec_Ctr <= Rec_Ctr + 1;
     end
  end
endmodule
```

End of Example 10.18

The simulation results for *Divider_RR_STG* are shown in Figure 10-79, with 17710 177_{10} divided by 510 5_{10} giving a quotient of 35_{10} and a remainder of 2_{10}. The displayed waveforms include the status signals generated by the datapath unit.

10.6.4 Division of Signed (2s Complement) Binary Numbers

The simplest method of forming the quotient of two signed numbers is to divide their magnitudes and then adjust the sign of the result, if necessary. Other, more complex, algorithms exist, but will not be considered here [1].

10.6.5 Signed Arithmetic

Ordinarily, Verilog performs signed arithmetic operation on 32-bit integers, and evaluates expressions with signed arithmetic only if every operand is signed. If an operand is unsigned an operation is unsigned. In Verilog 1995, only 32-bit integers are signed. Verilog 2001/2005 introduced several options for signed data types, ports, and functions [14]. It uses the reserved keyword ***signed*** to declare that a ***reg*** or a net type variable is signed, and supports signed arithmetic on sized literal integers and on vectors of any size, not just 32-bit values. Module ports can be declared to be signed, with the effect that the data type of the port will be treated as a signed variable. Likewise, the value returned by a function can be declared to be signed. Verilog 2001/2005 also supports type casting operators *$signed* and *$unsigned*. Care must be taken to ensure that the results of operations and synthesis of code are done with attention to the issue of sign extension [15].

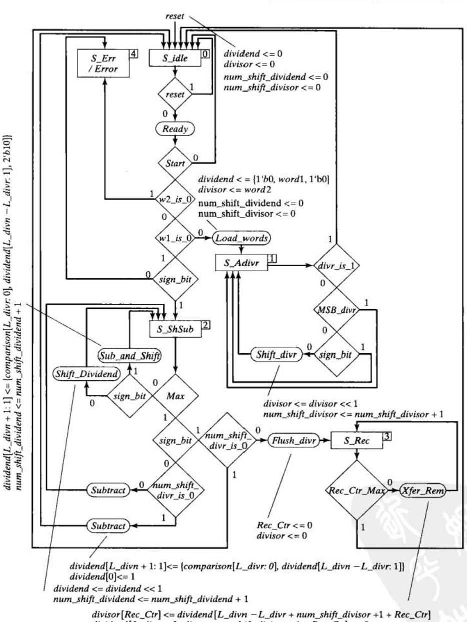


FIGURE 10-78 ASMD chart for *Divider_STG_RR*.

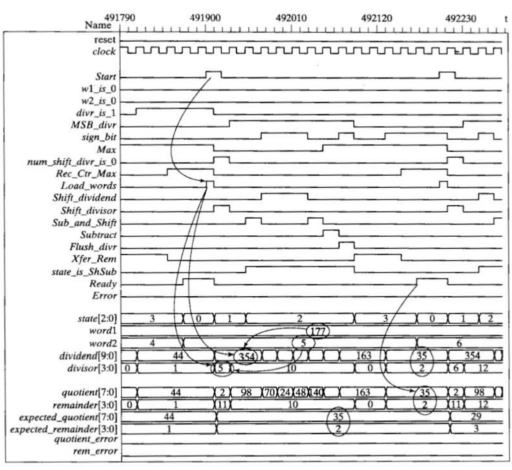


FIGURE 10-79 Simulation results for *Divider_STG_RR*, a binary divider with reduced registers. The model is efficient in its use of hardware and in its execution time.

Example 10.19

```
module Add_Sub (
  output signed [63: 0] sum_diff;
  input signed [63: 0] a, b;
...
```

```
function signed [64: 0] sum;
...
endmodule
```

End of Example 10.19

REFERENCES

1. Weste NHE, Eshraghian K. *Principles of CMOS VLSI Design*. Reading, MA: Addison-Wesley, 1993.
2. Katz RH. *Contemporary Logic Design*. Redwood City, CA: Benjamin/Cummings, 1994.
3. Smith MJS. *Application-Specific Integrated Circuits*. Reading, MA: Addison-Wesley, 1997.
4. Heuring V, Jordan H. *Computer Systems Design and Architecture*. Reading, MA: Addison-Wesley 1997.
5. Johnson EL, Karim MA. *Digital Design—A Pragmatic Approach*. Boston, MA: PWS, 1987.
6. Arnold MG. *Verilog Digital Computer Design*. Upper Saddle River, NJ: Prentice-Hall, 1999.
7. Ciletti MD. *Modeling, Synthesis, and Rapid Prototyping with the Verilog HDL*. Upper Saddle River, NJ: Prentice-Hall, 1999.
8. Thomas DE, Moorby PR. *The Verilog Hardware Description Language*, 3rd ed. Boston, MA: Kluwer, 1996.
9. Booth AD. "A Signed Binary Multiplication Technique." *Quarterly Journal of Mechanics and Applied Mathematics, 4*, 1951.
10. Patterson DA, Hennessy JL. *Computer Organization and Design—The Hardware/Software Interface*. San Francisco, CA: Morgan Kaufman, 1994.
11. Cavanaugh JJF. *Digital Computer Arithmetic*. New York: McGraw-Hill, 1984.
12. Kehtarnavaz N, Keramat M. *DSP System Design Using the TMS320C6000*. Upper Saddle River, NJ: Prentice-Hall, 2001.
13. Sternheim, E. et al., *Digital Design and Synthesis with Verilog HDL*. San Jose, CA: Automaton Pub. Co., 1993.
14. Ciletti, MD, *Starters Guide to Verilog 2001*, Upper Saddle River, NJ: Prentice-Hall, 2004.
15. Tumbusch, G., "Signed Arithmetic in Verilog 2001—Opportunities and Hazards", DVCon 2005, San Jose, 2005.

PROBLEMS

1. The 4-bit multiplier shown in Figure 10-13 can be modified to exploit a 4-bit carry look-ahead adder instead of a ripple-carry adder. Compare the performance and area of the two models.
2. Modify the models for *Multiplier_STG_0* to terminate its activity if the multiplier or the multiplicand is 0 (see Example 10.2).
3. Write behavioral models of the flip-flops, shift registers, and adders in Figure 10-16, and then build a structural model of the datapath architecture shown in Figure 10-16. Develop and verify an STG-based model for the controller.

4. Modify *Multiplier_STG_0* and *Multiplier_STG_1* (see Example 10.1) to terminate at any intermediate state if the multiplier is empty. Synthesize and compare the area of the new machine to that of the baseline machine.
5. The description in *Multiplier_ASM_0* (see Example 10.3) asserts *Ready* one cycle after asserting *Done*, and is efficient when the data word for the multiplier is 1, but does not treat the special case in which the data word for the multiplicand is 1. Modify the ASMD chart of *Multiplier_ASM_0* to form a chart for a machine that does not waste clock cycles when the data word for *multiplicand* is 1.
6. Develop logic to terminate the activity of multiplication in *Multiplier_RR_ASM* as soon as the subword corresponding to the shifted multiplier is empty of 1s (see Example 10.5).
7. Design and verify *Multiplier_IMP_Alternative*, an alternative sequential multiplier obtained by embedding the datapath operations within the implicit state machine behavior that implements the controller. Use *Multiplier_IMP_1* (see Example 10.6) as a starting point for your design.
8. Develop and verify *Multiplier_Booth_STG_1*, a Booth multiplier that is patterned after *Multiplier_STG_1* (see Example 10.2).
9. Develop and verify *Multiplier_Booth_ASM_0*, a Booth multiplier that is patterned after *Multiplier_ASM_0* (see Example 10.3).
10. Develop and verify *Multiplier_Booth_ASM_1*, a Booth multiplier that is patterned after *Multiplier_ASM_1* (see Example 10.4), with parameterized word length.
11. Develop and verify *Multiplier_Booth_RR_ASM*, a Booth multiplier that is patterned after *Multiplier_RR_ASM* (see Example 10.5). (*Hint*: Consider the role of an arithmetic shift operation for sign extension. Select a set of data that demonstrates a significant reduction in clock cycles to multiply 16-bit words.)
12. Develop and verify *Multiplier_Booth_IMP_1*, a Booth multiplier that is patterned after *Multiplier_IMP_1* (see Example 10.6). Verify that the machine resets correctly from any intermediate state.
13. Write the Booth code for the multiplier word shown in Figure P10-13:

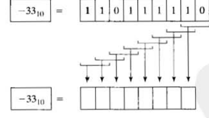


FIGURE P10-13

14. Develop and verify *Multiplier_Radix_4_STG_1*, a modified Booth multiplier using radix-4 encoding that is patterned after *Multiplier_STG_1* (see Example 10.2). Use the testbench given at the companion Web site to conduct exhaustive verification.
15. Develop and verify *Multiplier_Radix_4_ASM_0*, a modified Booth multiplier using radix-4 encoding that is patterned after *Multiplier_ASM_0* (see Example 10.3). Use the testbench given at the companion Web site to conduct exhaustive verification.

16. Develop and verify *Multiplier_Radix_4_ASM_1*, a modified Booth multiplier using radix-4 encoding that is patterned after *Multiplier_ASM_1* (see Example 10.4). Use the testbench given at the companion Web site to conduct exhaustive verification.
17. Develop and verify *Multiplier_Radix_4_RR_ASM*, a modified Booth multiplier using radix-4 encoding that is patterned after *Multiplier_RR_ASM* (see Example 10.5). (*Hint*: Consider the role of an arithmetic shift operation for sign extension. Use the testbench given at the companion Web site to conduct exhaustive verification.)
18. Develop and verify *Multiplier_Radix_4_IMP_1*, a modified Booth multiplier using radix-4 encoding, that is patterned after *Multiplier_IMP_1* (see Example 10.6). Use the testbench given at the Web site to conduct exhaustive verification. Verify that the machine resets correctly from any intermediate state.
19. The machine *Multiplier_Radix_4_STG_0* has a default state assignment that directs the state to *S_idle* if an unspecified state is encountered (see Example 10.12). This could cause a mistaken interpretation of *Ready*. Develop and verify a modified machine with a fail-safe behavior.
20. The machine *Multiplier_Radix_4_STG_0* has a simple binary coding of the controller's state (see Example 10.12). Develop and verify a modified machine with a one-hot code, and compare the results of synthesizing both designs.
21. Develop and verify *Multiplier_Radix_4_STG_1*, a more efficient version of the bit-pair encoding multiplier described by *Multiplier_Radix_4_STG_0* that uses additional logic to eliminate needless computation when *word1* or *word2* is 0 or 1 (see Example 10.12).
22. Write a testbench to exercise all of the state transitions of *Divider_STG_0* in Figure 10-56.
23. Using the testbench *test_Divider_STG_0* (given at the companion website), verify the features of *Divider_STG_0* that were specified in Example 10.13.
24. In Example 10.14, *Divider_STG_0* forms *remainder, Error*, and *Done* as combinational outputs by continuous assignment statements. This may lead to a higher than necessary signal activity because *remainder* has needless intermediate transitions while *quotient* is being formed. Similarly, *Error* and *Ready* generate needless simulation activity because their continuous assignment statements are activated every time *state* changes. Develop and verify a modified machine that reduces signal and simulation activity by forming *quotient, Error,* and *Ready* as registered outputs.
25. In Example 10.14 *Divider_STG_0*, an attempt to divide by 0 drives the state to *S_err*, where the machine remains until *reset* is asserted. Develop an STG for an alternative machine that recovers from *S_4* when *Start* is asserted. Develop and verify a Verilog description of the alternative machine.
26. The divider described by *Divider_STG_0* in Example 10.14 asserts *Ready* when the machine enters *S_3*. An external agent that is using *Ready* to determine whether *quotient* is valid must wait until the next clock cycle to read *quotient* (i.e., the agent reads *quotient* at the second clock after it is formed). However, there are conditions in which *quotient* is formed sooner. Modify the STG shown in Figure 10-67 so that *Ready* asserts as soon as *quotient* is valid, and modify *Divisor_STG_0* to form an alternative machine that implements the STG (i.e., allows an external agent to read *quotient* at the first clock cycle after it is formed).
27. The machine described by *Divider_STG_1* in Example 10.16 cannot be synthesized because *remainder* uses a dynamically determined (i.e., during simulation) shift operation. Develop a synthesizable implementation by (1) modifying

the machine's STG to accommodate an additional state in which the reminder is adjusted, (2) developing and verifying the Verilog model of the machine, and (3) verifying that the synthesized machine's behavior matches that of the behavioral description.

28. The self-aligning divider described in Example 10.16 (*Divider_STG_1*) has a more efficient alternative (speedwise) in which the operations of shifting and subtracting execute in the same clock cycle. Develop an STG for an alternative machine that combines the operations in the same state. Develop and verify a Verilog description of the alternative machine. Examine and discuss whether the architecture could have an overflow condition.
29. The self-aligning divider described in Example 10.16 (*Divider_STG_1*) has a more efficient (registerwise) alternative in which the bits of the quotient are loaded into the trailing end of the register, holding the dividend as the dividend is shifted. Note that in *Divider_STG_1* the operations of shifting and adding occur in different clock cycles. Be aware that if the shift and subtract operations are not combined in the same clock cycle, a data configuration might cause an overflow condition in which the quotient bit formed by subtraction would be targeted to occupy the rightmost cell of the register, holding *dividend* before it is vacated by a shift operation. Develop an STG for this alternative machine, with an additional output to signal the overflow condition and then develop and verify a Verilog description of the machine.
30. Compare the speed and area of a synthesized 4×4 array multiplier to that of a sequential multiplier.
31. The Verilog description of *Divider_STG_0* (see Example 10.14) shows alternative implementations of subtraction. Synthesize each version of the model and compare the results.
32. Synthesize and compare *Divider_STG_0* and *Divider_STG_0_sub* (see Example 10.14) to determine whether eliminating the comparator from the architecture results in a more efficient implementation.
33. Careful examination of the simulation results in Figure 10-72 reveals that *Divider_STG_1* wastes a clock cycle before and after aligning the dividend. Develop and verify an alternative machine that eliminates the wasted clock cycles.
34. The subtraction in *Divider_STG_0* (see Example 10.14) can be written directly as *Dividend - Divisor*, to exploit the built-in 2s complement arithmetic of Verilog. Conduct an experiment to determine whether your synthesis tool synthesizes the alternative implementation more efficiently.
35. Modify the machine *Divider_STG_1* (see Example 10.16) to detect an attempt to divide by 1, allowing the machine to have higher throughput.
36. The machine *Divider_STG_RR* (see Example 10.17) asserts *Ready* when the state returns to *S_idle* after completing a division. Explore the possibility of asserting *Ready* sooner, allowing the machine to have higher throughput.
37. The machine *Divider_STG_RR* (see Example 10.17) detects an attempt to divide by 1 in state *S_Adivr*. Explore the possibility of detecting a divide by 1 in state *S_idle*, allowing the machine to have higher throughput.
38. Determine whether the descriptions given by *Divider_STG_0*, *Divider_STG_0_sub*, *Divider_STG_1*, and *Divider_STG_RR* are synthesizable models. Modify any machine that cannot be synthesized so that it can be synthesized.
39. Using a feedforward cutset for the 4×4 binary multiplier shown in Figure 10-14, develop and verify a balanced one-stage pipelined implementation of the circuit.

40. Using a feedforward cutset for the 8 × 4 binary multiplier shown in Figure P10-40, develop and verify a balanced one-stage pipelined implementation of the circuit.
41. Identify the computational wavefronts of the systolic array in Figure 10-14, and develop a reservation table (see Table 9-1) that utilizes the maximum number of processors that could execute currently to form the 4 × 4 product in a synchronous environment with memory.
42. Identify the computational wavefronts of the systolic array in Figure P10-40, and develop a reservation table (see Table 9-1) that utilizes the maximum number of processors that could execute currently to form the 8 × 4 product in a synchronous environment with memory.
43. A carry-select (conditional-sum) adder can be used to improve the performance of operational units that require an adder circuit. The adder consists of full adders and 2:1 muxes in a configuration in which the datapaths of the muxes are from two full adders with the same input data bits. Certain of the full-adder cells have a hardwired carry bit. The actual carry bit from a previous stage selects the correct cell for a given data pattern. Each cell generates a carry-out bit for the next stage. The architecture offers a speed improvement because the parallel addition at the inner stages of the adder occurs while the carry to those cells is developed and passed to the mux, rather than after the carry has arrived.

    a. Develop suitable low-level cells (with hypothetical propagation) delays, or use cells from a standard-cell library (with physical delays), and write a Verilog description of the carry-select adder shown in Figure P10-43. Use the ***supply0*** and ***supply1*** nets to implement the hardwired carries. Consider the issue of whether the hardwired carries should be driven by an internal signal or a signal passed through the port.
    b. Develop and verify the circuit using a nonexhaustive testbench to gain a high level of confidence in the functionality of your design. Carefully select a small but robust set of test vectors. Discuss the testing strategy that you used.
    c. Develop an automated testbench that includes a 6-bit behavioral adder and compares its output to that of the carry-select adder (at suitable times). Use an exclusive-or scheme to generate a *test_results_message* that the circuit operates correctly or not. (*Note*: Do not generate hard copy of the exhaustive simulation.) This scheme should exploit the ***$display*** system task to observe the outputs of the two adders at some time after each pattern is applied to their inputs. (*Note*: The ***$display*** task provides dynamic control over the display of information. It displays results only when the statement executes within a behavior. The ***$monitor*** tasks executes when its argument has an event. If the ***$monitor*** task is used instead of ***$display***, the comparison of the two responses will report errors until the output of the gate-level adders stabilizes.)

44. Develop an FPGA-based calculator with the following features: Data entry is through a hex keypad in which 10 keys are used to enter the decimal digits, and the remaining keys are used for the following functions: *Enter_data, addition, subtraction, multiplication, division*, and *decimal point*. The calculator is to have a 10-digit display.
45. Develop, verify, and synthesize an eight-stage finite-duration impulse response (FIR) filter for a 16-bit datapath in 2s complement format (see Figure 9-23). Each multiply and accumulate (MAC) unit is to use a Booth sequential multiplier and a two-stage pipelined adder. Determine the maximum clock frequency at which the data sequence can be supplied to the circuit and the frequency of the internal

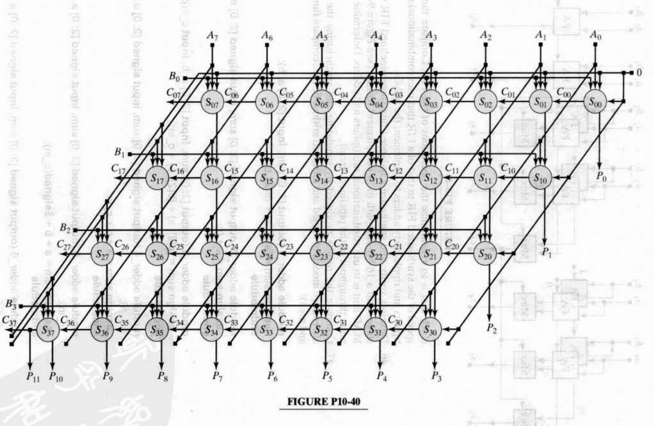


FIGURE P10-40

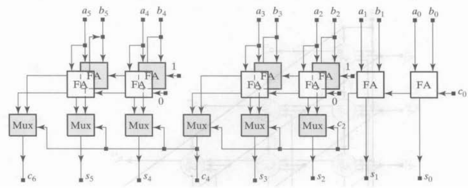


FIGURE P10-43

clock that is to synchronize the operations of the FIR. Compare the area and speed of the synthesized FIR to that of a FIR that uses combinational array multipliers and ripple carry adders to implement the MACs.

46. Develop, verify, and synthesize a balanced one-stage pipelined FIR with eight stages and a 16-bit datapath in 2s complement format (see Figure 9-23). Each MAC unit is to use combinational multipliers and adders. Determine the maximum throughput of the pipelined circuit.
47. Compare the results produced by simulating and syntheisizing the following alternative models of signed and unsigned arithmetic (i.e., compare functionalilty and area):

```
module adder_1 (output [3: 0] sum, input [2: 0] a, b);
  assign sum = {a[2], a} + {b[2], b};
endmodule

module adder_2 (output signed [3: 0] sum, input signed [2: 0] a, b);
  assign sum = a + b;
endmodule

module adder_3 (output [3: 0]  sum, input [2: 0] a, b, input c_in);
  assign sum = {a[2], a} + {b[2], b} + c_in;
endmodule

module adder_4 (output signed [3: 0] sum, input signed [2: 0] a, b, input
   c_in);
  assign sum = a + b + c_in;
endmodule

module adder_5 (output signed [3: 0] sum, input signed [2: 0] a, b, input
   c_in);
  assign sum = a + b + $signed(c_in);
endmodule

module adder_6 (output signed [3: 0] sum, input signed [2: 0] a, b, input
   c_in);
  assign sum = a + b + $signed({1'b0,c_in});
endmodule
```

CHAPTER 11

Postsynthesis Design Tasks

The design flow of an application-specific integrated circuit (ASIC) includes tasks for postsynthesis design validation, timing verification, test generation, and fault simulation.[1] Design validation confirms that the functionality of the synthesized gate-level netlist matches that of the register transfer level (RTL) model from which it was synthesized. Timing verification checks whether a physical implementation of the design meets timing specifications, and it determines the maximum frequency at which the clock of a synchronous circuit can operate. Test generation develops stimulus patterns that will detect the presence of a process-induced fault in the fabricated circuit. Fault simulation determines how well a set of stimulus patterns exposes the faults that could be in the circuit.

11.1 Postsynthesis Design Validation

Postsynthesis design validation is not intended to verify that the functionality of a design is correct. Functional verification is achieved earlier in the design flow, before synthesis, with RTL models that simulate more efficiently than gate-level models.

There are two approaches to design validation: formal methods and simulation. Formal methods are beyond the scope of this textbook; we will consider only simulation, which is widely used by the ASIC industry. For our purposes, postsynthesis design validation is to detect a simulation mismatch between the functionality of the RTL model and the gate-level netlist. The consequences of a simulation mismatch can be serious, because the physical circuit may fail to operate correctly. The RTL and gate-level models can be simulated simultaneously, or Verilog system tasks can be used to record the stimulus and response patterns of the RTL circuit and compare them to those of the gate-level circuit when it is exercised by the same stimulus.

[1]See Figure 1.1.

There are several potential sources of simulation mismatch between RTL and gate-level models of a circuit. The gate-level model of a circuit uses standard cells, which have built-in descriptions of technology-specific propagation delays of the devices; the RTL model is delay-free. Consequently, at sufficiently high clock frequencies, a timing violation will occur in the gate-level model, but not in the RTL model. Depending on how the models treat timing violations, the values that are propagated in the simulation of the gate-level circuit may differ from those propagated by the RTL model. If the clock speed is fixed by a specification, the RTL design must be remodeled and/or resynthesized to achieve timing margins that eliminate the mismatch between the circuits.

Simulation mismatch can occur if the modeling style allows software race conditions in a sequential machine. In general, software race conditions exist when multiple cyclic behaviors in a Verilog model make simultaneous assignments to the same variable. The order in which multiple cyclic behaviors are executed by a simulator is indeterminate, and there is no simulator-independent way to determine whether a variable will be assigned value by a procedural assignment in one behavior before or after it is referenced by a procedural assignment in another behavior. So beware of models in which a variable is assigned value in one behavior at the same time that the variable is referenced by another behavior. One way to prevent ambiguous outcomes caused by software race conditions in a latch-free sequential machine is to place all procedural assignments in a single cyclic behavior and order the statements to produce the correct sequence of assignments. However, this approach might not be convenient.

Sequential machines can have race conditions if there is feedback from a datapath to the state machine that controls the datapath. The methodology discussed in Chapter 7 produces race-free logic in such machines by using use nonblocking (<=) assignments in edge-sensitive cyclic behaviors, and blocked (=) assignments in level-sensitive behaviors, and by having no variable simultaneously referenced and assigned value by multiple blocked assignments.[2]

The style used to model circuits with latches can also lead to race conditions. A circuit will have a race condition if a latch is a reconvergent fanout node.[3] For example, if the enable line and the datapath of a latch have a common variable it is possible for the enable and the data to change simultaneously, and race. This style of design should be avoided because the outcome of the race is indeterminate.

Example 11.1

The output and the level-sensitive (latched-high, enabled-low) event control expression (sensitivity list) of the latches in Figure 11-1(a) are both conditioned on the datapath [1]. The RTL models *Latch_Race_1* and *Latch_Race_2* below differ only in the order in which the cyclic behaviors appear in the code. The comments in the code

[2]See Howe [1] for further discussion of simulation mismatch in design validation.

[3]A node is a reconvergent fanout node if there are multiple paths to it through combinational logic from some other node.

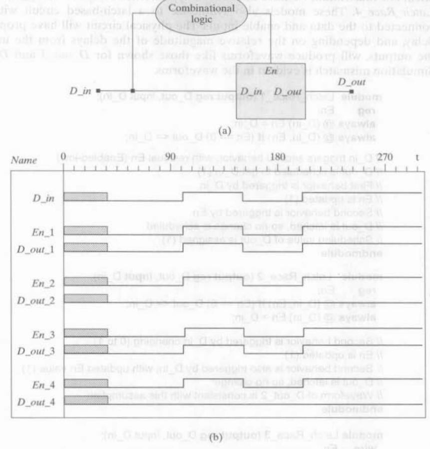


FIGURE 11-1 Latch circuit: (a) a race condition exists between the datapath and the enable input, and (b) the value that is latched in the RTL model depends on the order in which the statements are evaluated; the race condition in the physical circuit depends on the relative delay between the enabling input and the datapath through the device. *This style of design should be avoided.*

explain the sequence of events and evaluations that correspond to the observed waveforms.[4] The simulation results in Figure 11-1(b) show that *D_out_1* and *D_out_2* differ. The models synthesize to the same structure, a hardware latch with *D_in* connected to the data and enable inputs of the latch. These examples demonstrate the danger of having a race between the datapath and the enabling signal of a

[4]The simulator evaluates the second cyclic behavior before the the first cyclic behavior.

latch. For illustration, we have introduced hypothetical delays in *Latch_Race_3* and *Latch_Race_4*. These models also synthesize to a latch-based circuit with *D_in* connected to the data and enable inputs. The physical circuit will have propagation delay, and depending on the relative magnitude of the delays from the inputs to the outputs, will produce waveforms like those shown for *D_out_3* and *D_out_4*. Simulation mismatch is evident in the waveforms.

```
module Latch_Race_1 (output reg D_out, input D_in);
 reg     En;
 always @ (D_in) En = D_in;
 always @ (D_in, En) if (En == 0) D_out <= D_in;

// D_in triggers second behavior, with residual En (Enabled-low)
// D_out is scheduled to get D_in (1)
// First behavior is triggered by D_in
// En is updated (1)
// Second behavior is triggered by En
// D_out is latched, so no change is scheduled
// Scheduled value of D_out is assigned (1)
endmodule

module Latch_Race_2 (output reg D_out, input D_in);
 reg     En;
 always @ (D_in, En) if (En == 0) D_out <= D_in;
 always @ (D_in) En = D_in;

// Second behavior is triggered by D_in changing (0 to 1)
// En is updated (1)
// Second behavior is also triggered by D_in, with updated En value (1)
// D_out is latched, so no change
// Waveform of D_out_2 is consistent with this assumption
endmodule

module Latch_Race_3 (output reg D_out, input D_in);
 wire    En;
 buf #1 (En, D_in);
 always @ (D_in, En)
 if (En == 0) D_out <= D_in;
endmodule

// Change in D_in schedules delayed change in En
// Change in D_in schedules change in D_out
// D_out is updated
// Delayed change in En triggers cyclic behavior, but En has
// D_out latched.
// D_out_3 should exhibit change of D_out_3 coincident with D_in

module Latch_Race_4 (output reg D_out, input D_in);
 wire    En, D_in_del;
 buf #2 (D_in_del, D_in);
```

```
    buf (En, D_in);
    always @ (D_in_del, En)
    if (En == 0) D_out <= D_in;
endmodule
```

End of Example 11.1

11.2 Postsynthesis Timing Verification

Postsynthesis timing verification is necessary because functional verification with RTL models does not consider propagation delays, does not verify that hardware timing constraints are met, and does not verify that performance specifications for input–output (I/O) timing are met. Synthesis tools incorporate timing analysis in the engine that maps a generic design into a physical implementation. The accuracy of the timing analysis performed by synthesis tools is limited, because the tools use prelayout statistical estimates (e.g., wireload models) of the delay caused by the resistance and parasitic capacitance of the interconnect and loading of the nets in the design. The technology-mapping engine does not have access to the actual delays that result from the place-and-route step, so the netlist produced by the synthesis tool cannot account for those delays accurately. Estimates of the actual delays are used. The actual values of the resistance and capacitance of the interconnect must be extracted from the layout and used to back-annotate the delay models of the gates to obtain a more accurate analysis of the postlayout timing.[5]

The speed at which a circuit can operate correctly is ultimately limited by the longest logic path, the so-called *critical path*, and by the physical/operational constraints of the storage devices in the chip. The critical path and the paths whose lengths (in time) are close to that of the critical path must be verified to satisfy the timing constraints. Timing verification must consider the propagation delays of gates, the interconnect between gates, clock skew, I/O timing margins, and device constraints (e.g., setup, hold, and clock pulsewidth of flip-flops) to verify that the circuit will meet the timing specifications of the design without violating device constraints. If the setup or hold condition of an edge-triggered flip-flop is violated, the flip-flop may enter a metastable state (see Chapter 5).

Timing verification uses models of the devices and the interconnect in a circuit to analyze its timing and to determine whether hardware timing constraints and input–output timing specifications are met in the physical design (prelayout and postlayout). Timing verification can be achieved *directly*, by simulating the behavior of a circuit and confirming that hardware constraints and performance specifications are

[5]More recent developments in EDA tools seek to improve timing closure by making routing information available to the synthesis engine. See www.cadence.com.

TABLE 11-1 A comparison of methods for timing verification.

	Approaches to Timing Verification	
	Dynamic	**Static**
Method	Simulation	Path analysis
Needs Test Vectors	Yes	No
Coverage	Pattern-dependent	Pattern-independent
Risk	Missed alarms	False alarms
Min-Max Analysis	Discontinued	Yes
Couple with Synthesis	Not feasible	Yes
CPU Run time	Days/weeks	Hours
Memory Use	Heavy	Light–moderate

met, or *indirectly*, by analyzing all possible signal paths in the circuit and determining whether timing constraints are satisfied, without actually simulating the behavior of the circuit. These two approaches to timing verification are referred to as dynamic timing analysis (DTA) and static timing analysis (STA), respectively [2,3]. Table 11-1 compares features of dynamic and static timing analysis.

DTA uses behavioral, gate-level, and switch-level models of a circuit to simulate and analyze its functional paths; transistor-level models are simulated with analog simulators.[6] STA uses the same models as DTA, but creates a directed acyclic graph (DAG) of the circuit by systematically and exhaustively extracting the topology of a gate-level representation of the circuit and calculating the propagation delays on all paths. If all of the possible paths meet their timing constraints and specifications, then the circuit will meet them independently of the stimulus pattern that is applied in operation.

Dynamic and static timing analysis have different risks and costs. DTA may have missed alarms (i.e., fail to detect and report a timing violation). DTA is pattern-dependent, so the stimulus patterns used to exercise the circuit might not exercise the longest path. Developing a robust set of stimulus patterns to verify timing is difficult and impractical for today's large, complex circuits. On the other hand, STA, which exhaustively considers all possible topological paths of a circuit, may generate false alarms by reporting a timing violation on a nonfunctional path (i.e., one that cannot be exercised in operation). Also, event-driven simulation for DTA of large circuits requires significant memory capacity and is relatively slow compared to the time required for a static timing analyzer to analyze the design and detect timing violations.

Timing closure refers to the problem of placing and routing the cells, signal paths, and clock tree of a design to meet the timing specifications. Timing closure is an issue because synthesis tools do not have postlayout information about the delays of the routed interconnect. Failure to achieve timing closure requires resynthesis and/or re-placement and routing of cells, and incurs additional cost. Some synthesis tools address the problem

[6] Analog simulation is time consuming, and usually done to verify only the critical paths of a high-performance circuit.

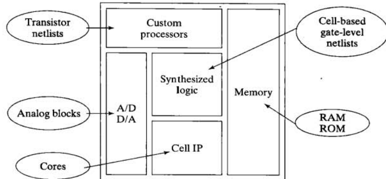


FIGURE 11-2 An SoC approach to design integrates on a single chip intellectual property from several sources.

of timing closure by having more accurate models of the interconnect loading, and by incorporating physical synthesis within the overall design flow.[7]

DTA is limited to circuits that have about 1 million gates. It has very limited application in an SoC (system on chip) environment, shown in Figure 11-2, where intellectual property (IP) from multiple vendors is integrated on a single chip. It can be very difficult to integrate the test patterns of the various IPs, thus limiting the coverage that can be achieved by simulation. On the other hand, STA does not depend on patterns, achieves full coverage, and requires significantly less time. STA is the appropriate method for timing analysis and verification in SoC applications. Because STA can exploit partitioning of complex circuits, timing analysis of multicore SoCs with millions of gates is feasible.

11.2.1 Static Timing Analysis

Our focus will be on full-chip, gate-level, static timing analysis for synchronous designs. STA forms a DAG from a netlist of a circuit. The nodes of the DAG represent gates, and the edges of the graph represent signal paths. The topological paths of a DAG include the timing paths of the circuit (i.e., paths that can be exercised by applying stimulus patterns to the primary inputs). The edges are annotated with the propagation delays of the paths. The DAG must be acyclic (i.e., it must not have a feedback path).

Example 11.2

The circuit in Figure 11-3(a) has the timing DAG shown in Figure 11-3(b). For simplicity, the gates in this example are shown with symmetric rising and falling delay.

[7]See www.magma-da.com.

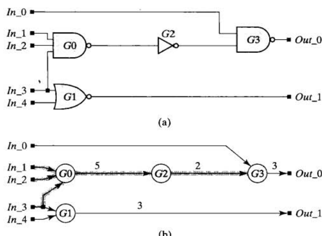


FIGURE 11-3 A combinational logic circuit and its timing DAG.

End of Example 11.2

A circuit may have four types of timing paths: (1) paths that originate at a primary input and terminate at the data input of a storage element, (2) paths that connect the output of a storage element to the data input of a storage element, (3) paths that originate at the output of a storage element and terminate at a primary output of the chip, and (4) paths that connect primary inputs to the primary outputs of the circuit. Each type of path passes through combinational logic. STA examines timing paths between their sources and destinations, commonly called *startpoints* and *endpoints*, as shown in Figure 11-4.

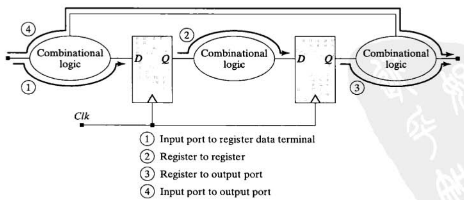


FIGURE 11-4 Startpoints and endpoints of signal paths for timing analysis in a synchronous circuit.

The startpoints of the paths of a circuit are its primary inputs (i.e., package input pins) and the clock pins of its storage elements (sequential devices). The physical path originating at a sequential device is attached to the output of the device, but the startpoint of the timing path is the clock pin, because the action of the clock initiates signal propagation along the physical path. The endpoints of the timing paths of a circuit are the primary outputs (package pins) and the data inputs of the storage elements. Not every topological path of the DAG is a timing path, and separate timing paths may exist between a given startpoint and endpoint, depending on whether rising and falling transitions of a signal have symmetric propagation delays.

11.2.2 Timing Specifications

Performance specifications may constrain the offsets of signals at the interfaces of a circuit with its environment, and may constrain the delays of internal paths.[8] An *input delay (offset) constraint* applies to paths from primary inputs (input pads) to a storage element of the design, and specifies the latest time of arrival of the input signal relative to the active edge of the clock that launched the signal. In a fully synchronous system, the signal arriving at the input pad of a circuit would have been launched by a clock edge in the circuit to which the input is connected. A timing analyzer uses the specified input constraint, t_{input_delay}, to determine the timing margin, denoted by t_{input_margin} in Figure 11-5, between the arriving signal and the next active edge of the clock. t_{input_margin} determines the time available for the arriving signal to pass through the internal combinational logic of the circuit and arrive at the path endpoint in time to meet the setup conditions of the flip-flop.

Output delay (offset) constraints apply to paths from storage elements to primary outputs. An output constraint specifies the latest time that a signal propagating from a startpoint may reach its endpoint, relative to the active edge of the clock at the startpoint, as shown in Figure 11-6. The timing analyzer uses t_{output_delay} to calculate

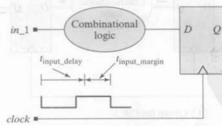


FIGURE 11-5 Input delay constraint.

[8]Path constraints influence the time required for a synthesis tool to converge on an implementation that satisfies the constraints; path constraints also influence the placement of logic within an FPGA. A design targeted to an FPGA should be synthesized initially without timing constraints and without pin assignments, to determine whether the design fits within the selected part.

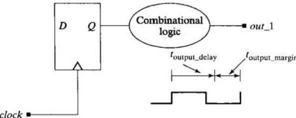


FIGURE 11-6 Output delay constraint.

t_{output_margin}, the time available for the output signal to reach its destination before the next active edge of the clock in the circuit to which the output is connected.

An *input–output (pad to pad) constraint* applies to a path whose startpoint is a primary input and whose endpoint is a primary output. The I/O delay constrains the maximum timing length of a combinational path between a primary input and a primary output.

A *cycle time (period) constraint* specifies the maximum period of the clock of a synchronous circuit, and applies to the paths between registers. The constraint specifies the period of a named clock. A timing analyzer might also allow specification of the duty cycle and offset (skew) of the clock waveform. If a circuit has multiple independent clock domains,[9] the paths are grouped with the clock that synchronizes the storage elements in the domain, as shown in Figure 11-7. Paths whose endpoints are not the data input of a storage device are placed in a default group.

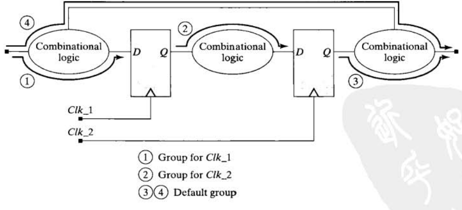


FIGURE 11-7 Path groups for a synchronous circuit with multiple clock domains.

[9] Timing analyzers may detect clocks that are derived from the same source (i.e., dependent clocks), but we will not consider them.

Timing analysis determines whether the timing constraints of the paths in a circuit are satisfied. No combinational feedback loops are allowed, and all register feedback paths are broken by the clock boundary. All four types of paths pass through combinational logic, and the propagation delay of each path is calculated for both rising and falling edge transitions by back-tracing from the endpoint of the path to its startpoint, accumulating the propagation delays along the path. Paths are sorted by their length and verified to meet I/O timing constraints. If the propagation delays of the devices are given as a range of values (i.e., min:max), the delay of the path is reported as a (min:max) range too.

Verification of an input constraint considers the setup time of the sequential device at the endpoint of the path. Likewise, an output constraint is verified by considering the clock-to-output delay of the clock that launches the signal along the path from the sequential device to the primary output. Verification of a cycle time constraint must consider the clock-to-output delay of the device at the path startpoint, the propagation delay through the combinational logic on the path, the setup time of the device at the endpoint of the path, and any skew of the clock. *The longest path through the combinational logic between storage elements will determine the minimum period of the clock.*

The paths in a circuit are assumed to be statically sensitized for timing analysis; that is, the side inputs of any gate along the path are fixed and do not block propagation of a signal through the gate. In Figure 11-8, the side inputs of the NAND gate must be 1.

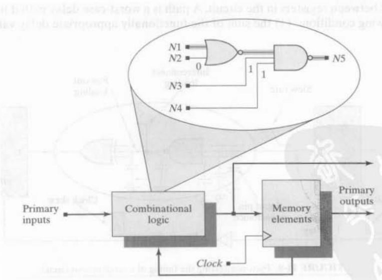


FIGURE 11-8 The side inputs of the gates along a statically sensitized path must not block propagation of a signal through the gate.

11.2.3 Factors That Affect Timing

In a typical synchronous circuit, signals propagate from a source register to a destination register through combinational logic as shown in Figure 11-9. The delay along the path is a result of the delay between the clock and the output of the register that propagates the signal, and the propagation delay of the gates along the path. The propagation delay of a gate is affected by the capacitance of the input pins of the gates that it drives (i.e., the fanout loads on the path) and the loading associated with the resistance and parasitic capacitance of the path.

The physical gates and nets that are attached to the output of a gate add to its intrinsic capacitance, thereby increasing the propagation delay between a change at the input of the gate and the effect of the change at the output of the gate. The slew rate of the signals driving a gate affect the propagation delay of the gate too, because the capacitor in an RC network will charge slower if the input has a large rise-time than if the input is a step signal. The registers in a given clock domain of a circuit are synchronized by a common clock, but the edges of the clock cannot be exactly aligned in a physical chip, because the clock signal propagates to its destinations along different physical paths. The misalignment of clock edges in a synchronous circuit is referred to as clock skew. Clock skew reduces the timing margin between the data and the clock at the destination register.

The worst-case delay along a path in a circuit is determined by the intrinsic propagation delays[10] of the combinational logic and memory elements along the path, the fanout loading of the gates on the path, the interconnect loading of the signal path, and the signal slew rate. The period of the clock must accommodate the worst-case path delay between registers in the circuit. A path is a worst-case delay path if it satisfies the following conditions: (1) the sum of the functionally appropriate delay values through

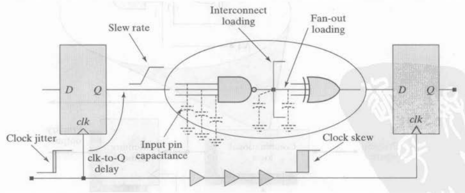


FIGURE 11-9 Factors affecting the timing of a synchronous circuit.

[10]The intrinsic delay is that which the gate exhibits independently of any fanout loads.

all of the gates in the worst-case path must not be less than that sum over any other combinational path, and (2) there must exist a vector of primary inputs and memory element values such that the logical value of the output depends on the logical value at every node on the path (i.e., the path from the startpoint to the endpoint can be sensitized), and, if the endpoint is a memory element, a clock transition will be enabled.

The following terms and notation will be used to describe the factors affecting the minimum period of the clock of a synchronous machine:

$t_{clk_to_Q}$	Delay between the active edge of the clock and the valid output of a flip-flop synchronized by the clock.
t_{comb_max}	Longest path delay through combinational logic.
t_{setup}	Setup time of a flip-flop driven by the combinational logic.
t_{skew}	Clock skew.

The delay t_{comb_max} depends on the intrinsic gate delay, slew, and loading due to fanout and routing-dependent interconnect. In deep submicron designs (i.e., those with physical dimensions $<.18\ \mu m$), the interconnect delay is considered to play a dominant role. The clock period must be long enough to allow the signal to satisfy the setup-time margin of the destination register (i.e., the signal must arrive at the data input of the register in time to be stable during the setup time of the register). Put another way, the longest path must satisfy the constraint: $t_{comb_max} < T_{clock} - t_{clk_to_output} - t_{setup} - t_{skew}$, or $t_{setup_time_margin} > 0$, where $t_{setup_time_margin} = T_{clock} - t_{clk_to_Q} - t_{setup} - t_{skew} - t_{comb_max}$. The circuit has a cycle-time violation if $t_{setup_time_margin} \leq 0$.

The hold-time margin of a register imposes a constraint on the shortest path through the logic. The shortest path must satisfy the constraint: $t_{comb_min} > t_{hold} - t_{clk_to_output} + t_{skew}$, or $t_{hold_time_margin} > 0$, where $t_{hold_time_margin} = t_{comb_min} - t_{hold} + t_{clk_to_output} - t_{skew}$. This constraint prevents a race condition between the output of the register at the startpoint of the path and the data input of a register at the endpoint of the path.[11] A change in the value of the signal at the startpoint of the path must not arrive too soon at the register at the endpoint.

The relationship between the clock period T_{clock} and the delays of a signal path is illustrated in Figure 11-10, for a constraint in which the skew of the clock is neglected.

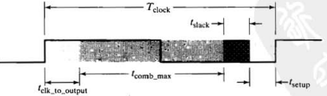


FIGURE 11-10 In a skew-free circuit the clock period must satisfy the constraint: $T_{clock} > t_{clk_to_output} + t_{comb_max} + t_{setup}$.

[11]Note that minimum path delays are used in the hold-time constraint.

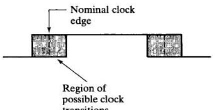


FIGURE 11-11 Skew introduces ambiguity in the location of the clock edge.

The clock period must be greater than the sum of: the clock-to-output delay, the maximum delay of the combinational path, and the setup time of the endpoint (assumed to be a flip-flop). The slack, t_{slack}, of a path is the difference between the clock period and the delay of the path. If $t_{slack} \leq 0$ for any path the circuit will have a setup timing violation if that path is exercised by the stimulus applied to the circuit.

Synchronous operation of a circuit requires that all memory elements be synchronized by the same clock edge. *Clock skew* is the variation in the arrival of the clock edges at their destinations, relative to the edges of the source of the clock signal. The skew can be due to the intrinsic jitter of the clock itself, or it can be due to routing-induced differential propagation delay of the clock to the cells that are driven by it. The routing-induced skew of the clock is due to the loading (RC metal interconnect and memory elements), and by buffer chains in the clock distribution path. Long metal runs introduce proportionately more delay. Path-induced skew is unavoidable and must be anticipated [4].

Figure 11-11 illustrates the ambiguity that describes the location of the edge of a clock with skew. The jitter establishes an interval of ambiguity on each side of the nominal location of the clock edge. The actual transition of the clock occurs within the shaded regions, but its actual location is unknown. Figure 11-12 illustrates the factors that determine the maximum frequency of a clock that has skew. The effect of the skew is to increase the minimum period of the clock, compared to a skew-free clock. The clock period must satisfy the following constraint in the presence of skew: $t_{clock} > t_{clk_to_output} + t_{comb_max} + t_{setup} + t_{skew}$.

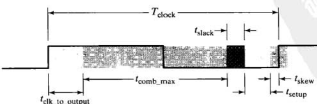


FIGURE 11-12 The period of the clock must be increased to compensate for skew, and must satisfy the constraint: $T_{clock} > t_{clk_to_output} + t_{comb_max} + t_{setup} + t_{skew}$.

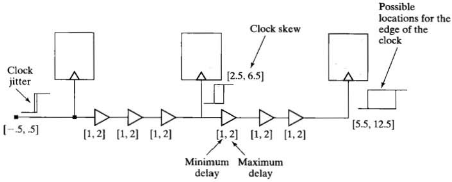


FIGURE 11-13 Buffers in a clock path progressively increase the skew of the clock at its destinations.

Buffers and any other logic in the path of the clock signal introduce clock skew. In Figure 11-13, the clock edge occurs at a different time at each of the registers. The ambiguity of the location of the edge of the clock increases along the path. For a given clock period, the allowed skew is bounded by $T_{\text{skew}} < T_{\text{clock}} - t_{\text{clk_to_output}} - t_{\text{comb_max}} - t_{\text{setup}}$.

Example 11.3

The shift register in Figure 11-14(a) is shown with unbalanced buffer delays on the clock distribution net. The simulation results in Figure 11-14(b) compare the output of a register with equal delays and with unbalanced delays. In the register with unbalanced delays, a sample of *D_in* passes through the register with clock skew in three cycles instead of in four cycles.

End of Example 11.3

Example 11.4

The clock divider in Figure 11-15(a) creates *clock_by_2* from *clock*, but the delay of the buffer skews *clock_by_2* relative to *clock*. The recommended circuit [5] in Figure 11-15(b) delivers a common clock to the flip-flops and eliminates a buffer.

End of Example 11.4

Clock distribution networks for ASICs must be designed carefully to achieve short transition times and to minimize the effect of skew. Equalizing the delays from

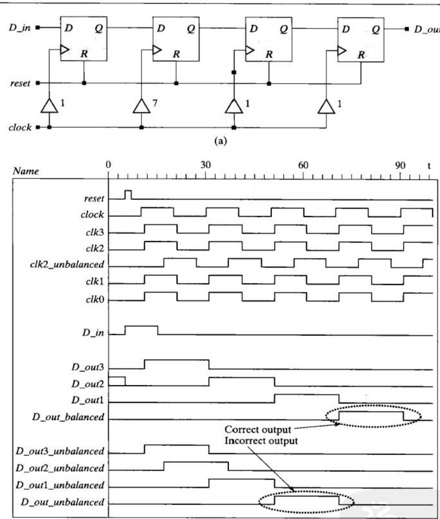


FIGURE 11-14 Effect of clock skew: (a) shift register with unbalanced clock distribution, and (b) waveforms showing incorrect register operation.

the clock pads to all memory elements, as shown in Figure 11-16, minimizes skew. Special compilers are used to place clock trees within a layout. Balanced clock trees achieve a balanced physical and temporal topology, leading to synchronous clock edges. Otherwise, the edges of the clocks that are farthest from the source of the

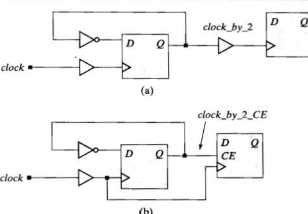


FIGURE 11-15 Alternative circuits that implement a clock divider: (a) *clock_by_2* will have skew relative to clock and (b) a common clock synchronizes both flip-flops, without skew.

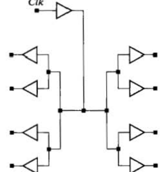


FIGURE 11-16 A clock tree balances the delay from the source of a clock signal to its destinations.

clock will occur later than the edges that are closest to the source. Clock trees (also called H-trees) are designed to balance the delays through the tree and to reduce the peak currents at the load stage [4]. The trees of multiple clocks should be routed to have similar loads to minimize the skew between the clock phases. Each branch should have a balanced load. A testbench can monitor the signals shown in Figure 11-17 to verify that they do not drift too far apart; static analysis can be performed to determine the arrival times of the clock signals at the endpoints of the clock tree to ensure balance.

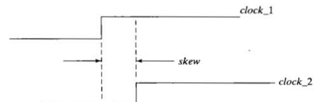


FIGURE 11-17 The skew between two signals in a design can be monitored by a testbench to detect a violation of a maximum skew constraint.

11.3 Elimination of ASIC Timing Violations

Table 11-2 summarizes options that can be exercised to eliminate timing violations in an ASIC. The simplest option is to lengthen the clock cycle. This approach can eliminate the timing violation if the maximum period of the clock is unconstrained. Otherwise, an alternative approach is to reroute the critical paths of the circuit to reduce their net delays. The devices along a path can be resized to reduce the path delay, and to use flip-flops having shorter setup and hold times, subject to the availability of parts in the cell library. In addition to the options shown in Table 11-2, the behavioral model should be reexamined to see whether it can be modified to synthesize into faster logic. For example, replacing ***if . . . else*** statements by ***case*** statements might lead to parallel logic; state codes might be changed (e.g., use one-hot coding), with the result that the synthesized circuit is faster.

There are fewer structural options for eliminating timing violations in a field-programmable gate array (FPGA), because its architecture is fixed. However, FPGAs include fast carry logic (to improve timing margins) and special clock buffer nets and delay-lock loops that are intended to minimize the clock skew in the design. FPGAs

TABLE 11-2 Options for eliminating timing violations in an ASIC.

Elimination of ASIC Timing Violations	
Action	**Effect**
Lengthen the clock cycle	Eliminates the violation, constrained by performance specfications
Reroute critical paths	Reduced net delays
Resize and substitute devices	Reduced device delays and improved setup and hold margins
Redesign the clock tree	Reduced clock skew
Substitute a different algorithm	Reduced path delays (e.g., carry look-ahead vs. ripple carry)
Substitute architectures	Reduced path delays (e.g., pipelining)
Change technologies	Reduced device and path delays

are rich in registers, so pipelining is an attractive option for increasing the throughput of multilevel combinational logic. I/O flip-flops in FPGAs have guaranteed setup, hold, and clock-to-output times, with programmable input delay and programmable output slew. For example, the programmable input delay of the Xilinx parts guarantees that the device will have 0 hold time, but an extended setup time. The software tools for synthesizing a design into an FPGA attempt to satisfy I/O and internal timing constraints. If the constraints cannot be met for a given device the only option is to lengthen the clock period or substitute a device that has a higher-rated clock frequency.

Error-free synchronous operation is the initial focus of timing verification. Setup constraints, hold constraints, and pulsewidth constraints are imposed on the design by the physical storage elements. Constraints on the clock skew are imposed by the interplay of the setup constraints, the longest path delay, and the physical layout and distribution of the clock through the circuit. STA can determine whether a circuit has sufficient slack to satisfy setup conditions, for example. Testbenches can monitor other conditions too, such as glitches and the relative skew between signals (e.g., the skew between the control lines of a memory). STA can analyze the distribution of path lengths, which can be used as a guide to determine whether device sizes can be reduced without violating the timing constraints.

We saw in Chapter 6 that nested ***if. . . else*** statements imply priority and will synthesize priority encoders. If this structure cannot be avoided, the timing-critical signals should be put in the first clause of the statement, because it will drive the final stage of logic and will have the shortest path to the output. ***case*** statements synthesize to smaller and faster circuits, but ***if. . . else*** statements offer more flexibility, implement priority logic, and might be needed to accommodate late-arriving signals. Nested ***if. . . else*** and nested ***case*** statements synthesize to multiple levels of logic, which might compromise performance.

11.4 False Paths

Not all physical paths in a circuit can be exercised. Static path analysis can produce a false alarm if the tool ignores the true functionality of a path.

Example 11.5

The path *In_1–w0–w1–w2–Out_0* in Figure 11-18 cannot be sensitized, because the conditions that sensitize the path through the gates that drive *w0* and *w1* also drive a 1 onto the output of the AND gate that drives the inverter. This condition constrains the output of the inverter to be 0, a condition that blocks the signal path by desensitizing the AND gate that drives *w2*. Note that *In_2* has two paths to *w2*, a condition known as reconvergent fanout. When a circuit has reconvergent fanout the path through the gate at which the signal reconverges might not be statically sensitizable, *because the side inputs of the gate cannot be sensitized independently of the signal that is to propagate through the gate.*

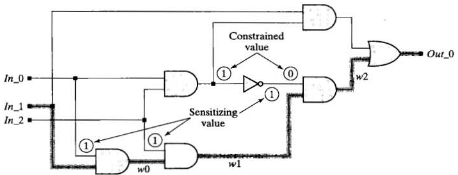


FIGURE 11-18 The path *In_1-w0-w1-w2-Out_0* in cannot be statically sensitized.

End of Example 11.5

The delay of a topological path may be reported incorrectly if the analysis does not account for the polarity of signal transitions along the path. In general, merely adding the maximum or minimum delay values of the gates along a path may produce a pessimistic calculation of the maximum or minimum path delay and lead to wasted effort to eliminate a timing violation. A timing analyzer must use the appropriate rising or falling delay of the devices when calculating the delay along a path.

Example 11.6

If the analysis of the delays along the path shown in Figure 11-19 simply adds the maximum values of the delays through the gates, the maximum rising delay is $t_{delay_rising} = 15$. Considering the polarity of the transitions gives $t_{max_rising} = 5 + 3 + 5 = 13$.

End of Example 11.6

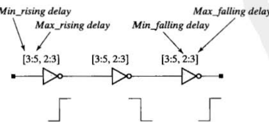


FIGURE 11-19 The maximum delay of a rising edge transition through the inverter path must account for the polarity of the signal transitions along the path.

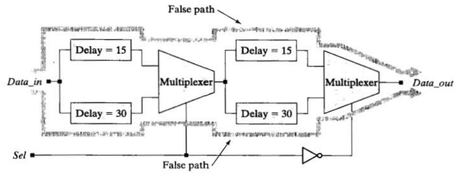


FIGURE 11-20 A circuit with false paths.

A topological path is said to be a false path if it is not a functional path under the action of the primary inputs. For example, mutually exclusive conditions that steer datapaths might lead to false paths being reported by a timing analyzer. The steering logic eliminates certain paths, which should not be reported by the tool.

Example 11.7

The controlling signals of the multiplexers that steer the datapaths in Figure 11-20 are not independent of each other. Two false paths cannot be exercised. The maximum topological delay through the circuit is $t_{topological} = 30 + 30 = 60$, but the maximum functional delay $t_{max} = 15 + 30 = 45$.

End of Example 11.7

A timing analyzer might report false path delays if redundant logic has been added to the design to reduce path delay. If the timing analyzer does not account for the reduced delay attributable to the redundant logic the reported delays will be pessimistic. Also, multicycle paths might trigger a false alarm unless the timing analyzer can identify such paths.

11.5 System Tasks for Timing Verification

Verilog has several built-in system tasks for performing timing checks during simulation. Some of these tasks can be included in models to (1) monitor simulation activity automatically, (2) detect a timing violation, and (3) report a timing violation.[12]

[12]Timing checks that are specific to a device are usually included in the standard cell library model of the device by including a ***specify . . . endspecify*** block in a module.

11.5.1 Timing Check: Setup Condition

An edge-triggered flip-flop will not operate correctly if the data at its input is not stable for a sufficient time before and after the active edge of the clock. Setup and hold times are logic-level constraints on the operation of storage elements. System failure can result from nondeterministic behavior of storage elements if the setup and hold times of the storage elements are violated. Figure 11-21 shows the setup interval before each active edge of the clock. The system task for detecting violations of the setup time of a device has the following syntax: ***$setup*** (*data_event, reference_event, limit*).

A setup time violation occurs when *data_event* is unstable within the specified time *limit* relative to the *reference_event*. In practice, data must be stable before the active edge of the clock of a flip-flop.

Setup-time violations are cause by paths that are long relative to the clock cycle. The delays that contribute to the late arrival of the data must be reduced, or the clock period must be increased to eliminate the timing violation (see Table 11-2).

Example 11.8

Figure 11-22 shows the waveforms of *sys_clk*, *sig_a*, and *sig_b*. Note that *sig_a* satisfies the setup timing constraint, but *sig_b* does not. The timing check would be activated by the tasks ***$setup*** (*sig_a*, ***posedge*** *sys_clk*, 5) and ***$setup*** (*sig_b*, ***posedge*** *sys_clk*, 5); the latter check would report a timing violation.

End of Example 11.8

11.5.2 Timing Check: Hold Condition

For correct operation of a flip-flop, the data at its input must be stable for a sufficiently long time after the active edge of its clock. A hold violation occurs if the datapath to the flip-flop is so short that a change in the data at the output of the flip-flop at the startpoint of the path propagates too quickly to the input of the flip-flop at the endpoint of the path. Figure 11-23 shows the hold interval in which the data path of a flip-flop must be stable.

Short paths through combinational logic are stretched automatically by synthesis tools to reduce the slack and meet timing constraints. It is desirable to achieve balance

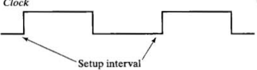


FIGURE 11-21 The data of a flip-flop must be stable during the setup interval located ahead the active edge of the clock.

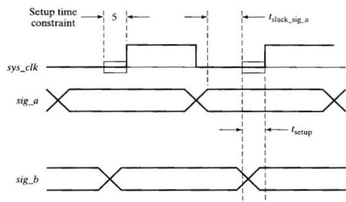


FIGURE 11-22 The setup time constraint on *sys_clk* requires *sig_a* and *sig_b* to be stable during the setup interval located ahead of the active edge of the clock. *sig_b* violates the setup constraint.

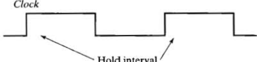


FIGURE 11-23 The data of a flip-flop must be stable during the hold interval located after the active edge of the clock.

in a design, so that paths are not faster or slower than necessary. Needlessly fast paths waste silicon area.

The system task for detecting violations of the *hold time* of a device has the following syntax: ***$hold*** (*reference_event*, *data_event*, *limit*). A hold time violation occurs when *data_event* is unstable within the specified time *limit* relative to the *reference_event*.

Example 11.9

For the waveforms in Figure 11-24, *sig_a* is not stable during the hold interval of *sys_clock*. A timing violation would be reported by the task ***$hold*** (**posedge** *sys_clk*, *sig_a*, 5).

End of Example 11.9

11.5.3 Timing Check: Setup and Hold Conditions

The timing task ***$setuphold*** monitors for violations of both setup and hold constraints between a *reference_event* and a *data_event*, according to the syntax: ***$setuphold***

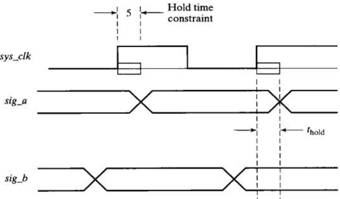


FIGURE 11-24 The hold time constraint on *sys_clk* is violated by *sig_a*.

(*reference_event*, *data_event*, *setup_limit*, *hold_limit*), where *reference_event* is the edge transition that synchronizes the device.

Example 11.10

In Figure 11-25 *sig_a* and *sig_b* satisfy both the setup and hold constraints.

End of Example 11.10

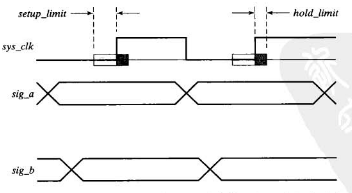


FIGURE 11-25 The setup and hold constraints are satisfied by *sig_a* and *sig_b* relative to the active edges of *sys_clk*.

11.5.4 Timing Check: Pulsewidth Constraint

The minimum width of a clock pulse is constrained for sequential devices. The clock of an edge-triggered device must endure for a time sufficient to charge internal signal nodes. The task ***$width*** (*reference_event, limit*) will detect a violation of the minimum pulse width (t_{MPW}). For example, the task ***$width*** (**posedge** *clk*, 4) will detect a violation if the interval between a positive edge of *clk* and the immediately following negative edge is less than 4. This task can also be used in simulation to detect potential glitches and degraded (narrow) clock pulses.

Example 11.11

The minimum pulsewidth constraint in Figure 11-26 is satisfied by *clock_a*, but violated by *clock_b*.

End of Example 11.11

11.5.5 Timing Check: Signal Skew Constraint

Clock skew is a critical issue in system performance. It results from unbalanced clock trees and degrades setup and hold time margins. The skew between two signals is monitored by the task ***$skew***(*reference_event, data_event, limit*); it will report a violation if the interval between *reference_event* and *data_event* is greater than *limit*.

Example 11.12

$skew(**posedge** *clk1*, **posedge** *clk2, 3*) detects a violation if the interval between the rising edge of *clk1* and the rising edge of *clk2* exceeds 3, as shown in Figure 11-27.

End of Example 11.12

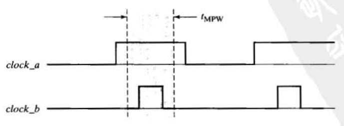


FIGURE 11-26 The task ***$pulsewidth*** will detect a pulse width violation by *clock_b*.

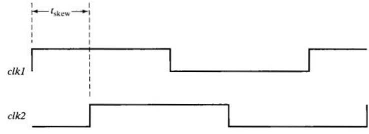


FIGURE 11-27 Skew is checked between signal edges by the system task ***$skew***.

11.5.6 Timing Check: Clock Period

The period of a clock is determined by the interval between successive active edges of the waveform. The task ***$period*** (*reference_event*, *limit*) will monitor successive edges of an edge-triggered *reference_event* and detect a timing violation error if the interval between successive events is less than *limit*. In design re-use within an SoC environment, this timing check would verify that a core cell can be used safely at the specified clock period. If the timing check is satisfied, the clock period is at least as long as the minimum period required by the cell.

Example 11.13

The task invoked by the procedural statement ***$period*** (**posedge** *clock_a*, 25) does not detect a timing violation in *clock_a*, shown in Figure 11-28.

End of Example 11.13

11.5.7 Timing Check: Recovery Time

The ***$recovery*** (*reference_event*, *data_event*, *limit*) task checks the time required for synchronous behavior to resume after a reset (asynchronous) or a clear condition has been de-asserted. The recovery check specifies the minimum time that an asynchronous input

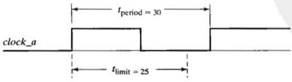


FIGURE 11-28 *clock_a* satisfies the constraint on its minimum period.

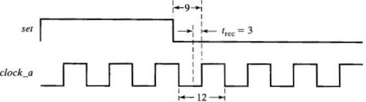


FIGURE 11-29 The interval between the falling (de-asserting) edge of *set* relative to the next rising edge of *clock_a* satisfies the constraint on the recovery time.

must be stable before the active edge of the clock. The recovery *limit* specifies the time between the referenced de-assertion event and the next active edge of the *data_event*.

Example 11.14

The task invoked by the procedural statement ***$recovery*(negedge** *set*, **posedge** *clock*, 3) checks whether the de-assertion of *set* precedes the active edge of *clock* by three time steps in Figure 11.29.

End of Example 11.14

11.6 Fault Simulation and Manufacturing Tests

A circuit that has been designed properly may still fail to operate in the field, where it is expensive to diagnose and repair. The failure of a circuit might be permanent, intermittent, or transient. A *permanent failure* exhibits incorrect operation of the circuit at all times. An *intermittent failure* is exhibited randomly, and for only a finite duration. Transient failures occur in the presence of certain environmental conditions that alter the performance characteristics of the devices, such as high temperature or background radiation. The failure might be caused by any of the following: (1) wafer defects, (2) contaminated atmosphere in the clean room (i.e., ambient particulates), (3) impure processing gasses, water, and chemicals, and (4) photomask misalignment.

A process-induced (i.e., manufacturing) defect might cause a high junction leakage current, a high contact resistance, an open circuit, a short circuit, or an out-of-spec threshold voltage. Our concern will be defects that cause failures that are apparent as functional errors during operation.

Contaminants in the clean-room environment can introduce defects in the manufactured circuit. The atmosphere of the clean room might contain particles of dust, smoke, shaving lotion, or other gaseous products. The wafer might bear residues

of materials that were not removed properly by a cleaning process, and the chemicals used in the fabrication process might not be pure enough. Clean rooms are constructed to purify and maintain a clean atmosphere, but the presence of equipment operators and technicians in the room inevitably introduces contaminants. The floor of a clean room is specially constructed to minimize the possibility that vibrations will cause a photomask to be misaligned, which would cause the devices to be exposed improperly and violate spatial constraints on the layout.

The consequences of device failure are so expensive that the semiconductor industry tests devices thoroughly before they are shipped. This testing does not revalidate the functionality of the device or test for design errors, because design errors are assumed to have been found during the verification steps that are executed in the design flow before the mask set is released for fabrication. In fact, it is likely that the set of stimulus patterns that verify the functionality of a device will detect only a few defects within the fabricated circuit. Failure analysis of defects might point to a design error, but that is not its purpose.

Production testing applies test signals and monitors the response of a circuit to identify the presence of defects. The set of test patterns (vectors) that are applied to a circuit must be sufficient to test the known failure modes of the chip and detect all defects. The vectors that are used to verify the functionally of the design are usually a starting point for production testing, but they are not sufficient if they fail to exercise some failure modes of the circuit. It is unlikely that the vectors used for design validation will provide sufficient coverage of a complex circuit (e.g., one with an embedded processor, memory controller, and perhaps an embedded DSP, and having about 500,000 to 1,000,000 gates).

Production testing is concerned with detecting permanent errors that are caused by manufacturing defects. It involves two major steps: test generation and fault simulation. We will examine how tests are generated, but first we must consider *what* we are testing, before we determine *how* to test it. The tests used in production testing are intended to detect failure modes, called *faults*, within the circuit. The set of test vectors is generated in conjunction with a process called *fault simulation*, which determines whether a test detects a fault at a particular site in the circuit. Fault coverage provides a measure of the quality of a set of test patterns.

11.6.1 Circuit Defects and Faults

Models of the failure modes of a circuit consider the *logical effects* that result from the *physical faults* in a circuit. The main physical faults that can occur are summarized in Table 11-3, which is based on a discussion in Smith [4]. Physical defects and the logical faults they cause are assumed to be uniformly distributed over a wafer.

When a circuit fails to operate correctly, the logic realized by the circuit is different from the logic that was specified for the design. There are a variety of modes by which a digital circuit may exhibit failure. One common failure mode occurs when a signal line is shorted to either the power rail or the ground rail. These failure modes are called "stuck" faults, and their location is called a *fault site*. A complementary metal-oxide semiconductor (CMOS) circuit has a *stuck-on* fault if the gate of a transistor is always on. Such defects in

TABLE 11-3 Physical faults and circuit-level effects in an integrated circuit.

Physical Faults	Degradation Fault*	Open-Circuit Fault	Short-Circuit Fault
Chip-Level Fault			
Leakage or short between package leads	x		x
Broken, misaligned, or poor wire bonding		x	
Surface contamination, moisture	x		
Metal migration, stress, peeling		x	x
Metallization (open or short)		x	x
Gate-Level Fault			
Contact opens		x	
Gate to source/drain junction short	x		x
Field-oxide parasitic device	x		x
Gate-oxide flaw, spiking	x		x
Mask misalignment	x		x

*Parametric fault (threshold voltage shift) or delay fault.

the manufactured part can occur when physical vibrations of the mask aligner occur during the photomasking steps, causing the conductor of a signal path to be misaligned and placed too close to the conductor of a neighboring signal path, or when over/under etching of the photoresist alters the physical location of a conductor. The resulting defects can also occur when the fabrication process does not operate within the range of statistical specifications that reduce the possibility of such failures.

Faults due to short-circuits occurring in the interconnect between transistors in a logic cell are called *bridging faults*. Bridging faults are detected by measuring the quiescent current, I_{DDQ}, through a circuit. Testing the quiescent current takes more time than tests for stuck faults, but tests for I_{DDQ} are needed for bridging faults, because the tests that reveal stuck faults are not intended to reveal a bridging fault, which are manifested by a high quiescent current rather than by an incorrect logic value.

Example 11.15

Figure 11-30(a) illustrates how a bridging fault in a two-input CMOS NOR gate connects the gates of the pull-up transistors. Ordinarily, a CMOS device will have a short between the power and ground rails of the circuit for only a brief time while the output of the device switches. The quiescent current through the device, I_{DDQ}, is 0, but the bridging fault in the pull-up logic of this circuit will cause a high current to flow

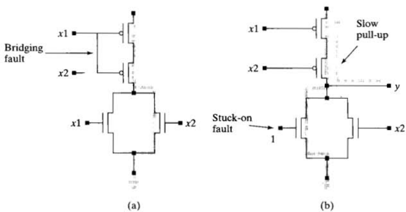


FIGURE 11-30 A two-input NOR gate with (a) a bridging fault and (b) a **stuck-on** fault.

between the rails when $x1 = 0$ and $x2 = 1$, causing thermal damage and early mortality of the device. I_{DDQ} testing, which monitors the quiescent current of the circuit, would detect the presence of the bridging fault [6].

The two-input NOR gate in Figure 11-30(b) has an internal defect in which the pull-down transistor for *x1* is stuck on, with *x1* effectively 1. Because the signal applied to the pull-up gate for *x1* cannot be exercised independently of the signal attached to the pull-down gate for *x1*, this condition cannot be detected externally by testing the logic of the circuit. Note that if $x1 = 0$ and $x2 = 0$ then $y = 1$, but the pull-down transistor with the stuck-on gate tends to pull *y* to 0. The stuck-on fault causes a current that always tends to discharge the output node of the device, making the pull-up action slower than that of a fault-free circuit. In addition, the high current through the pull-down transistor will lead to thermal degradation and early failure of the circuit.

End of Example 11.15

Faults that are caused by shorts of nets to the power or ground rails of the circuit are called *stuck at 1* or *stuck at 0 faults*, respectively. A given node in a circuit may be stuck at logical 1, denoted by *s-a-1*, or stuck at 0, denoted by *s-a-0*. The presence of a *s-a-1* or *s-a-0* fault compromises the logic implemented by the circuit.

Example 11.16

The three-input NAND gate in Figure 11-31 is shown in (a) without a fault, in (b) with an *s-a-0* fault on one input, and in (c) with an *s-a-1* fault on one input. The

FIGURE 11-31 A three-input NAND gate with (a) no faults, (b) *x1* with an *s-a-0* fault, and (c) *x1* with an *s-a-1* fault.

logic of the gate in the fault-free circuit has $y_{good} = (x1\ x2\ x3)'$, but the circuit with the fault *x1 s-a-0* has $y_{x1\ s\text{-}a\text{-}0} = 1$ its output is stuck at 1. The circuit with *x1* stuck at 1 has $y_{x1\ s\text{-}a\text{-}1} = (x2\ x3)'$ (i.e., the circuit realizes a two-input NAND gate). The logic realized by the circuits having a fault differs from the logic of the fault-free circuit.[13]

End of Example 11.16

All of the nets attached to the inputs and outputs of the gates in a logic circuit are possible sites for *s-a-0* and *s-a-1* faults. Stuck-at faults cause the disappearance of a literal or an implicant in the realized logic, or force the logic function to a fixed value. A test must detect the difference between the logic of the fault-free circuit and the logic of the circuit having an *s-a-0* or *s-a-1* fault. The *s-a-0* fault in Figure 11-31(b) could be detected by a test that applied $x1 = 1, x2 = 1$ and $x3 = 1$. If $y_{x1=1,\,x2=1,\,x3=1} = 1$ then the test reveals the presence of the fault, because $y_{x1=1,\,x2=1,\,x3=1} = 1$ does not match $y_{good} = 0$. Similarly, the *s-a-1* fault in Figure 11-31(c) will be detected by a test that applies $x1 = 0$ and $x2 = 1$ and $x3 = 1$ because $y_{x1=0,\,x2=1,\,x3=1} = 0$ does not match $y_{good} = 1$ if the fault is present.

The faults in the isolated three-input NAND gate in Figure 11-31 can be tested easily by applying signals directly to the inputs of the gate, and the output can be monitored directly too. This is not the typical situation, because the fault sites at the inputs and the outputs of most of the gates in a circuit are not directly available at the chip's pins. Moreover, a chip is limited to at most a few hundred pins, but may have millions of fault sites within the device. Nonetheless, the same principles that were used to test the isolated device can be used to develop tests to detect faults at sites embedded within a chip. An embedded fault can be detected if (1) primary inputs can assert a logic value at the site of the fault, and (2) those inputs are compatible with the inputs that propagate the fault to a primary output of the device.

[13]Note that the internal bridging fault in Figure 11.31(b) affects the gate of the pull-down transistor driven by *x1* and does not affect the corresponding pull-up transistor. A stuck fault at *x1* affects both transistors.

11.6.2 Fault Detection and Testing

If a *s-a-0* or *s-a-1* fault in a combinational logic circuit is testable, it can be detected by applying a set of inputs, known as a *test pattern*, and observing whether the output of the circuit with the fault differs from that of a fault-free circuit. A basic scheme for testing circuits for stuck faults is shown in Figure 11-32. Identical input signals are applied to a fault-free circuit, and to a circuit in which a fault has been injected at a particular site. The outputs of the circuits are compared to determine whether they differ. If the error signal asserts, the applied pattern is a test for the injected fault. If the outputs of the circuits do not differ, the applied pattern cannot distinguish between the faulty circuit and the fault-free circuit.

Example 11.17

A test for the fault *x1 s-a-0* in the circuit in Figure 11-33(a) is shown in the table in Figure 11-33(b). When the stimulus pattern is applied to the inputs of the circuit, the outputs of the good and faulty circuits differ, so the stimulus pattern (*x*1 2 3) = (1 1 1) detects the fault and is a test for *x1 s-a-0.*

End of Example 11.17

Note that if *x1* is stuck at 0 the output of the circuit in Figure 11-33 has a value of 1, independently of the inputs. In general, test for a fault by applying a pattern that asserts the correct (not stuck) value at the fault site, and compare the outputs of the circuit with and without a fault. If the outputs do not match, the stimulus pattern will detect such a fault in the manufactured circuit.

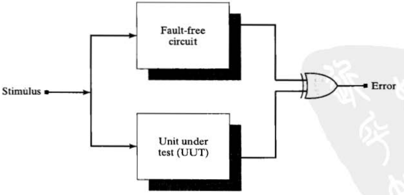
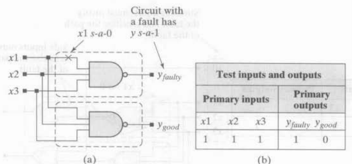


FIGURE 11-32 Fault simulation compares the response of a fault-free circuit to the response of a circuit with a fault.



Test inputs and outputs				
Primary inputs			Primary outputs	
x1	x2	x3	y_{faulty}	y_{good}
1	1	1	1	0

FIGURE 11-33 A test for *x1 s-a-0* detects a difference between the faulty circuit and the fault-free circuit.

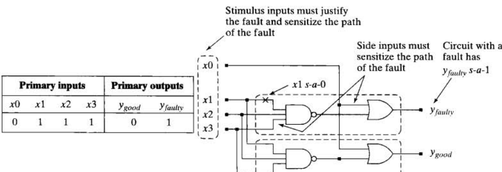
A test for a *s-a-0* and a *s-a-1* fault in combinational logic requires a single input pattern, and consists of the following four steps:

(a) Inject a fault in the circuit.
(b) Justify the fault (i.e., apply inputs that complement the value of the fault at the fault site).
(c) Sensitize one or more paths to propagate the effect of the fault to an output.
(d) Compare the sensitized output to that of a good circuit.

To test for a fault in a combinational logic circuit, the primary inputs of the circuit must be chosen to (1) counter-assert (justify) the assumed value of the fault, and (2) propagate the effect of the fault to a primary output (i.e., sensitize the output to the fault). If the test is to detect the condition that a node is stuck at 1, the primary inputs must be those that would assert the value of 0 at the fault site in a fault-free circuit. The primary inputs must also propagate a signal value from the site of the fault to a primary output. If the sensitized output differs from the output of the fault-free circuit the test detects a fault. The test pattern that detects a fault does not necessarily isolate the location of the fault to a particular site, because a test might detect more than one fault.

Example 11.18

Two copies of a circuit are shown in Figure 11-34, with the so-called faulty circuit having the fault *x1 s-a-0*. To detect the fault, the stimulus pattern has the value of *x1* set to 1 to justify the fault. The side inputs of the NAND gate (*x1*, *x2*, *x3*) are set to 1 to *sensitize* the output of the gate to the value of *x1*. The side input of the OR gate (*x0*) is set to 0 to sensitize the path propagating the value of *x1* to the primary output of the circuit. The fault-free circuit has $y_{good} = 0$; the circuit with *x1 s-a-0* has $y_{faulty} = 1$. The stimulus pattern (*x0 x1 x2 x3*) = (0 1 1 1) is a test for the fault *x1 s-a-0*.



Primary inputs				Primary outputs	
$x0$	$x1$	$x2$	$x3$	y_{good}	y_{faulty}
0	1	1	1	0	1

FIGURE 11-34 The stimulus pattern ($x0$ $x1$ $x2$ $x3$) = (0 1 1 1) detects the fault *x1 s-a-0* by justifying the fault and sensitizing the output to the fault.

End of Example 11.18

11.6.3 *D*-Notation

A special symbolic notation, called *D-notation* [7], is used in test generation and fault detection. In *D*-notation the logical values of the signals in a circuit are denoted by the symbols *D* and *D'*. The symbol *D* indicates that the value of the signal is 1 in a good (fault-free) circuit, and *D'* indicates that the value of the signal is 0 in a good circuit.

Example 11.19

In Figure 11-35, signal *x1* has the value *D* when the pattern ($x1$ $x2$ $x3$) = (1 1 1) is applied, meaning that *x1* = 1 in the fault-free circuit, and 0 in the circuit having x1 *s-a-0*.

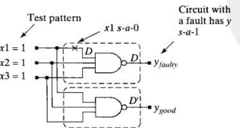


FIGURE 11-35 The *D*-notation indicates that a net in a fault-free circuit has the value 1 and has a value 0 in the circuit with a fault.

Likewise, the output of the fault-free circuit has $y_{good} = D'$. The symbolic output of the faulty circuit has $y_{faulty} = D$.

End of Example 11.19

The *D* notation for faults is useful in identifying faults that can be detected by the same stimulus pattern, and reducing the number of tests that must be applied to the circuit. It is desirable to have a high number of faults detected by a small number of tests.

Example 11.20

The two-level circuit in Figure 11-36(a) has faults sites at nine primary inputs, three internal nets, and one primary output. The tables in Figure 11-36(b) and 11.36(c) use *D*-notation to indicate the values of the internal and primary output nets for the indicated stimulus patterns.[14] The tables are annotated to identify the faults that are detected by the pattern. The tests are developed under the assumption that a single fault is present in the circuit (the so-called single stuck fault model [6]), but a given test may actually detect more than one fault. In this circuit, only three tests are needed to detect all of the possible *s-a-1* faults, and another three tests detect all of the *s-a-0* faults.

End of Example 11.20

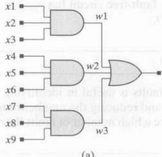
Example 11.21

When a test is applied to detect the fault *w2 s-a-0* in the multilevel combinational logic circuit in Figure 11-37(a), the signals have the values displayed in *D*-notation in table in Figure 11-37(b). Of the possible tests for the fault, the one with ($x1$ $x2$ $x3$ $x4$ $x5$) = (1x 1 xx) is shown, where x denotes a don't care. The values of the primary inputs that sensitize the fault do not conflict with the values that justify the fault, so the pattern detects the fault. The output of the fault-free circuit is $y = 1$.

End of Example 11.21

Fault sensitization in multilevel networks is accomplished by tracing a path forward from the fault site to a primary output, then setting all of the side inputs of the gates on the path to their sensitizing values (e.g., the side inputs of AND gates are set to 1). Then paths are traced backward from the side inputs through gates to the primary inputs, to establish

[14]In this context, the symbol x denotes a don't-care.



(a)

Tests for *s-a-1* faults						
	#1	site	#2	site	#3	site
x1	0	x1	1		1	
x2	1		0	x2	1	
x3	1		1		0	x3
x4	0	x4	1		1	
x5	1		0	x5	1	
x6	1		1		0	x6
x7	0	x7	1		1	
x8	1		0	x8	1	
x9	1		1		0	x9
w1	D'	w1	D'	w1	D'	w1
w2	D'	w2	D'	w2	D'	w2
w3	D'	w3	D'	w3	D'	w3
y	D'	y	D'	y	D'	y

(b)

Tests for *s-a-0* faults						
	#1	site	#2	site	#3	site
x1	1	x1	0		x	
x2	1	x2	x		x	
x3	1	x3	x		0	
x4	0		1	x4	x	
x5	x		1	x5	x	
x6	x		1	x6	0	
x7	0		0		1	x7
x8	x		x		1	x8
x9	x		x		1	x9
w1	D	w1	D'		D'	
w2	D'		D	w2	D'	
w3	D'		D'		D	w3
y	D	y	D	y	D	y

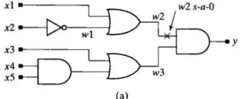
(c)

FIGURE 11-36 All of the *s-a-0* and *s-a-1* faults in the circuit in (a) are detected by the patterns in (b) and (c).

values of the primary inputs that sensitize the path from the site of the fault to the primary output. This step is called *line justification*. In general, multipath sensitization must be considered [6], because single-path sensitization might not produce a test for a testable fault.

Example 11.22

The fault *G2 s-a-1* in the circuit in Figure 11-38, known as Schneider's circuit [6], can be justified by setting $x2 = 1$ and $x3 = 1$. To sensitize a single path from the



(a)

	Justification	Sensitization	Test
*x*1	1	*x*	1
*x*2	*x*	*x*	*x*
*x*3	*x*	1	1
*x*4	*x*	*x*	*x*
*x*5	*x*	*x*	*x*
*w*1	*x*	*x*	
*w*2	*D*	*x*	
*w*3	*x*	1	
y	*D*	*D*	

(b)

FIGURE 11-37 Fault detection: (a) a combinational logic circuit and (b) signal values in *D*-notation for justification and sensitization of the fault *w2 s-a-0*.

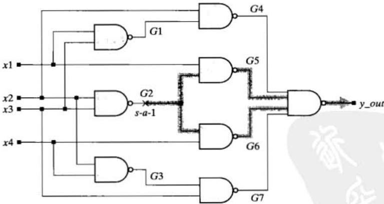


FIGURE 11-38 The fault *G2 s-a-1* cannot be sensitized by a single path.

fault site to *y_out*, set $x1 = 1$, which drives the output of *G5* to 1. The previous choices for *x1*, *x2*, and *x3* are compatible with setting to 1 the side inputs of the NAND gate forming *y_out*, but setting *G6* to 1 requires setting *x4* to 0, because the value of *G2* is not known. With $x3 = 1$ and $x4 = 0$, the value of *G7* becomes 0,

which desensitizes the path of the fault. If the single path through *G6* is chosen to propagate the fault, a similar condition blocks the propagation of the fault to the output. Consequently, the fault *G2 s-a-1* cannot be detected by sensitizing a single path from the fault site to the output of the circuit. The fault can be detected only by simultaneously sensitizing the paths through both *G5* and *G6*, with the stimulus pattern ($x1\ x2\ x3\ x4$) = (1 1 1 1).

End of Example 11.22

11.6.4 Automatic Test Pattern Generation for Combinational Circuits

It is not feasible to develop tests for faults in large combinational circuits by applying manual methods. The test for the fault *w2 s-a-0* in Figure 11-37 could be developed by manual methods because the circuit was simple. An alternative approach to testing combinational logic is to apply all possible inputs to the circuit and observe whether the outputs of the circuit match those of the fault-free circuit. Although exhaustive test-pattern generation is straightforward, it becomes unwieldy when a circuit has a large number of inputs, and it could be unnecessary. We saw in Example 11.20 that only six patterns were needed to detect the entire set of faults. Exhaustive pattern generation would have produced 512 patterns.

Algorithms exist for automatically and efficiently finding/constructing a small set of test patterns that will detect most, if not all, of the faults in a combinational circuit. Commercial tools for test generation incorporate various features of a variety of algorithms. The *D*-algorithm [7], which uses the *D*-notation discussed above, is widely used. It tests for stuck faults at all signal lines embedded in a combinational circuit. The *D*-algorithm has been incorporated in software for automatic test pattern generation (ATPG). It uses multipath sensitization and is guaranteed to find a test for a logical fault in a combinational circuit if a test exists, but its efficiency degrades if the circuit contains a large number of exclusive-or gates.

Two alternative algorithms, PODEM (path-oriented decision making) [8], and FAN (fanout–oriented test-generation algorithm) [9], are more efficient than the *D*-algorithm. The PODEM algorithm uses forward implication from a circuit's primary inputs to replace the alternating backtracing and forward propagation steps of the *D*-algorithm. The FAN algorithm uses additional strategies to reduce backtracing and is even more efficient than PODEM. See Abramovici et al. [6] and Fujiwara and Shimono [10] for details of these algorithms.

ATPG seeks a stimulus pattern that will detect a given fault, but can fail to produce a test for a combinational circuit that has reconvergent fanout. Some faults might not be detectable. Untestable faults are due to (1) redundant logic (see Problem 11.14),[15]

[15]Although synthesis tools remove redundant logic, a circuit may be modified after synthesis to add redundant logic to improve the speed of the circuit or to cover hazards. Such modifications are problematic for test generation because they can reduce the efficiency of the effort to generate tests.

(2) uncontrollable nets, and/or (c) unobservable nets. A fault cannot be tested if the net on which the fault is located cannot be controlled or if no output paths can be sensitized to observe the fault.

Example 11.23

A test for the fault *w2 s-a-0* in Figure 11-39 does not exist. The signal *x5* has reconvergent fanout at *y*, so the side inputs affected by *x5* cannot be set independently. Sensitization of *w3* requires setting *x5* to 1, but sensitization of *y* requires setting *x5* to 0. Table 11-4 shows the signal values in *D*-notation and shows the conflicts between the values of the primary inputs required to sensitize *w3* and *y*. The reconvergent fanout of *x5* forces *y* to 0,

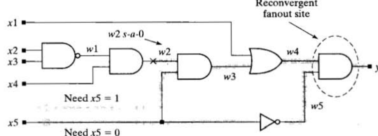


FIGURE 11-39 A test for the fault *w2 s-a-0* does not exist. The reconvergent fanout of *x5* leads to conflicting conditions to test the fault *w2 s-a-0*.

TABLE 11-4 The reconvergent fanout causes a conflict between the values of *x5* required to sensitize the fault *w2 s-a-0*.

	Justification	**Sensitization**	**Test**
x1	*x*	0	0
x2	0	*x*	0
x3	*x*	*x*	*x*
x4	1	*x*	1
x5	*x*	1 for *w3* 0 for *y*	conflict
w1	1	*x*	
w2	*D*	*D*	
w3	*x*	*D*	
w4	*x*	*D*	
w5	*x*	conflict	
y	*x*	0	

independently of the fault *w2 s-a-0*. A test cannot distinguish between the faulty circuit and the good circuit.

End of Example 11.23

11.6.5 Fault Coverage and Defect Levels

Testing for faults ensures the quality of the parts that are shipped to a customer. The probability W of shipping a defective part is related to the test coverage T and the fractional manufacturing yield Y by the expression:

$$W = 1 - Y^{(1-T)}$$

where Y represents the fractional yield of the ASIC manufacturing process that produced the chips (e.g., $Y = 0.75$), and T is the fraction of faults (i.e., coverage) that have been tested (by fault simulation) [10,11]. It is desirable to have high yield and high coverage. Table 11-5 and Figure 11-40 show how the average defect level depends on the coverage of the faults, for a given process yield. The curves provide a quantitative measure of the payoff of achieving an incremental improvement in coverage. It is commonplace for semiconductor vendors to ensure that their fault coverage exceeds 99% of the faults in a circuit.

11.6.6 Test Generation for Sequential Circuits

It can be very difficult to find direct tests for a sequential circuit, because the test might require a long sequence of inputs to drive internal sequential devices to a known state that justifies a fault site and/or sensitizes a path. It is impractical to verify that two sequential circuits have the same functionality (i.e., are equivalent, by applying input

TABLE 11-5 The quantity of undetected defective parts depends on the process maturity and the test vector coverage. Less mature technologies require significantly higher fault coverage to reduce the average defect level.

		Test Vector Coverage		
		70%	90%	99%
	Process Yield	**% Defects Undetected**		
Advanced Technology	10%	50%	21%	2%
Maturing Technology	50%	19%	7%	0.7%
Mature Technology	90%	3%	1%	0.1%

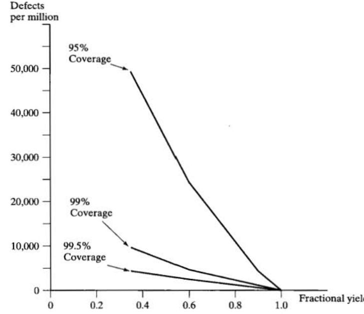


FIGURE 11-40 Defects per million parts versus test coverage and process yield.

sequences and observing output sequences). An alternative is to treat a sequential circuit as an iterative network.

Example 11.24

The circuit in Figure 11-41 requires a sequence of three stimulus patterns to detect the fault *w1 s-a-1*. The patterns are developed in reverse order, beginning with the values that the nets must have in cycle 3, the cycle in which the effect of the fault is to be observed at *y*, and working backward in time to determine the values that must exist in cycle 2, etc. The effect of the fault can be observed in the last cycle. Table 11-6 shows the values of the signals in each cycle of the test.

End of Example 11.24

Because direct test generation for a sequential circuit can be very difficult, it is usually avoided in favor of scan-path methods. The electronics industry uses scan methods to modify a circuit and make it testable by methods that apply to combinational circuits.

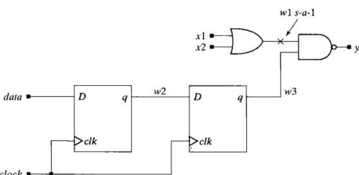


FIGURE 11-41 A sequence of three patterns detects the fault *w1 s-a-1*.

TABLE 11-6 Test-pattern sequence for testing the fault *w1 s-a-1* in Figure 11-42. The test pattern must propagate a 1 through the shift register to sensitize the output *y* to the fault.

		Justification	Sensitization	Test
Cycle 3	*w1*	*D'*	*D'*	
	w2	*x*	*x*	
	w3	*x*	1	
	x1	0	*x*	0
	x2	0	*x*	0
	data	*x*	*x*	*x*
	y	*D*	*D*	*D*
Cycle 2	*w1*	*x*	*x*	
	w2	*x*	1	
	w3	*x*	*x*	
	x1	*x*	*x*	*x*
	x2	*x*	*x*	*x*
	data	*x*	*x*	*x*
	y	*x*	*x*	x
Cycle 1	*w1*	*x*	*x*	
	w2	*x*	*x*	
	w3	*x*	*x*	
	x1	*x*	*x*	*x*
	x2	*x*	*x*	*x*
	data	*x*	1	1
	y	*x*	*x*	*x*

There are various options to the scan-path method, depending on the extent to which the registers in the machine are linked in a scan chain. An ASIC implements a partial- or a full-scan method depending on whether some or all of its internal flip-flops are replaced by scan cells and connected to form one or more shift registers that

are controlled by an external tester. Partial scan is implemented as a trade-off between the fault coverage provided by a scan chain and the additional logic required to implement the chain.

Scan design replaces ordinary flip-flops by scan flip-flops to form dual-port registers, called scan registers, making a circuit more controllable and/or observable. Scan registers can shift data through a serial port and can load data through a parallel port. In test mode, logic values of a test pattern are shifted into the flip-flops. The values loaded into the flip-flops drive combinational logic paths in one clock cycle, and the logic values at the destinations of those paths can be captured in parallel in the next clock cycle. The captured data can be shifted out of the register and analyzed to detect internal faults in the logic.

Figure 11-42 shows a set of dual-port scan cells connected to form a 4-bit-wide scan register. In normal operation, with $T = 0$, data is loaded in parallel from $D[3:0]$. In test mode ($T = 1$), data is shifted through the register from *x_in3_scan_in* to *y3_scan_out*. Depending on the application and the extent to which scan cells are used, the circuit can be designed to have 100% controllability and observability of its internal nodes via the serial scan path.

Full scan replaces all of the flip-flops in a design by scan cells; boundary scan places scan cells at the I/O ports of an ASIC and links them together to form a boundary scan chain used for board-level testing. The test patterns for an ASIC core with full scan are those used to test the combinational logic of the core, because the restructured circuit has the configuration shown in Figure 11-43. The test procedure for a circuit with full scan tests the circuit of the scan path and then tests the combinational logic. The scan path is tested by placing the circuit in test mode ($T = 1$) and toggling the clock $n + 1$ times to propagate a test sequence through the chain. Placing the circuit in test mode, shifting a test pattern into the scan register, and applying the primary inputs prepares the combinational logic for testing. With the test pattern in the scan register, the mode is set to normal ($T = 0$), and the response of the circuit is observed at the primary outputs and at the inputs to the scan register. The clock is again toggled to latch the parallel inputs to the scan register. Then the circuit is placed in test mode again, and the captured pattern is shifted out of the register for analysis. At the same time, another pattern is shifted into the register.

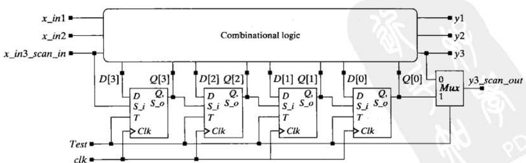


FIGURE 11-42 A scan register is formed with dual-port register cells.

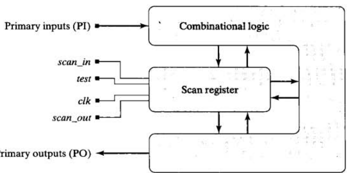


FIGURE 11-43 Circuit structure for full-scan testing.

11.7 Fault Simulation

Fault simulation compares the behavior of a circuit that has a fault with the behavior of a circuit that is fault-free. If the outputs of the two circuits are different under the application of a stimulus pattern, the pattern is said to be a test for the presence of the fault. Fault simulation determines the extent to which a given set of stimulus patterns (test patterns) detects faults. Fault simulation is now integrated within tools for ATPG.[16] Nevertheless, we will discuss here some of the key concepts that are helpful for understanding this important aspect of the overall design flow.

We will consider only tests that detect faults in combinational logic with a single stuck fault. *Fault coverage*, the degree to which a set of test patterns detects the possible faults in a circuit, is defined below:

Fault Coverage = Number of Detected Faults / Total Number of Faults

It is important that a set of tests provide a high level of coverage in order to ensure that bad devices are not shipped to customers. *Fault grading*[17] determines the coverage provided by a set of test patterns by checking whether a fault is detectable and possibly finding a test for the fault. In many applications, the coverage must exceed 99.5% of the single stuck faults that a circuit could have.

Fault simulators exercise a circuit by cataloging the fault sites, injecting faults, applying stimulus patterns, and comparing the output of the circuit to a fault-free circuit. Patterns that do not reveal the presence of a fault are not useful in testing. Fault simulation is done in conjunction with test generation to evaluate stimulus patterns and to guide construction of a set of test patterns that will achieve a high level of coverage of the faults in a circuit. Various measures are used to reduce the time and effort required for test generation.

---

[16]See, for example, the *TetraMAX*© ATPG tool at www.synopsys.com.

[17]Also called *fault coverage analysis*.

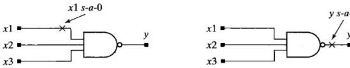


FIGURE 11-44 The faults *x1 s-a-0* and *y s-a-1* are equivalent faults.

11.7.1 Fault Collapsing

The efficiency of testing has an impact on the amortization of expensive testers. Needless testing wastes tester resources and time, so it is important to test only the faults that need to be tested and to quickly find test patterns that reveal as many faults as possible. Efficient fault simulators use *fault collapsing* to form equivalence classes of faults that are detectable by the same test. Faults that are detected by the same test are indistinguishable and are called *equivalent faults*. Fault simulators test only one fault in an equivalence class and eliminate the needless simulation of the other faults in the class. The same test detects any fault in the class, so only one fault in the class needs to be tested.

Example 11.25

The fault *x1 s-a-0* in Figure 11-44 cannot be distinguished from the *fault y s-a-1*, so the test that detects *x1 s-a-0* will also detect *y s-a-1*. The faults are members of the same equivalence class of faults.

End of Example 11.25

There are three major approaches to fault simulation: serial, parallel, and concurrent fault simulation [5]. Serial fault simulation is the slowest of the three approaches, but it is the simplest to understand and implement.

11.7.2 Serial Fault Simulation

Serial fault simulation considers a circuit's faults, one fault at a time, and executes the following sequential steps to determine whether an applied stimulus pattern reveals the fault:

```
(1) Create a list of fault sites;
(2) Inject a fault into the circuit and remove it from the list of faults;
(3) While (the list of fault sites is not empty) {
    Apply a pattern and simulate the good machine and the machine with
      the injected fault;
    Compare the good machine's output to that of the machine with the fault;
```

```
        if (there is a difference between the output of the machines)
        the pattern detected the fault;
        else
          the pattern did not detect the fault;
        inject another fault and remove it from the list of faults;
    }
```

11.7.3 Parallel Fault Simulation

Parallel fault simulation simultaneously simulates multiple copies of a machine (circuit), each with its own distinct injected fault, and compares their responses to that of the good machine. Although it is faster than serial fault simulation, parallel fault simulation requires more memory and efficient memory management techniques to accommodate the multiple copies of the circuit.

11.7.4 Concurrent Fault Simulation

Concurrent fault simulation is the most widely used algorithm. It limits the scope of fault simulation to the portion of a circuit that is relevant to backward justification (fanin) and forward sensitization (fanout) from a fault site. Concurrent fault simulation requires sophisticated topological analysis of a circuit to limit the scope of the searches that are used to justify and sensitize faults. The payoff is that a concurrent fault simulator is faster than serial or parallel fault simulators.

11.7.5 Probabilistic Fault Simulation

Another method for fault simulation, called probabilistic fault simulation, identifies test vectors that have high toggle coverage and uses them as the basis for test vectors to detect faults. There is a high degree of correlation between a test pattern that toggles a high number of nodes in a circuit and test patterns that detect a high number of faults [4]. Toggle tests are simpler to perform and faster than other methods of fault detection.

11.8 JTAG[18] Ports and Design for Testability

Design for testability (DFT) ensures that manufactured circuits can be tested for defects. DFT usually requires that a circuit is designed/modified to support testing, because the relatively small number of available I/O pins on a typical chip is inadequate for testing the internal nodes. There are methods for evaluating the testability of a chip by measuring the difficulty of controlling and observing internal nodes [12]. There are also several approaches to improving the testability of a circuit [6], but we will focus on scan-based methods, which extend the concept introduced in Section 11.7.6 for testing sequential circuits. We do so because scan-based approaches have become widespread

[18]The JTAG port is named after the Joint Test Action Group, a team of industry experts who defined the IEEE 1149.1 and 1149.1a standards.

and important. They not only support testing of circuits for defects, but also support debugging of embedded processors during software development, and support the field programmability of complex programmable logic devices (CPLDs) and FPGAs.

There are several practical problems in chip and board-level testing: (1) Sequential machines are difficult to test because they require a sequence of test patterns. (2) The internal nodes of large circuits cannot be observed at output pins, and may not be controlled easily by the available input pins. (3) The manufacturing process for printed circuit boards forms copper traces for signal paths. The circuit has a defect if traces are shorted or open. (4) An ASIC chip might not have a good bond between the board and a pin of the ASIC or between the pin and the core logic. (5) The core logic of a chip that is mounted on a board might have to be tested in the field, without removing the chip from the unit. (6) It might be necessary to isolate the location of a fault to a particular ASIC or module to reduce the cost of repairing a unit. The electronics industry has circumvented these problems by adopting a standard circuit interface that uses scan chains for board-level and chip-level testing.

11.8.1 Boundary Scan and JTAG Ports

Boundary scan is an extension of the scan register concept discussed in Section 11.7.6 for testing sequential circuits. A *boundary scan chain* is added to the netlist of an ASIC[19] by inserting *boundary scan cells* (BSCs) at its I/O pins and linking them to form a shift register around the chip. The same cell can also be used to replace the flips-flops within the core logic and to form internal scan paths consisting of one or more test-data registers linked together in a serial connection. When used internally, the cells are referred to as *data register* (DR) cells.

A typical BSC, or DR cell, is shown in Figure 11-45. The cell allows data to be scanned through the chip without affecting its normal operation (e.g., in on-line monitoring of the chip's operation). Two muxes control the datapaths of the cell. The input mux determines whether the capture/scan flip-flop is connected to *data_in* or to the serial input, *scan_in*. The output mux determines whether *data_in* or the output flip-flop is connected to *data_out*.

With *mode* = 0 the cell is in normal mode, and *data_in* is passed through the output multiplexer to *data_out* and to the capture/scan flip-flop, where it can be loaded by a pulse of *clockDR*. The capture/scan flip-flops support a boundary scan chain; the output register flip-flops hold their data while new data are scanned into the capture/scan flip-flop. *data_in* and *data_out* of a BSC are connected to the inputs and outputs of the core logic of the ASIC. In test mode, a pattern of data can be shifted into the capture/scan flip-flops under the control of *clockDR*. When the scan chain holds a desired pattern, the data in the capture/scan register can be loaded in parallel by toggling *updateDR* to update the *output register* flip-flops.

If a BSC resister cell is connected to an input pin of the chip, *data_in* is connected to the input pad of the chip, and *data_out* is connected to the input pad of the ASIC

[19]We will discuss scan chains for ASICs, but they are also used in FPGAs and other devices.

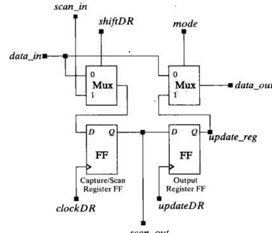


FIGURE 11-45 The DR cells used to implement boundary scan test registers and test-data registers include a capture/scan flip-flop and an output flip-flop.

core's logic. If the cell is used as an output, the core logic of the ASIC is connected to *data_in*, and *data_out* is connected to the output pin of the ASIC. A Verilog model of the BSC cell is given below.

```
module BSC_Cell (output data_out, output reg scan_out,
  input data_in, mode, scan_in, shiftDR, updateDR, clockDR
);
  reg update_reg;

  always @ (posedge clockDR) begin
    scan_out <= shiftDR ? scan_in : data_in;
  end

  always @ (posedge updateDR) update_reg <= scan_out;
  assign data_out = mode ? update_reg : data_in;
endmodule
```

11.8.2 JTAG Modes of Operation

The modes of operation of a boundary scan cell (data register cell) are summarized in Table 11-7. In *normal mode* (i.e., with *mode* = 0) data pass directly through the cell, from *data_in* to *data_out*. The multiplexer driving *data_out* adds a slight propagation delay to the signal path. In *test mode*, with *mode* = 1, cell *data_out* is driven by the output register (*update_reg*).

In *scan mode*, the boundary scan cells are connected as a shift register, with *scan_out* from one cell connected to *scan_in* of the next cell in the chain. A test pattern can be shifted into the register and then loaded into the *output register* to establish a

TABLE 11-7 Modes of operation of a BSC.

Mode	Operation
Normal	With *mode* = 0, *data_in* is connected directly to *data_out*, and the scan chain does not affect the operation of the ASIC.
Scan	With *shiftDR* = 1, data enter through *scan_in* and leave through *scan_out*, at the active edge of *clockDR*.
Capture	With *shiftDR* = 0, *data_in* is loaded into the capture register at the active edge of *clockDR*.
Update	With *mode* = 1, the output of the capture register is shifted to the update register at the active edge of *updateDR*.

logic value at *data_out* for the purpose of testing the core logic. With *shiftDR* = 1, data are shifted through the capture/scan register flip-flops on the rising edges of *clockDR*, from *scan_in* to *scan_out*.

The *capture mode* of the cell captures data from the ASIC without interfering with its operation. The data can be scanned out later, while the chip is operating. This mode is operational with *shiftDR* = 0, which connects the scan path to the capture/scan flip-flop. A subsequent clock pulse of *clockDR* loads *data_in* into the scan register. In this mode, *data_out* can be driven by *data_in* (*mode* = 0) or by the output flip-flop (*mode* = 1).

The *update mode* drives *data_out* by the contents of the output register, with *mode* = 1. The output register is loaded with the content of the scan register by applying a pulse of *updateDR*. If *data_out* is attached to the input pins of the ASIC, the pattern could be a test that is to be applied to the chip. The response of the chip can be captured by a pulse of *clockDR* while *shiftDR* = 0.

Boundary scan methods can test multiple chips on a PC board, the board traces between chips, and the connections between the pins of a chip and its core logic. A tester can isolate and test the core logic of ASICs that are equipped with boundary scan circuitry and a special test access port (TAP), also called a JTAG port. The TAP allows devices on a board to be linked together and tested in-place. A finite-state machine called a *TAP controller* controls a TAP. The JTAG standards IEEE 1149.1 [13] and 1149.1a specify the implementation of a TAP. The JTAG port of a chip can be daisy chained to the JTAG port of another chip, so that a scan chain can connect all of the chips on a board. If the chips on a board are equipped with a JTAG port and a boundary scan chain, a tester can detect shorts and opens in board traces, between the chip's I/O pins and the board, and between the ASIC core and its pad frame. An external tester can use the JTAG port to detect internal faults of the ASIC.

The JTAG port has assumed a much larger role than testing ASICs and printed circuit (PC) boards for production defects. The port is used to program configurable PLDs [14] and FPGAs.[20] It is also used to develop and debug software for embedded processors by controlling the processor and providing access to its internal registers.[21]

[20]See, for example, www.altera.com and www.xilinx.com.
[21]See www.agilent.com.

11.8.3 JTAG Registers

Each chip implementing the JTAG methodology must include a boundary scan register (formed by linking BSCs), a bypass register, and an instruction register. These mandatory registers can be viewed as having the configuration shown in Figure 11-46. The bypass register holds a single bit. The size of the instruction register and the other data registers can be customized to an application. An optional 32-bit wide device identification register can be used to hold data describing the part number, the manufacturer's name, and other information that can be accessed by an external tester. The current instruction in the instruction register determines which register is connected between test-data input (*TDI*) and test-data output (*TDO*). The actual register can be formed by linking one or more test-data registers (TDRs) of internal scan chains.

The single-bit bypass register has a cell like that shown in Figure 11-47, and the cells of the instruction register have an architecture like that shown in Figure 11-48.

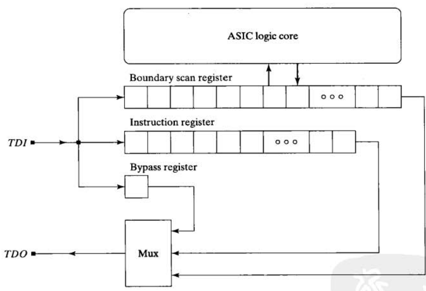


FIGURE 11-46 Register structure required by the JTAG specification.

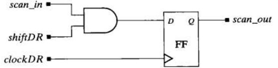


FIGURE 11-47 The bypass register (BR) cell for boundary scan.

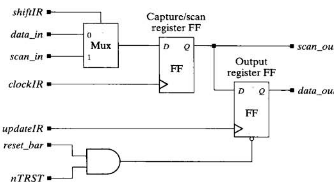


FIGURE 11-48 The instruction register (IR) cell for boundary scan.

The bypass register bypasses an ASIC in the scan chain on a PC board, reducing the length of a test by reducing the number of shifts that must occur before a pattern is located in a given scan register. A Verilog model of a bypass register cell is described below by *BR_Cell*. The signal *shiftDR* gates the scan path, and *clockDR* synchronizes the cell.

```
module BR_Cell (output reg scan_out, input scan_in, shiftDR, clockDR);
  always @ (posedge clockDR) scan_out <= scan_in & shiftDR;
endmodule
```

The instruction register specifies instructions and controls the internal datapath of the TAP. An instruction defines the serial test-data register path connected between *TDI* and *TDO* during scan operations. The cells of the instruction register have asynchronous set/reset, which can be programmed by the parameter *SR_value* to assert either 0 or 1 at the output, depending on the instruction held when the TAP's state machine enters its reset state. The instruction register has serial I/O through *scan_in/scan_out*, and parallel input/output through *data_in/data_out*. A Verilog model of the IR cell is given below.

```
module IR_Cell (
  output reg data_out, scan_out,
  input data_in, scan_in, shiftIR, reset_bar, nTRST, clockIR, updateIR
);
  parameter SR_value = 0;
  wire              S_R = reset_bar & nTRST;

  always @ (posedge clockIR) scan_out  <= shiftIR ? scan_in: data_in;
  always @ (posedge updateIR or negedge S_R)
   if (S_R == 0) data_out <= SR_value;
   else data_out <= scan_out;
endmodule
```

Note that a new instruction can be shifted into the scan register while an instruction is held in the output register. The signal *shiftIR* selects the input datapath of the cell, which can be the serial path connected to *scan_in*, or a parallel path connected to *data_in*. The latter provides a means of including data (e.g., status bits) from the ASIC in an instruction. The signal *reset_bar* is generated synchronously by the TAP controller when it enters its reset state; *nTRST* is an optional, asynchronous, active-low fifth input to the TAP.

11.8.4 JTAG Instructions

The mandatory instructions specified by the JTAG standard are summarized in Table 11-8. The BYPASS instruction scans data from *TDI* to *TDO* through a 1-bit bypass register, rather than through the entire boundary scan chain. This bypasses a chip that is not being tested and shortens the scan chain needed to test other components.

The EXTEST (external test) instruction is used to test the interconnect that is external to the chip. A pattern is scanned into the capture/scan register, then the data are loaded (in parallel) into the output register of the boundary scan chain. When the chip is placed in the test mode, the pattern appears at the output pins of the chip and drives the interconnect to other chips. Signal values from other chips can be captured and scanned out for analysis of the integrity of the interconnect structure.

TABLE 11-8 Instructions specified by IEEE standard 1149.1.

Instruction	Action
BYPASS	Shifts data through a single-cell bypass register, bypassing the ASIC's boundary scan register, reducing the length of the scan path needed to test other components.
EXTEST	Drives known values onto the output pins of the ASIC for testing board-level interconnect and logic external to the ASIC.
SAMPLE/PRELOAD	SAMPLE captures the data values present at the system pins and loads (in parallel) the data into the capture register flip-flops. PRELOAD places a test data pattern into the output register.
INTEST*	Applies a test pattern to the ASIC logic and captures the response from the logic. Connects only the boundary scan register between *TDI* and *TDO*.
RUNBIST*	The host ASIC can execute a self-test while the TAP controller is in the state *S_Run_Idle*.
IDCODE*	Shifts out the data in the IDCODE register (device identification register), providing the tester with the device manufacturer's name, part number, and other data. The instruction defaults to the BYPASS register if there is no IDCODE register in the TAP.

*denotes an optional instruction.

The SAMPLE/PRELOAD instruction captures data from the I/O pins of the ASIC without interfering with normal operation. The captured data can be scanned out for analysis of the operation of the chip.

The INTEST (internal test) instruction isolates and tests the internal circuits of individual components on a board. A pattern is scanned into the capture register, then the data are loaded in parallel into the output register of the boundary scan chain. When the chip is placed in the test mode the cells of the output register that are connected to input ports of the ASIC exercise the logic of the chip, forming outputs that can be captured at the cells of the capture/scan register and then scanned out for analysis. While a pattern is being scanned out another pattern can be scanned into the register.

The codes for the instructions implemented by the TAP are partially specified by the JTAG standard. The code for the BYPASS instruction is required to be all 1s. The code for the EXTEST instruction is all 0s. The TAP may also include optional test-data registers for implementing internal scan and other tests. Each test-data register corresponds to an internal scan chain that can be exercised by an external tester under the control of the TAP and the instruction register.

11.8.5 TAP Architecture

A TAP has the architecture shown in Figure 11-49. The *TDI*, *TMS*, and *nTRST* inputs have pull-ups to conform to the JTAG requirement that if, for example, the *TDI* input is disconnected the "... undriven input produces a response identical to the application of a logic 1." This has implications for the behavior of the TAP controller state machine, which will be discussed later.

An ASIC or other device with a JTAG port requires a test bus of four dedicated input pins (*TDI*, *TDO*, *TMS*, and *TCK*) to support boundary scan and internal testing.[22] The *TDI* and *TDO* pins of the TAP connect to the first and last cells in the boundary scan register chain and serve as an interface to the chip. The *test-data input* (*TDI*) pin serves as input for test patterns that are applied serially to the port; a *test-data output* (*TDO*) pin serves as a serial output port. The mode of operation of the TAP is controlled by the *test mode select* (*TMS*) input. A master clock is applied at the test clock (TCK) input pin for testing. A PC board implementing the JTAG architecture requires four extra pins to accommodate the *TDI*, *TDO*, *TMS*, and *TCK* signals, and possibly one more pin for *nTRST*, as shown in Figure 11-49. Each ASIC acts as a bus slave; an external agent serves as the bus master, and uses *TMS* and *TCK* to control the slave devices.

The TAP of each ASIC on the PC board includes a TAP controller, a state machine to which the four pins dedicated to JTAG are attached. The *TMS* input controls the state transitions of the TAP controller, with each transition occurring

[22] An optional fifth pin may be used to apply an asynchronous, active-low test-reset input signal (*nTRST*) to reset the TAP controller. Like *TMS* and *TDI*, *nTRST* must be attached to a pull-up device.

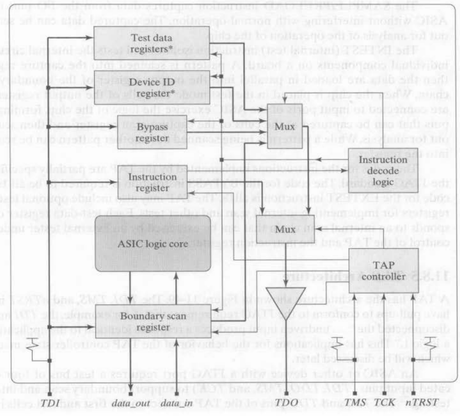
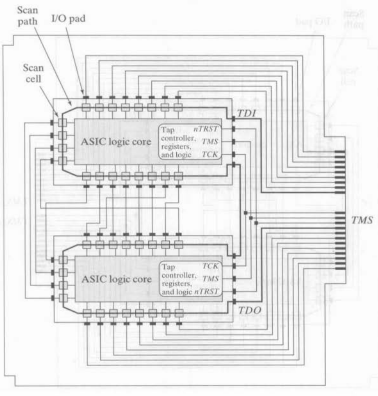


FIGURE 11-49 Chip architecture with JTAG test access port. (* denotes optional registers. The active-low input *nTRST* is also optional.)

on the positive edges of *TCK*.[23] Signals generated by the TAP controller drive the *shiftDR*, *mode*, *clockDR*, *updateDR*, *shiftIR*, *clockIR*, and *updateIR* inputs of the register cells. Figure 11-50(a) shows a PC board with two ASICs with boundary scans chain connected in a ring configuration [6]. For simplicity, the TAP control signals are not shown. In a ring configuration, each chip is driven by the same TAP signals. In a more general configuration known as a star (see Figure 11-50(b)), the serial ports of the chips are daisy-chained, but each chip in the chain has its own test mode select (*TMS*) signal. A bus master controls the TAPs by controlling the individual TMS signals, thereby allowing the individual TAP controllers to be controlled independently.

[23]Required by the JTAG specification.



(a)

FIGURE 11-50 A PC board with ASICs equipped with boundary scan cells and JTAG ports: (a) in test mode the chips on the board are daisy-chained in a ring configuration for *TMS* and (b) in test mode the chips on the board are daisy-chained in a star configuration for *TMS*.

11.8.6 TAP Controller State Machine

The instruction register and the TAP controller state machine control the datapath of the TAP. The ASM chart of the machine is shown in Figure 11-51, with decimal annotation showing the state codes.[24] All state transitions occur on the positive (rising) edge of *TCK*; the actions of the test logic in the ASIC are to occur on either the rising or the falling edge of *e* in each state of the controller.

[24]The state codes of the TAP are not specified by the JTAG standard. For clarity, we have added the prefix *S_* to the names of the states of the TAP controller shown in Figure 11.51. For brevity, the states *Test-Logic-Reset, Run-Test-Idle, Select-DR-Scan*, and *Select-IR-Scan* specified in the JTAG standard are named *S_Reset, S_Run_Idle, S_Select_Dr*, and *S_Select_IR*, respectively, in Figure 11.51 and in our model of the TAP controller.

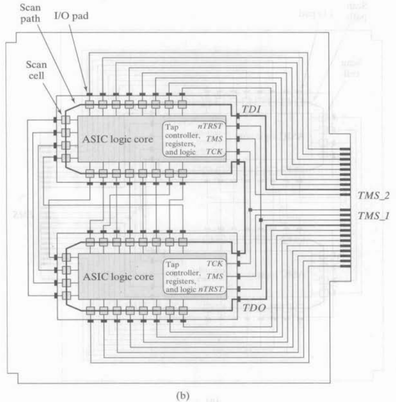


FIGURE 11-50 Continued

The TAP controller's ASM chart is nearly symmetric, with one path controlling the activity of the data register, and the other path controlling the activity of the instruction register of the TAP. If the state of the machine is in *S_Run_Idle* and *TMS* is asserted, the state will return to *S_Reset* if *TMS* is held asserted for two clock cycles, after moving through *S_Select_DR*, and *S_Select_IR*. It will reside in *S_Reset* until *TMS* is de-asserted to cause a transition to *S_Run_Idle*.

Note how alternating values of *TMS* control the activity flow of the TAP controller. That is, the value of *TMS* that causes a transition into *S_Reset*, *S_Run_Idle*, *S_Shift_DR*, *S_Pause_DR*, *S_Shift_IR*, or *S_Pause_IR* will cause the machine to remain in that state until the value of *TMS* is switched. The states *S_Capture_DR*, *S_Exit1_DR*, *S_Exit2_Dr*, *S_Capture_IR*, *S_Exit1_IR*, and S_*Exit2_IR* are temporary states. The machine passes through them in one cycle. *S_Capture_Dr* and *S_Capture_IR* are entered and occupied for one cycle when the corresponding capture/scan register is loaded. Note that the so-called exit states (e.g., *S_Exit_2*) enable a single controlling signal, *TMS*, to effectively direct the activity flow of the machine. For example, the flow from *S_Pause_DR* has three ultimate

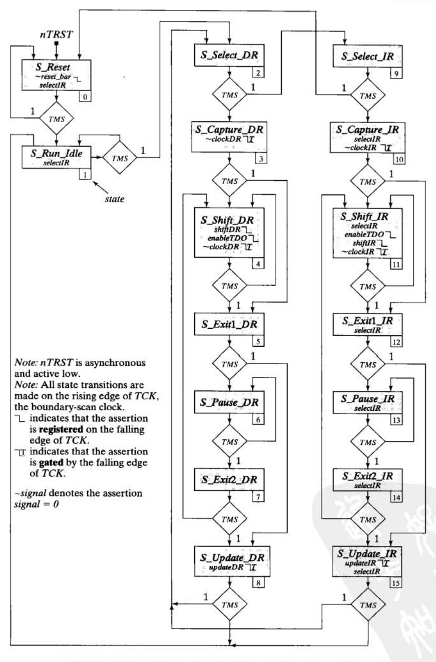


FIGURE 11-51 ASM chart for the TAP controller state machine.

destinations (e.g., *S_Pause_DR*, *S_Update_DR*, or *S_Shift_DR*). A single-bit control signal selects between three possibilities by sequencing the decisions over two clock cycles.

The content of the instruction register determines whether the boundary scan register or one of the (optional) test-data registers is affected by the operation of the controller. Note that the TAP controller input *nTRST* is optional, because the state *S_reset* can be reached from any other state by asserting *TMS* for at most five clock cycles. In post-synthesis simulation, *nTRST* might be needed to drive the gate-level model of the TAP controller into a known initial state.

The control states affecting the data register are described in Table 11-9. The control states affecting the instruction register have a similar description.

TABLE 11-9 Control states of the TAP controller state machine.

State	Activity
S_Reset	The reset state of the TAP controller. The test logic of the TAP is disabled and the host ASIC operates normally. If the machine has a device identification register, the IDCODE instruction is loaded into the instruction register; otherwise, the BYPASS instruction is loaded.
S_Run_Idle	The TAP controller resides in *S_Run_Idle* while the host ASIC executes an internal test, such as *BIST*. The instruction register must be preloaded with information supporting the test.
S_Select_DR	An assertion of *TMS* while the controller is in *S_Run_Idle* drives the state to *S_Select_DR*, where it resides for one cycle, before passing to *S_Capture_DR* to initiate a scan data sequence, or to *S_Select_IR*, where a sequence can be initiated to update the instruction register or terminate the activity
S_Capture_DR	While the state resides in *S_Capture_DR*, the capture/scan register of the boundary scan register or the test-data register specified by the current instruction can be loaded in parallel (via *data_in*). Capture is initiated by a pulse of *clockDR*, with *shift_DR* low.
S_Shift_DR	A test-data register selected by the instruction register is shifted toward its serial output by one cell at each active edge of *TCK*. A data bit enters the register from the *TDI* port, and leaves from the *TDO* port. The buffer driving *TDO* is active only during shifting.
S_Exit1_DR	A temporary state, entered from *S_Shift_DR* (after a shifting sequence), or from *S_Capture_DR* (bypassing an initial shifting sequence). After one cycle, the state transitions to *S_Pause_DR* to pause until *TMS* is again asserted, or to *S_Update_DR*, where the captured data are loaded into the output register.
S_Pause_DR	The state resides in *S_Pause_DR* to temporarily halt the scanning process, with $TMS = 0$, until *TMS* is asserted, with the capture/scan register cells holding their state.
S_Exit2_DR	A temporary state. The state resides in *S_Exit2_DR* for one cycle, before a transition to *S_Shift_DR*, where it initiates a scan sequence, or to *S_Update_DR*, where the scan process is terminated and the captured data are loaded into the output register.
S_Update_DR	The state resides in *S_Update_DR* for one cycle after the clock that loads the output register from the capture/scan register, before making a transition to *S_Select_DR*, where it initiates a scan sequence or an instruction sequence, or to *S_Run_Idle*, where it resides while the ASIC executes operations. In test mode the contents of the output register drive the parallel output.

TABLE 11-10 Moore-type outputs generated by the TAP controller state machine.

Output	Function
reset_bar	Resets the instruction register (IR) to IDCODE or to BYPASS.
shiftIR	Selects the serial input to the capture/scan flip-flop in the instruction register cells.
clockIR	Captures data at the input of the IR or shifts the contents of the IR toward the test data output. Action is gated by the falling edge of *TCK*.
updateIR	Loads the output register flip-flop with the content of the capture flip-flop of the IR. Action is gated by the falling edge of *TCK*.
shiftDR	Selects the serial input to the capture/scan flip-flop in the TDR cells.
clockDR	Captures data at the input of the IR or shifts the contents of the TDR toward the test data output. Action is gated by the falling edge of *TCK*.
updateDR	Loads the output register with the content of the TDR capture/scan flip- flop. Action is gated by the falling edge of *TCK*.
selectIR	Selects either the instruction register or a test-data register to be connected between the TDI and TDO pins of the TAP.
enable TDO	Enables the three-state buffer that drives the test-data output (*TDO*).

The output signals generated by the TAP controller state machine to control the operation of the scan registers are shown in Table 11-10.

11.8.7 Design Example: Testing with JTAG

This example shows how to augment an ASIC with a boundary scan chain and a TAP controller for JTAG. Then the BYPASS and INTEST commands will be demonstrated. Exercises at the end of the chapter will deal with additional features of the controller. The ASIC is a simple 4-bit adder, but it is sufficient for demonstrating the same procedure that is taken for more complex ASICs. We will consider architectural and operational details.

Testing an ASIC with JTAG requires systematic execution of several steps. For example, to test an ASIC core consisting of combinational logic, the state of the machine must be directed to *S_Shift_DR* and remain there for as many cycles as needed to shift a test pattern into the boundary scan register. At the end of the shifting operations, the test inputs should reside in the cells of the capture/scan register that drive the inputs of the chip. Toggling *update_DR* will transfer the content of the capture/scan register into the output register. In test mode, the test patterns in the output register will drive the input pins of the ASIC. The response of the ASIC will appear at the *data_in* pins of the capture/scan cells that connect to the outputs of the ASIC. With *shiftDR* de-asserted, toggling *clockDR* will capture the data at the input pins and load the capture/scan register with the response of the circuit to the test pattern. Then, with *shiftDR* asserted, successive toggling of *clockDR* will scan the data out of the scan chain. Another pattern can be scanned in while the previous pattern is scanned out.

The overall structure of an ASIC that has been modified to include a TAP for JTAG is shown in Figure 11-52. For simplicity, the ASIC is an embedded 4-bit adder. The TAP controller and the control signals for the TAP are omitted in the illustration, but are included in the model of the JTAG-enhanced ASIC.

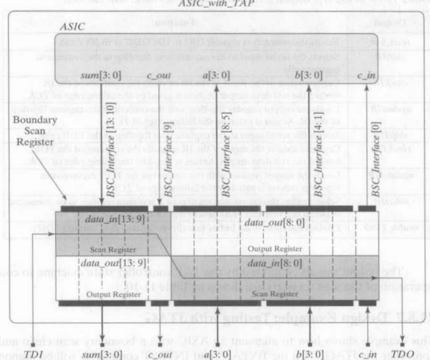


FIGURE 11-52 Boundary scan register and port interface structure.

A key step in the implementation is to create an interface between the ASIC and its environment. Note that the ASIC retains its port structure, but is wired directly to the boundary scan register through the bus *BSC_Interface* [13: 0], which accomplishes a *port interface mapping* between the ASIC, the boundary scan register, and the environment. The mode of a port of the ASIC determines whether the associated wires of *BSC_Interface* are connected to an input or output port of the boundary scan register. The outputs of the ASIC are wired to those input pins of the boundary scan register that are connected to the capture/scan register by *data_in* [13: 9]. The corresponding output register cells are connected to the outputs of *ASIC_with_TAP* (i.e., to the primary outputs). Likewise, the output register cells at *data_out* [8: 0] drive the inputs of the ASIC. The corresponding capture/scan register cells are driven by the external (primary) inputs of the chip through *data_in* [8: 0].

The capture/scan register and output register of the boundary scan register unit are shown separately, with a datapath through them representing the scan path. The scan register (shaded cells) is connected to the outputs of the ASIC and the primary input pins of the chip; the output register is connected to the primary output pins and

to the inputs of the ASIC. For example, the *Sum*[3: 0] and *c_out* ports of the ASIC are connected to *data_in*[13: 9], and *data_out*[13: 9] is connected to {*sum*[3: 0], *c_out*} at the interface between *ASIC_with_TAP* and its host environment.

The structure shown in Figure 11-52 is flexible, and the ordering of the interface signals is arbitrary. It is important to note that (1) the port structure can be modified to accommodate the ports of a different ASIC (2) the boundary scan register can be resized, and(3) the mapping of *BSC_Interface* can be declared to match the I/O of the ASIC.

As a preliminary step in the overall development and verification of *ASIC_with_TAP*, we show below models of an instruction register and an 8-bit boundary scan register and the results of a brief simulation exercise, demonstrating the parallel and serial I/O modes of the boundary scan register. For this exercise, *shiftDr*, *clockDR*, *updateDR*, and *mode* were controlled by the testbench. The simulation results in Figure 11-53 are

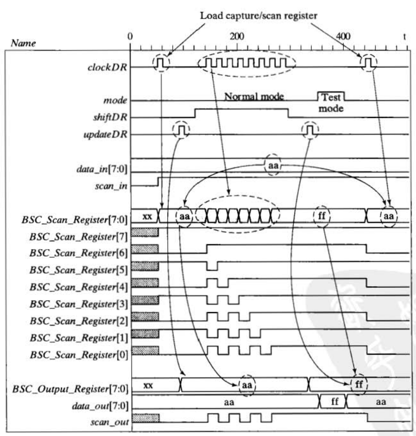


FIGURE 11-53 Simulation results demonstrating correct operation of an 8-bit boundary scan register.

annotated to highlight the normal and test modes of operation, and the activity of the registers. The first pulse of *clockDR* captures parallel data at *data_in* (8'haa), which is loaded into *BSC_Scan_Register*. With *shiftDR* de-asserted, the first pulse of *updateDR* demonstrates that the scan register (8'haa) is loaded into the output register, without interfering with normal operation (*data_in* and *data_out* are not affected). With *shiftDR* asserted, the pulses of *clockDR* scan a 1 into the scan register, and data exits through *scan_out*. The second pulse of *updateDR* loads the value of the capture/scan register (8'hff) into the output register. When the test mode is asserted, the value in the output register drives the bus *data_out*. When mode is again de-asserted, *data_out* reverts back to 8'haa, the value of *data_in*. The last pulse of *clockDR* captures *data_in* and loads the value 8'haa into the scan register again.

```
module Boundary_Scan_Register #(parameter size = 14)(
  output      [size -1: 0]      data_out,
  output                        scan_out,
  input       [size -1: 0]      data_in,
  input                         scan_in,
  input                         shiftDR, mode, clockDR, updateDR
);

  reg         [size -1: 0]      BSC_Scan_Register, BSC_Output_Register;

  always @ (posedge clockDR)
   BSC_Scan_Register <= shiftDR ? {scan_in, BSC_Scan_Register [ size -1: 1]} :
    data_in;

  always @ (posedge updateDR) BSC_Output_Register <= BSC_Scan_Register;

  assign scan_out = BSC_Scan_Register [0];
  assign data_out = mode ? BSC_Output_Register : data_in;
endmodule

module Instruction_Register #(parameter IR_size = 3)(
  output      [IR_size -1: 0]   data_out,
  output                        scan_out,
  input       [IR_size -1: 0]   data_in,
  input                         scan_in,shiftIR, clockIR, updateIR, reset_bar
);
  reg         [IR_size -1: 0]   IR_Scan_Register, IR_Output_Register;

  assign                        data_out = IR_Output_Register;
  assign                        scan_out = IR_Scan_Register [0];

  always @ (posedge clockIR)
   IR_Scan_Register <= shiftIR ? {scan_in, IR_Scan_Register [IR_size - 1: 1]} : data_in;
  always @ ( posedge updateIR, negedge reset_bar)     // Asynchronous per
                                                        1140.1a.
   if (reset_bar == 0) IR_Output_Register <= ~(0);    // Fills IR with 1s
                                                      // for BYPASS instruction
   else IR_Output_Register <= IR_Scan_Register;

endmodule
```

A model of the TAP controller state machine is given below. The state of the machine has a binary code. Also, for simplicity, gated clock signals are generated for *clockDr*, *updateDR*, *clockIR*, and *updateIR*. For actual implementation, the signals would be connected to the clock inputs of special flip-flops having a multiplexed input (see Section 6.11).

```
module TAP_Controller (
  output reg  reset_bar, selectIR, shiftIR,
  output reg  shiftDR, enableTDO,
  output      clockDR, updateDR, clockIR, updateIR,
  input       TMS, TCK
);
  parameter S_Reset           = 0,
            S_Run_Idle        = 1,
            S_Select_DR       = 2,
            S_Capture_DR      = 3,
            S_Shift_DR        = 4,
            S_Exit1_DR        = 5,
            S_Pause_DR        = 6,
            S_Exit2_DR        = 7,
            S_Update_DR       = 8,
            S_Select_IR       = 9,
            S_Capture_IR      = 10,
            S_Shift_IR        = 11,
            S_Exit1_IR        = 12,
            S_Pause_IR        = 13,
            S_Exit2_IR        = 14,
            S_Update_IR       = 15;

  reg [3:0]   state, next_state;

  pullup (TMS);    // Required by IEEE 1149.1a; ensures that an undriven input
  pullup (TDI);    // produces a response identical to the application of a logic 1."
               // Program for Xilinx implementation

  always @ (negedge TCK) reset_bar <= (state == S_Reset) ? 0 : 1; // Registered
                                                                     active low

  always @ (negedge TCK) begin
    shiftDR <= (state == S_Shift_DR) ? 1 : 0;      // Registered select for scan mode
    shiftIR <= (state == S_Shift_IR) ? 1: 0;
                                                 // Registered output enable
    enableTDO <=  ((state == S_Shift_DR) || (state == S_Shift_IR)) ? 1 : 0;
  end

  // Gated clocks for capture registers
  assign clockDR = !(((state == S_Capture_DR) || (state == S_Shift_DR)) &&
    (TCK == 0));
  assign clockIR =   !(((state == S_Capture_IR) || (state == S_Shift_IR)) &&
    (TCK == 0));

  // Gated clocks for output registers
  assign updateDR = (state == S_Update_DR)  && (TCK == 0);
```

```
assign updateIR =  (state == S_Update_IR)  && (TCK == 0);

always @ (posedge TCK ) state <= next_state;

always @ (state, TMS) begin
 selectIR = 0;
 next_state = state;

 case (state)
  S_Reset:          begin
                     selectIR = 1;
                     if (TMS == 0) next_state = S_Run_Idle;
                    end
  S_Run_Idle:       begin selectIR = 1; if (TMS)  next_state = S_Select_DR; end
  S_Select_DR:      next_state = TMS ? S_Select_IR: S_Capture_DR;
  S_Capture_DR:     begin next_state = TMS ? S_Exit1_DR: S_Shift_DR; end
  S_Shift_DR:       if (TMS) next_state = S_Exit1_DR;
  S_Exit1_DR:       next_state = TMS ? S_Update_DR: S_Pause_DR;
  S_Pause_DR:       if (TMS) next_state = S_Exit2_DR;
  S_Exit2_DR:              next_state = TMS ? S_Update_DR: S_Shift_DR;
  S_Update_DR:      begin next_state = TMS ? S_Select_DR: S_Run_Idle; end
  S_Select_IR:      begin
                     next_state = TMS ? S_Reset: S_Capture_IR;
                    end
  S_Capture_IR:     begin
                     selectIR = 1;
                     next_state = TMS ? S_Exit1_IR: S_Shift_IR;
                    end
  S_Shift_IR:       begin selectIR = 1; if (TMS) next_state = S_Exit1_IR;end
  S_Exit1_IR:       begin
                     selectIR = 1;
                     next_state = TMS ? S_Update_IR: S_Pause_IR;
                    end
  S_Pause_IR:       begin selectIR = 1; if (TMS) next_state = S_Exit2_IR; end
  S_Exit2_IR:       begin
                     selectIR = 1;
                     next_state = TMS ? S_Update_IR: S_Shift_IR;
                    end
  S_Update_IR:      begin
                     selectIR = 1;
                     next_state = TMS ? S_Select_DR: S_Run_Idle;
                    end
   default          next_state = S_Reset;
  endcase
 end
endmodule
```

The parameterized model listed below, *ASIC_with_TAP*, instantiates the following modules: *ASIC*, *TAP_Controller*, *Boundary_Scan_Register*, *Instruction_Register*, *Instruction_*Decoder, and *TAP_*Controller. In general, the instruction register of a TAP

loads a parallel data path (*data_in*) in the state *S_Capture_IR*, which provides the TAP with design-specific information generated in the host component.[25] In this example, *Dummy_data* = 3'b001 is passed through the port *data_in*.

```
module ASIC_with_TAP #(parameter size = 4)(
  output        [size -1: 0]     sum,        // ASIC interface I/O
  output                         c_out,
  input         [size -1: 0]     a, b,
  input                          c_in,
  output                         TDO,              // TAP interface signals
  input                          TDI, TMS, TCK
);

  parameter     BSR_size = 14;

  parameter     IR_size = 3;
  wire          [BSR_size -1: 0] BSC_Interface;  // Declarations for bound-
                                                    ary scan register I/O
  wire                           reset_bar,        // TAP controller outputs
                                 selectIR, enableTDO,
                                 shiftIR, clockIR, updateIR,
                                 shiftDR, clockDR, updateDR;

  wire                           test_mode, select_BR;
  wire                           TDR_out;               // Test data
                                                          register serial
                                                          datapath
  wire          [IR_size -1: 0]  Dummy_data = 3'b001;   // Captured in
                                                          S_Capture_IR
  wire          [IR_size -1: 0]  instruction;
  wire                           IR_scan_out;           // Instruction
                                                          register
  wire                           BSR_scan_out;          // Boundary scan
                                                          register
  wire                           BR_scan_out;           // Bypass register

  assign    TDO = enableTDO ? selectIR ? IR_scan_out : TDR_out : 1'bz;
  assign    TDR_out = select_BR ? BR_scan_out : BSR_scan_out;

  ASIC M0 (
   .sum (BSC_Interface [13: 10]),
   .c_out (BSC_Interface [9]),
   .a (BSC_Interface [8: 5]),
   .b (BSC_Interface [4: 1]),
   .c_in (BSC_Interface [0]));
```

[25]The JTAG standard requires that the cells of the two least significant bits of the instruction register shall load the pattern 2'b01 in the state *S_Capture_IR*. The remaining bits have fixed (0 or 1) but application-dependent values.

```
Bypass_Register M1(
  .scan_out (BR_scan_out),
  .scan_in (TDI),
  .shiftDR (shift_BR),
  .clockDR (clock_BR));

Boundary_Scan_Register M2(
  .data_out ({sum, c_out, BSC_Interface [8: 5], BSC_Interface [4: 1],
    BSC_Interface [0]}),
  .data_in ({BSC_Interface [13: 10], BSC_Interface [9], a, b, c_in}),
  .scan_out (BSR_scan_out),
  .scan_in (TDI),
  .shiftDR (shiftDR),
  .mode (test_mode),
  .clockDR (clock_BSC_Reg),
  .updateDR (update_BSC_Reg));

Instruction_Register M3 (
  .data_out (instruction),
  .data_in (Dummy_data),
  .scan_out (IR_scan_out),
  .scan_in (TDI),
  .shiftIR (shiftIR),
  .clockIR (clockIR),
  .updateIR (updateIR),
  .reset_bar (reset_bar));

Instruction_Decoder M4 (
  .mode (test_mode),
  .select_BR (select_BR),
  .shift_BR (shift_BR),
  .clock_BR (clock_BR),
  .shift_BSC_Reg (shift_BSC_Reg),
  .clock_BSC_Reg (clock_BSC_Reg),
  .update_BSC_Reg (update_BSC_Reg),
  .instruction (instruction),
  .shiftDR (shiftDR),
  .clockDR (clockDR),
  .updateDR (updateDR));

TAP_Controller M5 (
  .reset_bar(reset_bar),
  .selectIR (selectIR),
  .shiftIR (shiftIR),
  .clockIR (clockIR),
  .updateIR (updateIR),
  .shiftDR (shiftDR),
  .clockDR (clockDR),
  .updateDR (updateDR),
```

```
  .enableTDO (enableTDO),
  .TMS (TMS),
  .TCK (TCK));

endmodule

module ASIC #(parameter size = 4) (
 output      [size -1: 0]       sum,
 output                         c_out,
 input       [size -1: 0]       a, b,
 input                          c_in
);

 assign {c_out, sum} = a + b + c_in;
endmodule

module Bypass_Register (
 output reg scan_out,
 input        scan_in, shiftDR, clockDR
);

 always @ (posedge clockDR) scan_out <= scan_in & shiftDR;
endmodule

module Instruction_Decoder #(parameter IR_size = 3) (
 output reg              mode, select_BR, clock_BR, clock_BSC_Reg,
                           update_BSC_Reg,
 output                  shift_BR, shift_BSC_Reg,
 input [IR_size -1: 0]  instruction,
 input                   shiftDR, clockDR, updateDR
);
 parameter  BYPASS              = 3'b111;       // Required by 1149.1a
 parameter  EXTEST              = 3'b000;       // Required by 1149.1a
 parameter  SAMPLE_PRELOAD      = 3'b010;
 parameter  INTEST              = 3'b011;
 parameter  RUNBIST             = 3'b100;
 parameter  IDCODE              = 3'b101;

 assign     shift_BR = shiftDR;
 assign     shift_BSC_Reg = shiftDR;

 always @ (instruction, clockDR, updateDR) begin
  mode = 0; select_BR = 0;                 // default is test-data register
  clock_BR = 1; clock_BSC_Reg = 1;
  update_BSC_Reg = 0;

  case (instruction)
   EXTEST:               begin mode = 1; clock_BSC_Reg = clockDR;
                         update_BSC_Reg = updateDR; end
   INTEST:               begin mode = 1; clock_BSC_Reg = clockDR;
                         update_BSC_Reg = updateDR; end
   SAMPLE_PRELOAD:       begin clock_BSC_Reg = clockDR;
```

```
                    update_BSC_Reg = updateDR; end
      RUNBIST:      begin end
      IDCODE:       begin select_BR = 1; clock_BR = clockDR; end
      BYPASS:       begin select_BR = 1; clock_BR = clockDR; end
      default:      begin select_BR = 1; end

   endcase
  end
endmodule
```

The structure of the testbench (*t_ASIC_with_TAP*) used to test *ASIC_with_TAP* is shown in Figure 11-54. Two arrays, *Array_of_TAP*_Instructions and *Array_of ASIC_Test_Patterns*, hold patterns for scanning instructions and test patterns into the boundary scan register. The test sequence selects a pattern from one of the registers and loads it into *Pattern_Register*. When the test sequence asserts a load signal the pattern held in *Pattern_Register* is loaded into the register *TDI_Reg* within *TDI_Generator*. The TAP controller scans the pattern from *TDI_Reg* into the TDI port

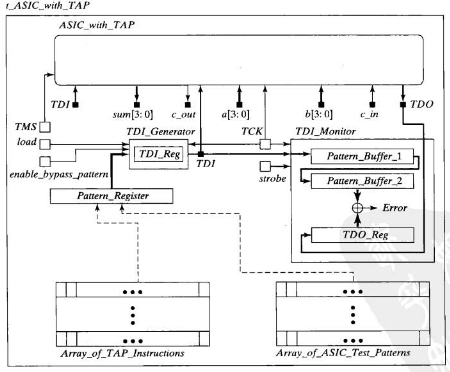


FIGURE 11-54 Structure of a testbench for *ASIC_with_TAP*.

of the TAP and into *Patttern_Buffer_1*. The pattern from the TDO port of the TAP is scanned into *TDO_Reg* within TDO_Monitor, and at the same time the pattern in *Pattern_Buffer_1* is scanned into *Pattern_Buffer_2*. When the scan activity is complete, the contents of *TDO_Reg* and *Pattern*_Buffer_2 are compared to detect an error.

The testbench *t_ASIC_with_TAP* is given below, with comments identifying some of the functional features that need to be verified.

```
module t_ASIC_with_TAP ();                                 // Testbench
 parameter                        size = 4;
 parameter                        BSC_Reg_size = 14;
 parameter                        IR_Reg_size = 3;
 parameter                        N_ASIC_Patterns = 8;
 parameter                        N_TAP_Instructions = 8;
 parameter                        Pause_Time = 40;
 parameter                        End_of_Test = 1500;
 parameter                        time_1 = 350, time_2 = 550;

 wire        [size -1: 0]         sum;
 wire        [size -1: 0]         sum_fr_ASIC = M0.BSC_Interface [13: 10];

 wire                             c_out;
 wire                             c_out_fr_ASIC = M0.BSC_Interface [9];
 reg         [size -1: 0]         a, b;
 reg                              c_in;
 wire        [size -1: 0]         a_to_ASIC = M0.BSC_Interface [8: 5];
 wire        [size -1: 0]         b_to_ASIC = M0.BSC_Interface [4: 1];
 wire                             c_in_to_ASIC = M0.BSC_Interface [0];

 reg         TMS, TCK;
 wire        TDI;
 wire        TDO;
 reg         load_TDI_Generator;
 reg         Error, strobe;
 integer     pattern_ptr;
 reg         [BSC_Reg_size -1: 0]   Array_of_ASIC_Test_Patterns
                                      [0: N_ASIC_Patterns -1];
 reg         [IR_Reg_size -1: 0]    Array_of_TAP_Instructions
                                      [0: N_TAP_Instructions -1];
 reg         [BSC_Reg_size -1: 0]   Pattern_Register;    // Size to maximum
                                                            TDR
 reg           enable_bypass_pattern;

 ASIC_with_TAP M0 (sum, c_out, a, b, c_in, TDO, TDI, TMS, TCK);

 TDI_Generator M1(
  .to_TDI (TDI),
  .scan_pattern (Pattern_Register),
  .load (load_TDI_Generator),
  .enable_bypass_pattern (enable_bypass_pattern),
  .TCK (TCK));
```

```
TDO_Monitor M3 (
  .to_TDI (TDI),
  .from_TDO (TDO),
  .strobe (strobe),
  .TCK (TCK));

initial #End_of_Test $finish;

initial begin TCK = 0; forever #5 TCK = ~TCK; end

/* Summary of a basic test plan for ASIC_with TAP

Verify default to bypass instruction
Verify bypass register action: Scan 10 cycles, with pause before exiting
Verify pull up action on TMS and TDI
Reset to S_Reset after five assertions of TMS
Boundary scan in, pause, update, return to S_Run_Idle
Boundary scan in, pause, resume scan in, pause, update, return to S_Run_Idle
Instruction scan in, pause, update, return to S_Run_Idle
Instruction scan in, pause, resume scan in, pause, update, return to S_Run_Idle
*/
// TEST PATTERNS
// External I/O for normal operation

initial fork
 // {a, b, c_in} = 9'b0;
{a, b, c_in} = 9'b_1010_0101_0; // sum = F, c_out = 0, a = A, b = 5, c_in = 0
join

/* Option to force error to test fault detection

initial begin :Force_Error
force M0.BSC_Interface [13: 10] = 4'b0;
end
*/

initial begin                    // Test sequence: Scan, pause, return to S_Run_Idle
  strobe = 0;
  Declare_Array_of_TAP_Instructions;
  Declare_Array_of_ASIC_Test_Patterns;
  Wait_to_enter_S_Reset;

// Test for power-up and default to BYPASS instruction (all 1s in IR), with default path
// through the Bypass Register, with BSC register remaining in wakeup state (all x).
// ASIC test pattern is scanned serially, entering at TDI, passing through the
  bypass register,
// and exiting at TDO. The BSC register and the IR are not changed.

   pattern_ptr = 0;
   Load_ASIC_Test_Pattern;
   Go_to_S_Run_Idle;
   Go_to_S_Select_DR;
   Go_to_S_Capture_DR;
```

```
    Go_to_S_Shift_DR;
    enable_bypass_pattern = 1;
    Scan_Ten_Cycles;
    enable_bypass_pattern = 0;
    Go_to_S_Exit1_DR;
    Go_to_S_Pause_DR;
    Pause;
    Go_to_S_Exit2_DR;
    /*
    Go_to_S_Shift_DR;
    Load_ASIC_Test_Pattern;        // option to re-load same pattern and scan again
    enable_bypass_pattern = 1;
    Scan_Ten_Cycles;
    enable_bypass_pattern = 0;
    Go_to_S_Exit1_DR;
    Go_to_S_Pause_DR;
    Pause;
    Go_to_S_Exit2_DR;
    */
    Go_to_S_Update_DR;
    Go_to_S_Run_Idle;
  end

// Test to load instruction register with INTEST instruction

  initial #time_1 begin
   pattern_ptr = 3;
   strobe = 0;
   Load_TAP_Instruction;
   Go_to_S_Run_Idle;
   Go_to_S_Select_DR;
   Go_to_S_Select_IR;
   Go_to_S_Capture_IR;                  // Capture dummy data (3'b011)
    repeat (IR_Reg_size) Go_to_S_Shift_IR;
   Go_to_S_Exit1_IR;
   Go_to_S_Pause_IR;
   Pause;
   Go_to_S_Exit2_IR;
   Go_to_S_Update_IR;
   Go_to_S_Run_Idle;
  end

// Load ASIC test pattern
 initial #time_2 begin
   pattern_ptr = 0;
   Load_ASIC_Test_Pattern;
   Go_to_S_Run_Idle;
   Go_to_S_Select_DR;
   Go_to_S_Capture_DR;
   repeat (BSC_Reg_size) Go_to_S_Shift_DR;
```

```
      Go_to_S_Exit1_DR;
      Go_to_S_Pause_DR;
      Pause;
      Go_to_S_Exit2_DR;
      Go_to_S_Update_DR;
      Go_to_S_Run_Idle;

// Capture data and scan out while scanning in another pattern
      pattern_ptr = 2;
      Load_ASIC_Test_Pattern;
      Go_to_S_Select_DR;
      Go_to_S_Capture_DR;
      strobe = 1;
      repeat (BSC_Reg_size) Go_to_S_Shift_DR;

      Go_to_S_Exit1_DR;

      Go_to_S_Pause_DR;
      Go_to_S_Exit2_DR;
      Go_to_S_Update_DR;
      strobe = 0;
      Go_to_S_Run_Idle;
    end

    /****************************** TAP CONTROLLER TASKS ******************************/
     task Wait_to_enter_S_Reset;
      begin
      @ (negedge TCK) TMS = 1;
     end
     endtask

     task Reset_TAP;
      begin
        TMS = 1;
        repeat (5) @ (negedge TCK);
      end
     endtask

    task Pause;                 begin #Pause_Time;              end endtask

    task Go_to_S_Reset;         begin @ (negedge TCK) TMS = 1; end endtask
    task Go_to_S_Run_Idle;      begin @ (negedge TCK) TMS = 0; end endtask

    task Go_to_S_Select_DR;     begin @ (negedge TCK) TMS = 1; end endtask
    task Go_to_S_Capture_DR;    begin @ (negedge TCK) TMS = 0; end endtask
    task Go_to_S_Shift_DR;      begin @ (negedge TCK) TMS = 0; end endtask
    task Go_to_S_Exit1_DR;      begin @ (negedge TCK) TMS = 1; end endtask
    task Go_to_S_Pause_DR;      begin @ (negedge TCK) TMS = 0; end endtask
    task Go_to_S_Exit2_DR;      begin @ (negedge TCK) TMS = 1; end endtask
    task Go_to_S_Update_DR;     begin @ (negedge TCK) TMS = 1; end endtask

    task Go_to_S_Select_IR;     begin @ (negedge TCK) TMS = 1; end endtask
    task Go_to_S_Capture_IR;    begin @ (negedge TCK) TMS = 0; end endtask
```

```
task Go_to_S_Shift_IR;        begin @ (negedge TCK) TMS = 0; end endtask
task Go_to_S_Exit1_IR;        begin @ (negedge TCK) TMS = 1; end endtask
task Go_to_S_Pause_IR;        begin @ (negedge TCK) TMS = 0; end endtask
task Go_to_S_Exit2_IR;        begin @ (negedge TCK) TMS = 1; end endtask
task Go_to_S_Update_IR;       begin @ (negedge TCK) TMS = 1; end endtask
task Scan_Ten_Cycles;         begin repeat (10)  begin @ (negedge TCK)
                                TMS = 0;
                              @ (posedge TCK) TMS = 1; end end endtask

/******************************** ASIC TEST PATTERNS ********************************/
task Load_ASIC_Test_Pattern;
 begin
  Pattern_Register = Array_of_ASIC_Test_Patterns [pattern_ptr];
  @ (negedge TCK ) load_TDI_Generator = 1;
  @ (negedge TCK) load_TDI_Generator = 0;
 end
endtask

task Declare_Array_of_ASIC_Test_Patterns;
 begin
 //s3 s2 s1 s0_ c0_a3 a2 a1 a0_b3 b2 b1 b0_c_in;

 Array_of_ASIC_Test_Patterns [0] = 14'b0100_1_1010_1010_0;
 Array_of_ASIC_Test_Patterns [1] = 14'b0000_0_0000_0000_0;
 Array_of_ASIC_Test_Patterns [2] = 14'b1111_1_1111_1111_1;
 Array_of_ASIC_Test_Patterns [3] = 14'b0100_1_0101_0101_0;
end endtask

/****************************** INSTRUCTION PATTERNS ******************************/
 parameterBYPASS= 3'b111;// pattern_ptr = 0
 parameterEXTEST= 3'b001;// pattern_ptr = 1
 parameterSAMPLE_PRELOAD= 3'b010;// pattern_ptr = 2
 parameterINTEST= 3'b011;// pattern_ptr = 3
 parameterRUNBIST= 4'b100;// pattern_ptr = 4
 parameterIDCODE= 5'b101;// pattern_ptr = 5

task Load_TAP_Instruction;
 begin
  Pattern_Register = Array_of_TAP_Instructions [pattern_ptr];
  @ (negedge TCK ) load_TDI_Generator = 1;
  @ (negedge TCK) load_TDI_Generator = 0;
 end
endtask

task Declare_Array_of_TAP_Instructions;
 begin
  Array_of_TAP_Instructions [0] = BYPASS;
  Array_of_TAP_Instructions [1] = EXTEST;
  Array_of_TAP_Instructions [2] = SAMPLE_PRELOAD;
  Array_of_TAP_Instructions [3] = INTEST;
  Array_of_TAP_Instructions [4] = RUNBIST;
```

```
    Array_of_TAP_Instructions [5] = IDCODE;
   end
   endtask
endmodule

module TDI_Generator #(parameter BSC_Reg_size = 14)(
  output                                to_TDI,
  input       [BSC_Reg_size -1: 0]      scan_pattern,
  input                                 load, enable_bypass_pattern, TCK
);
  reg         [BSC_Reg_size -1: 0]      TDI_Reg;
  wire                                  enableTDO = t_ASIC_with_TAP.M0.enable
                                          TDO;
  assign to_TDI = TDI_Reg [0];

  always @ (posedge TCK) if (load) TDI_Reg <= scan_pattern;
   else if (enableTDO || enable_bypass_pattern)
    TDI_Reg <= TDI_Reg >> 1;
endmodule

module TDO_Monitor #(parameter BSC_Reg_size = 14)(
  input                                 to_TDI,
  input                                 from_TDO, strobe, TCK
);
  reg         [BSC_Reg_size -1: 0]      TDI_Reg, Pattern_Buffer_1,
                                          Pattern_Buffer_2,
                                        Captured_Pattern, TDO_Reg;
  reg                                   Error;
  parameter                             test_width = 5;
  wire                                  enableTDO = t_ASIC_with_TAP.M0.enable
                                          TDO;
  wire        [test_width -1: 0]        Expected_out =
                                        Pattern_Buffer_2 [BSC_Reg_size -1
                                        : BSC_Reg_size - test_width];
  wire        [test_width -1: 0]        ASIC_out =
                                        TDO_Reg [BSC_Reg_size - 1:
                                          BSC_Reg_size - test_width];
  initial                               Error = 0;

  always @ (negedge enableTDO) if (strobe == 1) Error = |(Expected_out ^
   ASIC_out);
  always @ (posedge TCK) if (enableTDO) begin
   Pattern_Buffer_1 <= {to_TDI, Pattern_Buffer_1 [BSC_Reg_size -1: 1]};
   Pattern_Buffer_2 <= {Pattern_Buffer_1 [0], Pattern_Buffer_2 [BSC_Reg_size -1: 1]};
   TDO_Reg <= {from_TDO, TDO_Reg [BSC_Reg_size -1: 1]};
  end
endmodule
```

Note that the ASM chart of the TAP controller (see Figure 11-51) has the property that the value of *TMS* that causes a transition into a state of the chart is the same

for all paths that enter the state. With this observation, we write a set of testbench tasks to specify a sequence of inputs directing the flow along arcs of the ASM chart. The test patterns in the testbench conform to the port structure shown in Figure 11-54. The patterns demonstrate the flow of data in *ASIC_with_TAP* and demonstrate that the testbench detects an error that has been injected into the circuit. The instruction patterns conform to the instruction code in the model for *Instruction_Decoder*. The test sequences are generated by *TDI_Generator* and monitored by *TDO_Monitor*, and given below. The task *Load_ASIC_Test_Pattern* executes a test sequence to select and load a scan pattern into the register *TDI_Reg* within *TDI_*Generator. This pattern scans out of *TDI_Generator* while *enableTDO* or *enable_bypass_pattern* is asserted. *TDO_*Monitor includes a two-stage pipeline buffer whose input stage receives the pattern that is shifted into the ASIC. The first stage holds the pattern that is currently in the boundary scan register of *ASIC_with_TAP*; the second stage holds the previous pattern held by *ASIC_*with_*TAP*, and is used to compare the actual pattern scanned from *ASIC_with_TAP* to the expected pattern. Data from the cells of the boundary scan register that correspond to the outputs of the ASIC are compared with the test pattern data for those cells. A mismatch is detected as an error. The testbench includes an optional segment of code that injects an error into the bits corresponding to the sum generated by the adder, and that checks whether an error is detected by *TDO_Monitor*.

The simulation results shown in Figure 11-55 demonstrate that the default instruction is the BYPASS instruction. The signals *c_in*, *b*, *a*, *c_out*, and sum are external ports of *ASIC_with_TAP*; *c_in_to_ASIC*, *b_to_ASIC*, and *a_to_ASIC* are input ports of *ASIC*, and *c_out_fr_ASIC* and *sum_fr_ASIC* are output ports of *ASIC*. The system resides in an unknown state when simulation initiates at time $t = 0$. At the first active edge of *TCK* the state of the machine enters *S_Reset* (0),[26] where it remains (Figure 11-55(a)) until the sequence of inputs of *TMS* scans 10 bits of the *Pattern_Register* (1354_H) through the TAP (Figure 11-55(b)).[27] In the testbench, *Pattern_Register* holds the pattern selected by *pattern_ptr*. The pulse of *Load_TDI_Generator* loads the pattern into *TDI_Reg* (within *TDI_Generator*). The LSB of *TDI_Reg* drives *TDI*. After the machine enters *S_Shift_DR* (4), 10 cycles of *TCK* with *shiftDR* asserted scan ten bits of the pattern through the bypass register. Note in Figure 11-55(a) that the state transitions occur on the rising edges of *TCK*, and that the waveform of *TDO* is a copy of the waveform of *TDI* delayed by one cycle while *enableTDO* is asserted. Also note that *clock_BSC_Reg* is fixed (i.e., the boundary scan register is idle).

The JTAG specification for the bypass register requires that the output of the register be set to logical 0 on the rising edge of *TCK* following entry into the TAP controller *state S_Capture_DR*. Note in Figure 11-56(a) that this edge occurs at the transition between *S_Capture_DR* and *S_Shift_DR*, and that the output of the bypass register is 0. The output of the register will be the value scanned out of *TDO* and the next rising edge of *TCK* following assertion of *TDO_enable*.

[26]See Problem 21 at the end of this chapter.

[27]The data patterns and test sequence intervals have been chosen to illustrate the operation of the TAP.

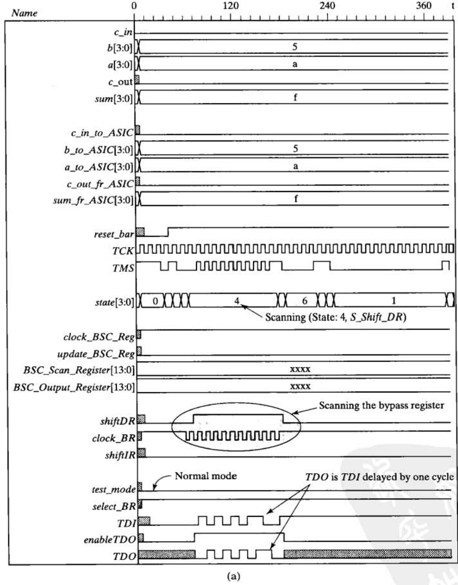
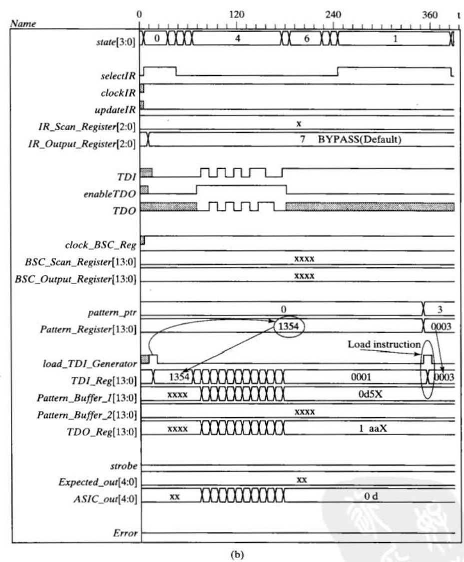


FIGURE 11-55 Simulation results—scanning a pattern through the bypass register of *ASIC_with_TAP* after power-up: (a) the pattern scans through the chip with a delay of one clock cycle and (b) control signals, TAP registers, and testbench registers.



(b)

FIGURE 11-55 Continued

The scanning process does not affect the values of the signals at the ports of *ASIC*. *clockBR* is active for one cycle in state *S_Capture_DR* (3) and for 10 cycles in state *S_Shift_DR* (4). *selectBR* connects the bypass register to *TDI* and *TDO*. *BSC_Scan_Register* and *BSC_output_Register* hold 14'Hx because they have not yet

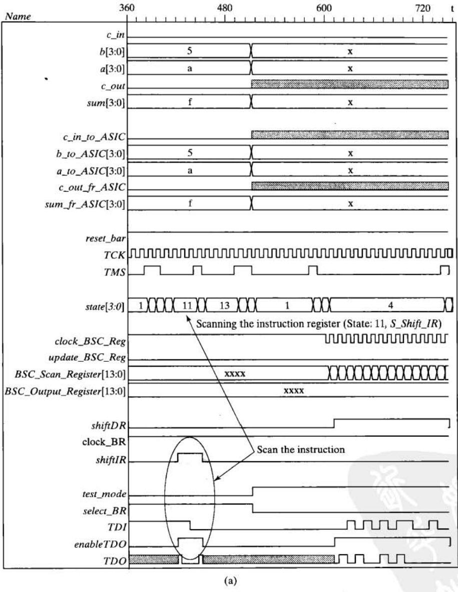
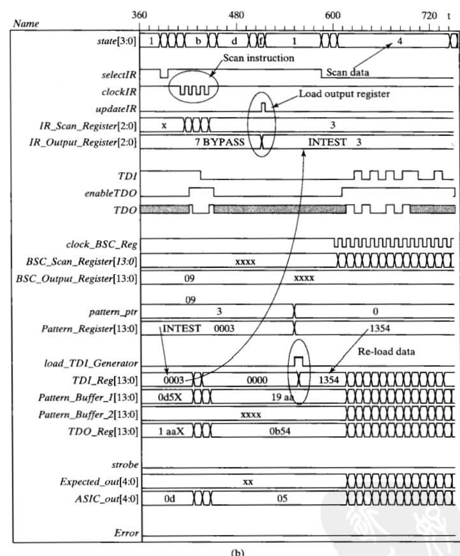


FIGURE 11-56 Simulation results—loading the instruction INTEST into the instruction register. (a) *enableTDO* is asserted only while scanning (otherwise *TDO* is in the high-impedance state), and (b) the instruction INTEST is loaded, then a data pattern is reloaded and scanned out of the *TDI_Generator* and into *ASIC_with_TAP*, via *TDI*.



(b)

FIGURE 11-56 Continued

received data. *reset_bar* is asserted (active-low) and resets the 3-bit instruction register to hold all 1s (the BYPASS instruction) in state *S_Reset* (0). The subsequent bits of *TDO* replicate the waveform of *TDI*.

The simulation results in Figure 11-57(a) show the BYPASS instruction being shifted out of the TAP and the instruction INTEST being loaded into the TAP while

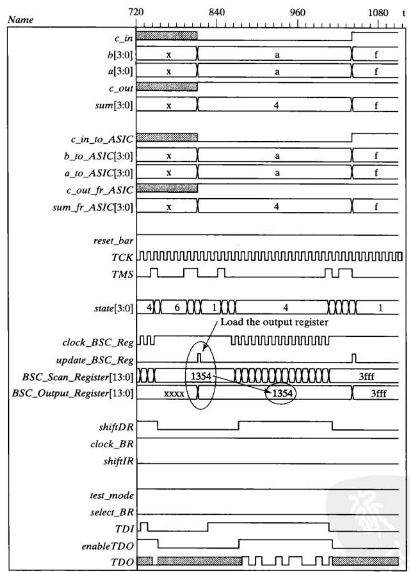
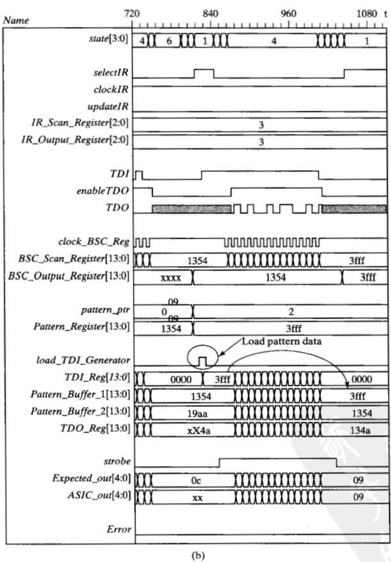


FIGURE 11-57 Simulation results: (a) after scanning the test pattern 1354_H into the capture/scan register, the boundary scan output register is loaded, and test inputs are applied to *ASIC* and (b) the outputs of *ASIC* are captured and scanned out through *TDO*, and shifted into *TDO_Reg* for comparison with *Pattern_Buffer_2* (see Figure 11.66).

the machine is in the state *S_Shift_IR* (11), with *shiftIR* and *enableTDO* asserted. The testbench loads *Pattern_Register* with INTEST, then asserts *load_TDI_Generator* to load *TDI_Reg* with INTEST. In state *S_Shift_IR*(11) the instruction scans into *IR_SCAN_Register* (Figure 11-57(b)) while BYPASS scans out of the register through



(b)

FIGURE 11-57 Continued

TDO. When the state of the TAP controller enters *S_Update_IR*, the instruction INTEST is loaded into *IR_Output_Register*. The waveforms in Figure 11-57(b) also show the testbench reloading *Pattern_Register* with 1354_H, transferring the pattern to *TDI_Reg*, and scanning the pattern into the *BSC_Scan_Register* while the state resides in *S_Shift_IR*. Also note that the three-state behavior of *TDO* conforms to the JTAG standard.

With *IR_Output_Register* holding the instruction INTEST, the pattern (1354_H) that was loaded into *BSC_Scan_Register* in Figure 11-56(b) is transferred to *BSC_Update_Register* in Figure 11-57(a) for execution of an internal test of *ASIC*. Note that the values at *c_in_to_ASIC*, *b_to_ASIC*, and *a_to_ASIC* change to the values specified by the applied test pattern, and that *c_out_fr_ASIC* and *sum_fr_ASIC* are produced by the adder within ASIC.[28] A second test pattern ($3fff_H$) is loaded into *TDI_reg* (see Figure 11-57(b)) and shifted into *BSC_Scan_Register* while the results of the previous test pattern are scanned out.

The test process is completed in Figure 11-58. The new test pattern is loaded into *BSC_Output_Register* (see Figure 11-58(a)), and *Expected*_out is compared to *ASIC_out* in Figure 11-58(b). The patterns match, and *Error* remains de-asserted.[29]

11.8.8 Design Example: Built-In Self-Test

Built-in self-test (BIST) logic allows an ASIC to test itself. BIST circuitry is used when it is not practical or possible to test an ASIC with an external tester. Some circuits must be tested in the field each time the host system is restarted; others must be tested as part of a board environment. For example, computers and other sequential machines use BIST to test block RAMs on power-up.

A simple architecture for a machine with BIST hardware is shown in Figure 11-59. In normal mode, the unit under test (UUT) is driven by external (primary) inputs, but in test mode, patterns are generated by built-in circuitry and applied to the circuit. The response of the circuit is monitored by additional hardware and compared to an expected response to the input pattern. A difference between the expected and actual response patterns indicates the presence of an internal fault. A controller (state machine) governs the overall process of applying patterns and observing the response of the machine.

The pattern generator for BIST can be implemented by storing stimulus patterns in memory and retrieving them during test mode, but that approach requires a relatively large amount of memory compared to other alternatives. We will consider an alternative using a linear feedback shift register (LFSR)[30] as a pseudo-random pattern generator (PRPG), and a multiple-input signature register (MISR) to monitor patterns. LFSRs are used as PRPGs because they require a small amount of hardware to generate a large set of patterns.

[28]The adder implemented in this example has 0 delay.

[29]The testbench includes an example in which a fault is injected into *ASIC* and detected by an applied test pattern.

[30]See Chapter 5.

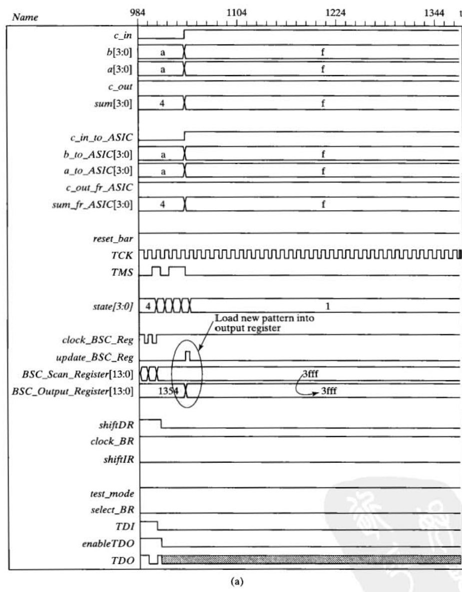


FIGURE 11-58 Simulation results—after scanning a second data pattern into *ASIC_with_TAP*: (a) the pattern ($3fff_H$ is loaded into *BSC_Output_Register*) and (b) the expected and actual outputs of *ASIC* match.

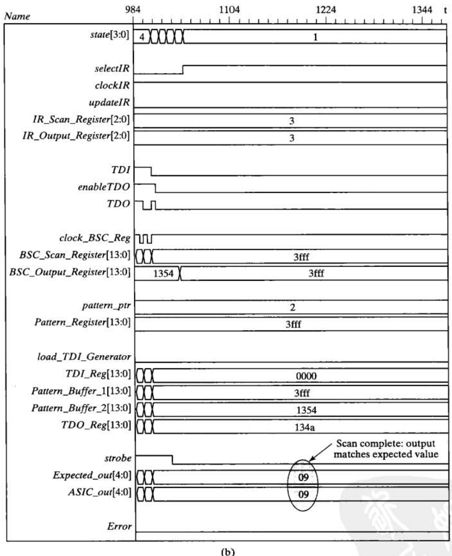


FIGURE 11-58 Continued

The coefficients of an n-bit autonomous LFSR can be chosen to produce a pseudo-random sequence of n-bit patterns that repeats after $2^n - 1$ steps (i.e., the sequence of patterns is cyclic). This method for generating patterns is attractive because the hardware required to generate the sequence of patterns is significantly less

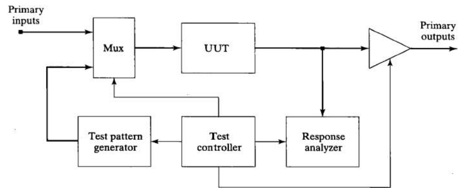


FIGURE 11-59 Architecture for a BIST machine.

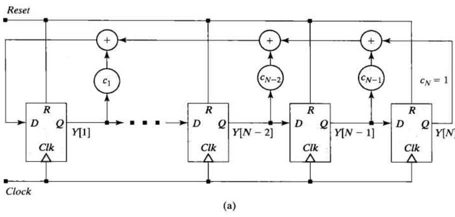
than the hardware that would be required to store the same patterns in memory. LFSRs that have an irreducible and primitive characteristic polynomial produce a sequence of patterns having maximum length [16].

Two types of LFSRs are shown in Figure 11-60. A Type I LFSR augments an ordinary shift register with "external" exclusive-or gates. This type of LFSR can use the same register for ordinary operation. The Type-I shift register in Figure 11-60(a) is tapped to feed cell outputs back to the first (left most) cell in the chain. The Type II structure shown in Figure 11-60(b) has exclusive-or gates in the shift register path at locations where the tap coefficients have the value 1. Both structures generate maximum-length pseudorandom binary sequences, depending on the tap coefficients.

Type II LFSRs are preferred in testing because their patterns are more random (i.e., less correlated) than those produced by the Type I machine [4]. Shift register tap coefficients that generate maximal-length pseudorandom binary sequences are given in Table 11-11. Note that the tap coefficients of the two types of LFSRs are labeled in ascending order in opposite directions in Figure 11-60.

The response of a BIST-driven circuit can be compared to its expected response to determine whether the circuit is operating correctly. Instead of storing the patterns of the expected responses, a MISR compresses the patterns generated by the circuit to form a signature [6]. The signature of a correctly operating circuit is stored for comparison to the actual response. Thus, the MISR circuit and the signature eliminate the need to monitor and compare the responses of the individual test patterns. The MISR in Figure 11-61 is driven by the response vectors of the circuit. The state *Y* of the circuit after a pattern has been applied is the circuit's signature.

The machine *ASIC_with_BIST* shows how an ASIC can be combined with additional hardware for built-in self-test. For simplicity, the ASIC will be modeled as a 4-bit adder with carries in and out. Figure 11-62 shows the architecture of *ASIC_with_BIST*, including ports for the adder's datapaths, a signal *test_mode*, which controls whether *ASIC_with_BIST* is operating in test mode or in normal mode, and a signal *reset* that drives an internal state machine to a reset state. The signal *done* asserts for one cycle of



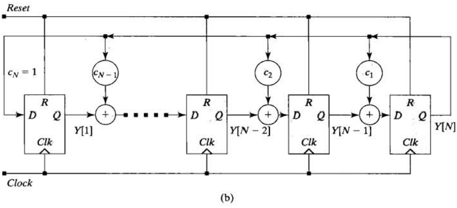


FIGURE 11-60 Linear feedback shift registers: (a) Type I (external xor gates) and (b) Type II (internal xor gates).

clock to indicate that the BIST test sequence is complete; *error* indicates that the signature produced by *Response_Analyzer* does not match the expected stored signature for the sequence of test vectors generated by the BIST circuit.

The model of *ASIC_with_BIST* includes Verilog modules *ASIC*, *Pattern_Generator*, *Response*_Analyzer, and *BIST_Control_Unit*. The BIST implementation does not modify *ASIC*, the circuit that is to be tested by the BIST hardware. *Pattern_Generator* is a customized LFSR, with the parameter *size*, specifying the size of the datapaths of the adder in

TABLE 11-11 Tap coefficients for maximum-length pseudo-random binary sequence generators.

n	Coefficient Vectors ($C_n \ldots C_2 C_1$)	Coefficents
2	11	$C_2 C_1$
3	101	$C_3 C_1$
4	1001	$C_4 C_1$
5	1_0010	$C_5 C_2$
6	10_0001	$C_6 C_1$
7	100_0100	$C_7 C_3$
8	1000_1110	$C_8 C_4 C_3 C_2$
9	1_0000_1000	$C_9 C_4$
10	10_0000_0100	$C_{10} C_3$
11	100_0000_0010	$C_{11} C_2$
12	1000_0010_1001	$C_{12} C_6 C_4 C_1$
13	1_0000_0000_1101	$C_{13} C_4 C_3 C_1$
14	10_0010_0010_0001	$C_{14} C_{10} C_6 C_1$
15	100_0000_0000_0001	$C_{15} C_1$
16	1000_1000_0000_0101	$C_{16} C_{12} C_3 C_1$
32	1000_0000_0010_0000_0000_0000_0000_0011	$C_{32} C_{22} C_2 C_1$

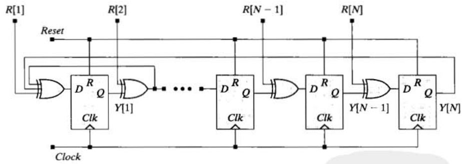


FIGURE 11-61 Multiple-input linear feedback shift register (MISR).

ASIC; *Length*, specifying the length of the shift register; and *initial_state*, specifying the state that results when an external reset is asserted, The maximum-length LFSR in *Pattern_Generator* will generate stimulus patterns, and a MISR in *Response_Analyzer* will generate a signature. At the end of the test sequence, *BIST_Control_Unit* will compare the signature and the stored pattern and assert an error signal if they do not match. The multiplexer and the three-state output buffer in Figure 11-62 are modeled by Verilog continuous-assignment statements in *ASIC_with_BIST*.

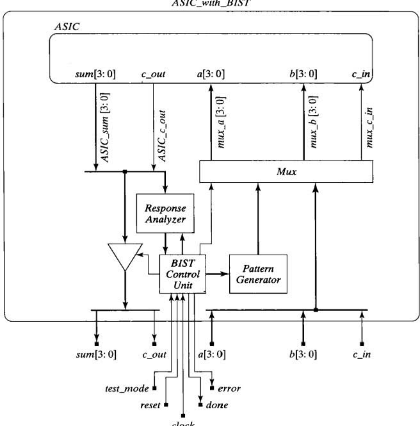


FIGURE 11-62 Architecture for *ASIC_with_BIST*.

The ASM chart in Figure 11-63 describes the state-machine controller for *ASIC_with_BIST*. The signals *clock*, *reset*, and *test_mode* are driven by the host environment. The BIST circuit includes a counter, which determines the length of the test sequence. The state remains in *S_test* while patterns are applied and then transitions to *S_compare*, where the signature produced by *Response_Analyzer* is compared to *stored_pattern*. If the patterns match, the state transitions to *S_done* and asserts the Moore-type output *done* for one clock cycle before returning to

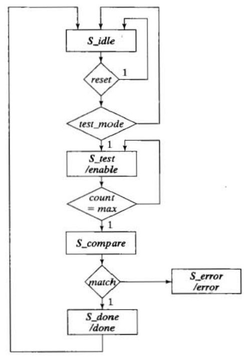


FIGURE 11-63 ASM chart for the controller of *ASIC_with_BIST*.

S_idle. If the patterns do not match, the state transitions to *S_error*, where it remains until *reset* is asserted.

```
module ASIC_with_BIST #(parameter size = 4)(
  output    [size -1: 0]      sum,     // ASIC interface I/O
  output                      c_out,
  input     [size -1: 0]      a, b,
  input                       c_in,
  output                      done, error,
  input                       test_mode, clock, reset
);
  wire      [size -1: 0]      ASIC_sum;
  wire                        ASIC_c_out;
  wire      [size -1: 0]      LFSR_a, LFSR_b;
  wire                        LFSR_c_in;
  wire      [size -1: 0]      mux_a, mux_b;
  wire                        mux_c_in;
```

```
wire                     enable;
wire     [1: size +1]    signature;
assign   {sum, c_out} = (test_mode) ? 'bz : {ASIC_sum, ASIC_c_out};
assign   {mux_a, mux_b, mux_c_in} = (enable == 0) ? {a, b, c_in} :
          {LFSR_a, LFSR_b, LFSR_c_in};
ASIC M0 (
  .sum (ASIC_sum),
  .c_out (ASIC_c_out),
  .a (mux_a),
  .b (mux_b),
  .c_in (mux_c_in));
Pattern_Generator M1 (
  .a (LFSR_a),
  .b (LFSR_b),
  .c_in (LFSR_c_in),
  .enable (enable),
  .clock (clock),
  .reset (reset)
);
Response_Analyzer M2 (
  .MISR_Y (signature),
  .R_in ({ASIC_sum, ASIC_c_out}),
  .enable (enable),
  .clock (clock),
  .reset (reset));
BIST_Control_Unit  M3 (done, error, enable, signature, test_mode, clock, reset);
endmodule

module ASIC #(parameter size = 4)(
 output    [size -1: 0]    sum,
 output                    c_out,
 input     [size -1: 0]    a, b,
 input                     c_in
);
 assign {c_out, sum} = a + b + c_in;
endmodule

module Response_Analyzer #(parameter size = 5)(
 input              [1: size] R_in,
 input                        enable, clock, reset
);
 always @  (posedge clock)
  if (reset == 0) MISR_Y <= 0;
  end
endmodule
```

```
module Pattern_Generator #(parameter size = 4, Length = 9)(
  output      [size -1: 0]      a, b,
  output                        c_in,
  input                         enable, clock, reset
);
  reg         [1: Length]       LFSR_Y;
  parameter [1: Length]         initial_state = 9'b1_1111_1111;
  parameter [Length: 1]         Tap_Coefficient = 9'b1_0000_1000;
  integer                       k;
  assign a = LFSR_Y[2: size + 1];
  assign b = LFSR_Y[size + 2: Length];
  assign c_in = LFSR_Y[1];
  always @ (posedge clock)
   if (reset == 1'b0) LFSR_Y <= initial_state;
   else if (enable) begin
    for (k = 2; k <= Length; k = k + 1)
     LFSR_Y[k] <= Tap_Coefficient[Length -k +1]
     ? LFSR_Y[k -1] ^ LFSR_Y[Length] : LFSR_Y[k -1];
    LFSR_Y[1] <= LFSR_Y[Length];
   end
endmodule

module BIST_Control_Unit #( parameter sig_size = 5, c_size = 10, size = 3,
  c_max = 510)(
  output reg              done, error, enable,
  input      [1: sig_size]      signature,
  input                         test_mode, clock, reset
);
  parameter  stored_pattern = 5'h1a;  // signature if fault-free
  parameter  S_idle = 0,
                   S_test = 1,
                   S_compare = 2,
                   S_done = 3,
                   S_error = 4;
  reg        [size -1: 0]       state, next_state;
  reg        [c_size -1: 0]     count;
  wire       match = (signature == stored_pattern);

  always @ (posedge clock) if (reset == 0) count <= 0;
   else if (count == c_max) count <= 0;
   else if (enable) count <= count + 1;

  always @ (posedge clock) if (reset == 0) state <= S_idle;
   else state <= next_state;

  always @ (state,  test_mode, count, match) begin
   done = 0;
```

```
      error = 0;
      enable = 0;
      next_state = S_error;
      case (state)
       S_idle:      if (test_mode) next_state = S_test; else next_state = S_idle;
       S_test:      begin enable = 1; if (count == c_max -1) next_state = S_compare;
                   else next_state = S_test; end
       S_compare:  if (match) next_state = S_done;
                   else next_state = S_error;
       S_done:      begin done = 1; next_state = S_idle; end
       S_error:      begin done = 1; error = 1; end
      endcase
    end
  endmodule
```

The testbench for *ASIC_with_BIST* executes the following tests: (1) power-up reset, (2) reset on-the-fly, (3) three-state action of *sum* and *c_out* and selection of the input datapath when *test_mode* is asserted, (4) initiation of activity in the LFSR pattern generator and the MISR when *enable* is asserted by *BIST_Control_Unit*, and (5) detection of an injected fault at an input pin of *ASIC*.

```
module t_ASIC_with_BIST #(parameter size = 4, End_of_Test = 11000);
 wire        [size -1: 0]        sum;      // ASIC interface I/O
 wire        c_out;
 reg         c_in;
 wire        done, error;
 reg         test_mode, clock, reset;
 reg         Error_flag = 1;

 ASIC_with_BIST M0 (sum, c_out, a, b, c_in, done, error, test_mode, clock, reset);

 initial begin Error_flag = 0; forever @ (negedge clock) if ( M0.error)
  Error_flag = 1; end
 initial #End_of_Test $finish;
 initial begin clock = 0; forever #5 clock = ~clock; end
 // Declare external inputs
 initial fork
   a = 4'h5;
   b = 4'hA;
   c_in = 0;
   #500 c_in = 1;
 join
// Test power-up reset and launch of test mode
 initial fork
   #2 reset = 0;
```

```
        #10 reset = 1;
        #30 test_mode = 0;
        #60 test_mode = 1;
      join
    // Test action of reset on-the-fly
      initial fork
        #150 reset = 0;
        #160 test_mode = 0;
      join

    // Generate signature of fault-free circuit
      initial fork
        #180 test_mode = 1;
        #200 reset = 1;
      join
    // Test for an injected fault
      initial fork
        #5350 release M0.mux_b [2] ;
        #5360 force M0.mux_b[0] = 0;
        #5360 begin reset = 0;  test_mode = 1; end
        #5370 reset = 1;
      join
    endmodule
```

The simulation results in Figure 11-64 demonstrate that power-up reset drives the state of the *BIST_Control_Unit* to *S_idle* (0), and resets the state of the LFSR to $1ff_H$. When *test_mode* is asserted the state transitions to *S_test* (1), where *enable* is asserted. The assertion of *enable* connects the datapath from the LFSR (see *mux_a*, *mux_b*, and *mux_c_in*) to the ports of *ASIC* and drives the output datapath (see *sum* and *c_out*) of *ASIC_with_BIST* into the high-impedance state. With *enable* asserted, the LFSR generates patterns driving the internal datapath for *ASIC_sum* and *ASIC_c_out*, and the MISR within *Response_Analyzer* begins to generate preliminary signatures. A second assertion of *reset* demonstrates that the machine resets on-the-fly.

The simulation results in Figure 11-65 demonstrate that the signature of the fault-free circuit matches the stored pattern. After 510 clock cycles the state enters *S_compare* (2), detects a match, and then enters *S_done* (3) for one cycle, before returning to *S_idle*. The testbench includes *Error_flag* to detect an error when multiple test sequences are applied to detect injected faults. The simulation results in Figure 11-66 were generated after a fault was injected at an input of *ASIC*. They demonstrate that the machine detects the mismatch between the stored pattern of the fault-free machine and the signature produced by the MISR when the machine has the injected fault.

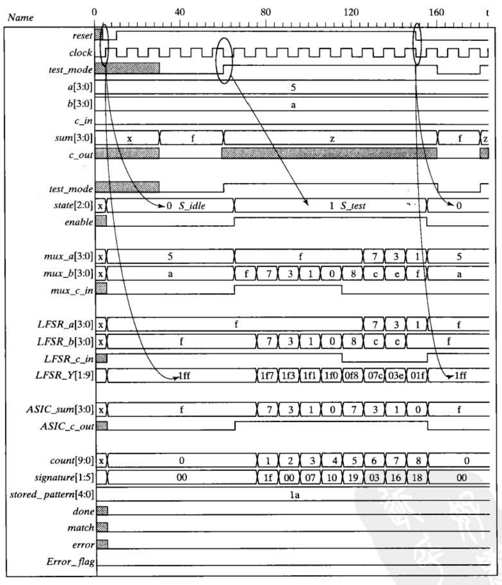


FIGURE 11-64 Simulation results showing (1) power-up reset, (2) reset on-the-fly, (3) three-state action of *sum* and *c_out* and selection of the input datapaths when *test_mode* is asserted, and (4) initiation of activity in the LFSR and the MISR when *test_mode* is asserted in *ASIC_with_BIST*.

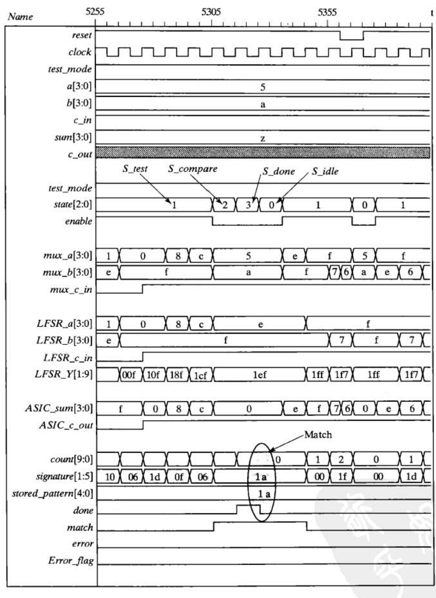


FIGURE 11-65 Simulation results showing match between the stored pattern and the signature of the fault-free circuit in *ASIC_with_BIST*.

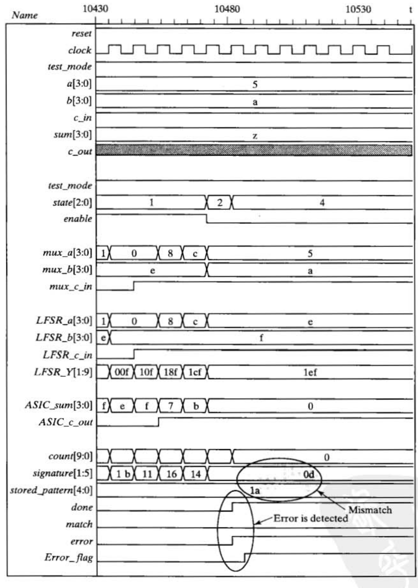


FIGURE 11-66 Simulation results showing detection of an injected fault in *ASIC_with_BIST*.

REFERENCES

1. Howe H. "Pre- and Postsynthesis Simulation Mismatches." Proceedings of the Sixth International Verilog HDL Conference, March 31–April 3, 1997, Santa Clara, CA.
2. McWilliams TM. "Verification of Timing Constraints on Large Digital Systems." Ph.D. Thesis, Stanford University, 1980.
3. Osterhout JK. "Crystal: A Timing Analyzer for nMOS VLSI Circuits." In: Bryant R, ed. *Proceedings of the Third Caltech Conference on VLSI*. Rockville, MD: Computer Science Press, 1983, 57–69.
4. Smith MJS. *Application-Specific Integrated Circuits*. Reading, MA: Addison-Wesley, 1997.
5. Xilinx University Program Workshop Notes—ISE 3.1i, Spring 2001.
6. Abramovici M, et al. *Digital Systems Testing and Testable Design*. New York: Computer Science Press, 1994.
7. Roth JP. "Diagnosis of Automatic Failures: A Calculus and a Method." *IBM Journal of Research and Development*, 10, 1966, 278–281.
8. Goel P. "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits." *IEEE Transactions on Computers*, C-30, 215–222, 1983.
9. Fujiwara H. *Logic Testing and Design for Testability*. Cambridge, MA: MIT Press, 1985.
10. Fujiwara H, Shimono T. "On the Acceleration of Test Generation Algorithms." *IEEE Transactions on Computers*, C-32, 1983,1137–1144.
11. Williams TW, Brown NC. "Defect Level as a Function of Fault Coverage." *IEEE Transactions on Computers*, C-30, 987–988, 1981.
12. Goldstein LH. "Controllability/Observability Analysis of Digital Circuits." *IEEE Transactions on Circuits and Systems*, CAS-26, 685–603, 1979.
13. *IEEE 1149.1–1990, IEEE Test Access Port and Boundary-Scan Architecture*. Piscataway, NJ: Institute of Electrical and Electronics Engineers, 1990.
14. Wakerly JF. *Digital Design Principles and Practices*. 4th ed. Upper Saddle River, NJ: Prentice-Hall, 2006.

PROBLEMS

1. The gates in Figure P11-1 are annotated with (min:max) delay ranges for rising and falling output transitions. Develop a DAG for the circuit, and enumerate the delay ranges of the paths through the circuit for rising and falling transitions of the outputs.
2. Using a static timing analyzer, determine the maximum clock frequency at which a synthesized implementation of *ALU_machine_4_bit* can operate without a timing violation (see Problem 7 in Chapter 8).e
3. Using a static timing analyzer, determine the maximum clock frequency at which a synthesized implementation of *UART_Transmitter_Arch* can operate without a timing violation (see Chapter 7).

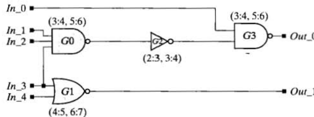


FIGURE P11-1

4. Using a static timing analyzer, determine the maximum clock frequency at which a synthesized implementation of *UART_8_Receiver* can operate without a timing violation (see Chapter 7).
5. Gated clocks can be problematic in ASICs and FPGAs. Compare the circuits shown in Figure P11-5, which can be used to generate a clock pulse when a binary counter reaches a specified count.
6. Conduct a static timing analysis of a 4-bit ripple-carry adder and a 4-bit carry look-ahead adder. The adders are to be implemented using cells from a CMOS standard-cell library (include the source code for the models of the adders). List in the table below the cells that are used in your design, along with their propagation delays (indicate the physical units).
   a. Using a static timing analyzer (e.g., Synopsys PrimeTime)
      i. Create a timing analysis report and a distribution of the path lengths of the ripple carry adder. Find the longest path, identify it below, and indicate its delay.

      Longest path begins at:_____
      Longest path ends at:_____
      Delay of the longest path_____ ns

      ii. Find the shortest path, identify it below, and indicate its delay.

      Shortest path begins at:_____
      Shortest path ends at:_____
      Path delay:_____ ns

      iii. Create a timing analysis report and a distribution of the path lengths of the carry look-ahead adder. Find the longest path, identify it below, and indicate its delay.

      Longest path begins at:_____
      Longest path ends at:_____
      Delay of the longest path_____ ns

      iv. Find the shortest path, identify it below, and indicate its delay.

      Shortest path begins at:_____
      Shortest path ends at:_____
      Path delay:_____ ns

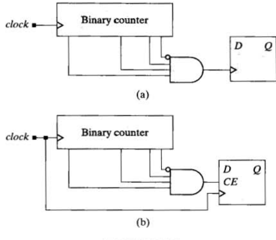


FIGURE P11-5

TABLE P11-6

Cell name	Propagation Delay (Rising)	Propagation Delay (Falling)

b. Using cell area data, compare the areas of the two implementations.

Area (ripple carry)_____
Area (look-ahead)_____
Provide data showing how the area was calculated.

c. Develop a testbench for the designs, and identify below input patterns that will exercise the longest and shortest paths, and the delays that are observed *in simulation* with these patterns. Provide waveforms of the results (annotate the results to show the delay).

Pattern for longest path:_____
Delay observed in simulation:_____
Pattern for shortest path:_____
Delay observed in simulation:_____

d. For 16-bit ripple carry (RCA) and carry look-ahead (CLA) adders implemented in 4-bit slices, find the shortest clock cycle under which data can be fetched from a register, added, and then stored in a register (see Figure 11-7), using the flip-flops of the cell library. (*Note*: The registers are not part of the adder). Provide annotated simulation results showing the above delays.

Shortest clock cycle:RCA:_____
Shortest clock cycle:CLA:_____

e. Discuss the significant differences between the adders.

7. Find a minimum set of test patterns that will cover all of the *s-a-0* and *s-a-1* faults in the gate-level model of a full adder circuit shown in Figure P11-7.
8. The circuit in Figure P11-8 was developed in Example 2.34, where the redundant logic of the AND gate driving *F2* was added to the circuit to cover a hazard. Determine whether the *s-a-0* fault on the input to the AND gate is detectable.

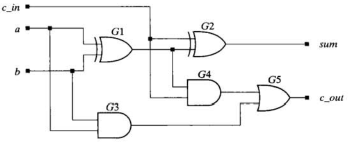


FIGURE P11-7

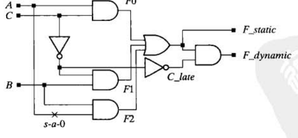


FIGURE P11-8

9. Develop tests for the faults *G4 s-a-0* and *G4 s-a-1* in Figure P11-7.
10. Verify that *ASIC_with_TAP* (see Section 11.8.7) is synthesizable. Check for latches in the netlist of the synthesized circuit.
11. What feature of the JTAG TAP controller (see Figure 11-51) enables the machine to enter *S_Reset* at the first active edge of the clock following power-up, in the absence of the optional active-low external asynchronous reset *nTRST*?
12. The model for *ASIC_with_TAP* (see Section 11.8.7) generates gated clocks in *Instruction_Decoder*. Develop, verify, and synthesize *ASIC_with_TAP_NGC*, a model that does not have gated clocks.
13. The model for *ASIC_with_TAP* (see Section 11.8.7) has parameters in several modules. Consequently, each module must be edited to resize the design. Using the ***defparam*** construct to define all of the parameters of ASIC_with_TAP, develop and include an annotation module in the testbench *t_ASIC_with_TAP* to parameterize the design.
14. The model *ASIC_with_TAP* (see Section 11.10.6) includes pull-up devices for *TMS*, *TDI*, and *nTRST*. Develop a testbench to verify that the models conform to the JTAG specification that e(at *TMS*, *TDI*, or *nTRST*) shall be identical to the application of a logic 1.
15. The state of the JTAG TAP controller enters *S_Reset* on power-up and remains there until *TMS* is asserted low. Using the ASM chart in Figure 11-63, explain how the host system is protected from a glitch on *TMS* (i.e., *TMS* inadvertently changes to 0 for one clock cycle before returning to 1).
16. Develop *ALU_4_bit_with_JTAG*, a JTAG-enhanced version of *ALU_4_bit* (see Table P8.7a in Chapter 8 for a functional specification of *ALU_4_bit*), and a testbench, *t_ALU_4_bit_with_JTAG* having scan patterns that exhaustively verify the functionality of the opcode *A_and_B* and tests for any *s-a-0* or *s-a-1* fault that can be detected at the output. Include in the testbench a test demonstrating that a fault injected at an output pin of *ALU_4_bit* will be detected.
17. Explain why the state *S_Exit1_DR* is needed in the ASM chart of the TAP controller.
18. Develop a model for *Board_with_Four_ASICS*, a module that contains four copies of *ASIC_with_TAP* (see Section 11.10.6) connected in a ring configuration (see Figure 11-62). The ASICs are to be connected to form a 16-bit ripple carry adder with the port structure (*sum[15:0], c_out, a[15:0], b[15:0], c_in*). Develop a testbench, *t_Board_with_Four_ASICS*, and test sequences that (1) bypass all four chips, (2) bypass all but the chip producing the most significant bit-slice of the board's output, (3) test the interconnect between the ASICs producing the least significant 8 bits of the machine, (4) tests the ASIC producing *sum[7:4]* for internal faults (use gate-level models for the bit-slice adders). Using the ***force . . . release*** construct, the testbench is to inject faults as needed to demonstrate that faults are detected.
19. Repeat Problem 11-17, but with the copies of *ASIC_with _TAP* connected in a star configuration.

20. The Type I and Type II LFSRs shown in Figure 11-60 do not enter the state $Y = 0$, because they could not exit from that state. Explain how the modified Type I LFSR shown in Figure P11-20 enters and leaves the state $Y = 0000$.

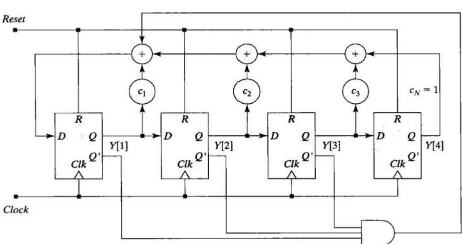


FIGURE P11-20

PDG

APPENDIX A

Verilog Primitives

Verilog has a set of 26 primitives for modeling the functionality of combinational and switch-level logic. The output terminals of an instantiated primitive are listed first in its primitive terminal list. The input terminals are listed last. The ***buf*** and ***not*** primitives ordinarily have a single input, but may have multiple scalar outputs. The other primitives may have multiple scalar inputs, but have only one output. In the case of the three-state primitives (***bufif1, bufif0, notif1, notif0, tranif1, tranif0, rtranif1, rtranif0***), the control input is the last input in the terminal list. When a vector of primitives is instantiated the ports may be vectors. If the inputs and outputs of a primitive are vectors, the output vector is formed on a bitwise basis from the input's vector.

Primitives may be instantiated with propagation delay, and may have strength assigned to their output net(s). Their input–output functionality in Verilog's four-valued logic system is defined by the following truth tables, where the symbol L represents 0 or z, and the symbol H represents 1 or z. These additional symbols accommodate simulation results in which a signal can have a value of 0 or z, or a value of 1 or z, respectively.

A.1 Multiinput Combinational Logic Gates

The truth tables of Verilog's combinational logic gates are shown for two inputs, but the gates may be instantiated with an arbitrary number of scalar inputs.

in_1
in_2
out

and	0	1	*x*	*z*
0	0	0	0	0
1	0	1	*x*	*x*
x	0	*x*	*x*	*x*
z	0	*x*	*x*	*x*

FIGURE A-1 Truth table for bitwise-and gate (***and***). Terminal order: *out, in_1, in_2*

nand	0	1	*x*	*z*
0	1	1	1	1
1	1	0	*x*	*x*
x	1	*x*	*x*	*x*
z	1	*x*	*x*	*x*

FIGURE A-2 Truth table for bitwise-nand gate (***nand***). Terminal order: *out, in_1, in_2*

or	0	1	*x*	*z*
0	0	1	*x*	*x*
1	1	1	1	1
x	*x*	1	*x*	*x*
z	*x*	1	*x*	*x*

FIGURE A-3 Truth table for bitwise-or gate (***or***). Terminal order: *out, in_1, in_2*

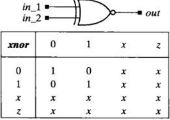
nor	0	1	*x*	*z*
0	1	0	*x*	*x*
1	0	0	0	0
x	*x*	0	*x*	*x*
z	*x*	0	*x*	*x*

FIGURE A-4 Truth table for bitwise-nor gate (***nor***). Terminal order: *out, in_1, in_2*

in_1
in_2
out

xor	0	1	*x*	*z*
0	0	1	*x*	*x*
1	1	0	*x*	*x*
x	*x*	*x*	*x*	*x*
z	*x*	*x*	*x*	*x*

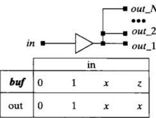
FIGURE A-5 Truth table for bitwise exclusive-or gate (***xor***). Terminal order: *out, in_1, in_2*



xnor	0	1	*x*	*z*
0	1	0	*x*	*x*
1	0	1	*x*	*x*
x	*x*	*x*	*x*	*x*
z	*x*	*x*	*x*	*x*

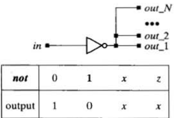
FIGURE A-6 Truth table for bitwise exclusive-Nor gate (***xnor***). Terminal order: *out, in_1, in_2*

A.2 Multioutput Combinational Gates



	in			
buf	0	1	*x*	*z*
out	0	1	*x*	*x*

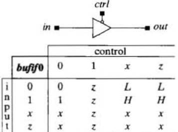
FIGURE A-7 Truth table for bitwise buffer (***buf***). Terminal order: *out_1, out_2, . . . , out_N, in*



not	0	1	*x*	*z*
output	1	0	*x*	*x*

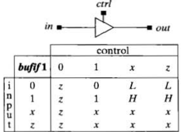
FIGURE A-8 Truth table for bitwise inverter (***not***). Terminal order: *out_1, out_2, . . . , out_N, in*

A.3 Three-State Logic Gates



	bufif0	control 0	1	*x*	*z*
input	0	0	*z*	*L*	*L*
	1	1	*z*	*H*	*H*
	x	*x*	*z*	*x*	*x*
	z	*x*	*z*	*x*	*x*

FIGURE A-9 Truth table for three-state buffer (***bufif0***) with active-low enable. Terminal order: *out, in, ctrl*



	bufif1	control 0	1	*x*	*z*
input	0	*z*	0	*L*	*L*
	1	*z*	1	*H*	*H*
	x	*z*	*x*	*x*	*x*
	z	*z*	*x*	*x*	*x*

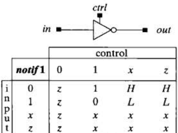
FIGURE A-10 Truth table for three-state buffer (***bufif1***). Terminal order: *out, in, ctrl*

ctrl

in → out

	notif0	control 0	1	x	z
input	0	1	z	H	H
	1	0	z	L	L
	x	x	z	x	x
	z	x	z	x	x

FIGURE A-11 Truth table for three-state inverter (***notif0***) with active-low enable. Terminal order: *out, in, ctrl*



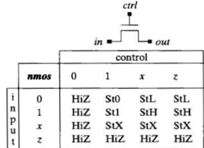
	notif1	control 0	1	x	z
input	0	z	1	H	H
	1	z	0	L	L
	x	z	x	x	x
	z	z	x	x	x

FIGURE A-12 Truth table for three-state inverter (***notif1***). Terminal order: *out, in, ctrl*

A.4 MOS Transistor Switches

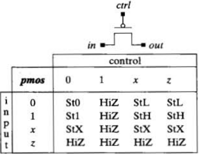
The **cmos, rcmos, nmos, rnmos, pmos,** and **rpmos** gates may be accompanied by a delay specification with one, two, or three values. A single value specifies the rising, falling, and turn-off delay (i.e., to the z state) of the output. A pair of values specifies the rising and falling delays, and the smaller of the two values determines the delay of transitions to x and z. A triple of values specifies the rising, falling, and turn-off delay, and the smallest of the three values determines the transition to x. Delays of transitions to L and H are the same as the delay of a transition to x.[1]

[1] See Ciletti MD. *Modeling Synthesis and Rapid Prototyping with the Verilog HDL* (Prentice-Hall, Upper Saddle River, NJ: 1999) for a discussion of the rules governing the strength of nets driven by switch-level primitives.



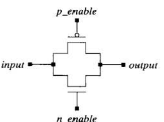
	nmos	control 0	1	x	z
input	0	HiZ	St0	StL	StL
	1	HiZ	St1	StH	StH
	x	HiZ	StX	StX	StX
	z	HiZ	HiZ	HiZ	HiZ

FIGURE A-13 nmos pass transistor switch (***nmos***). Terminal order: *out, in, ctrl*



	pmos	control 0	1	x	z
input	0	St0	HiZ	StL	StL
	1	St1	HiZ	StH	StH
	x	StX	HiZ	StX	StX
	z	HiZ	HiZ	HiZ	HiZ

FIGURE A-14 pmos pass transistor switch (***pmos***). Terminal order: *out, in, ctrl*



cmos	control	input			
n_enable	*p_enable*	0	1	*x*	*z*
0	0	St0	St1	StX	HiZ
0	1	HiZ	HiZ	HiZ	HiZ
0	*x*	StL	StH	StX	HiZ
0	*z*	StL	StH	StX	HiZ
1	0	St0	St1	StX	HiZ
1	1	St0	St1	StX	HiZ
1	*x*	St0	St1	StX	HiZ
1	*z*	St0	St1	StX	HiZ
x	0	St0	St1	StX	HiZ
x	1	StL	StH	StX	HiZ
x	*x*	StL	StH	StX	HiZ
x	*z*	StL	StH	StX	HiZ
z	0	St0	St1	StX	HiZ
z	1	StL	StH	StX	HiZ
z	*x*	StL	StH	StX	HiZ
z	*z*	StL	StH	StX	HiZ

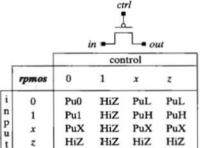
FIGURE A-15 CMOS transmission gate (***cmos***). Terminal order: *output, input, n_enable, p_enable*

ctrl

in out

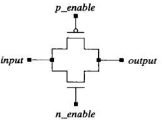
	rnmos	control 0	1	*x*	*z*
input	0	HiZ	Pu0	PuL	PuL
	1	HiZ	Pu1	PuH	PuH
	x	HiZ	PuX	PuX	PuX
	z	HiZ	HiZ	HiZ	HiZ

FIGURE A-16 High-resistance nmos pass transistor switch (***rnmos***). Terminal order: *out, in, ctrl*



	rpmos	control 0	1	*x*	*z*
input	0	Pu0	HiZ	PuL	PuL
	1	Pu1	HiZ	PuH	PuH
	x	PuX	HiZ	PuX	PuX
	z	HiZ	HiZ	HiZ	HiZ

FIGURE A-17 High-resistance pmos pass transistor switch (***rpmos***). Terminal order: *out, in, ctrl*



rcmos	control	input			
n_enable	p_enable	0	1	x	z
0	0	Pu0	Pu1	PuX	HiZ
0	1	HiZ	HiZ	HiZ	HiZ
0	x	PuL	PuH	PuX	HiZ
0	z	PuL	PuH	PuX	HiZ
1	0	Pu0	Pu1	PuX	HiZ
1	1	Pu0	Pu1	PuX	HiZ
1	x	Pu0	Pu1	PuX	HiZ
1	z	Pu0	Pu1	PuX	HiZ
x	0	Pu0	Pu1	PuX	HiZ
x	1	PuL	PuH	PuX	HiZ
x	x	PuL	PuH	PuX	HiZ
x	z	PuL	PuH	PuX	HiZ
z	0	Pu0	Pu1	PuX	HiZ
z	1	PuL	PuH	PuX	HiZ
z	x	PuL	PuH	PuX	HiZ
z	z	PuL	PuH	PuX	HiZ

FIGURE A-18 High-resistance cmos transmission gate (***rcmos***). Terminal order: *output, input, n_enable, p_enable*

A.5 MOS Pull-Up/Pull-Down Gates

The pull-up (***pullup***) and pull-down (***pulldown***) gates place a constant value of 1 or 0 with strength *pull*, respectively, on their output. This value is fixed for the duration of simulation, so no delay values may be specified for these gates. The default strength of these gates is *pull*. *Note*: The ***pulldown*** and ***pullup*** gates are not to be confused with ***tri0*** and ***tri1*** nets. The latter are nets that provide connectivity, and may have multiple drivers; the former are functional elements in the design. The ***tri0*** and ***tri1*** nets may have multiple drivers. The net driven by a ***pullup*** or ***pulldown*** gate may also have multiple drivers. Verilog's ***pullup*** and ***pulldown*** primitives can be used to model pull-up and pull-down devices in electrostatic-discharge circuitry tied to unused inputs on flip-flops.

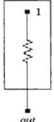


FIGURE A-19 Pull-up device. Terminal order: *out*

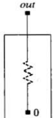


FIGURE A-20 Pull-down device. Terminal order: *out*

A.6 MOS Bidirectional Switches

Verilog includes six predefined bidirectional switch primitives: ***tran, rtran, tranif0, rtranif0, tranif1***, and ***rtranif1***. Bidirectional switches provide a layer of buffering on bidirectional signal paths between circuits. A signal passing through a bidirectional switch is not delayed (i.e., output transitions follow input transitions without delay).

Note: The ***tran*** and ***rtran*** primitives model bidirectional pass gates, and may not have a delay specification. These bidirectional switches pass signals without delay. The ***rtranif0, rtranif1, tranif1***, and ***rtranif1*** switches are accompanied by a delay specification,

which specifies the turn-on and turn-off delays of the switch; the signal passing through the switch has no delay. A single value specifies both delays, a pair of values (turn-on, turn-off) specifies both delays, with the turn-on being the first item and turn-off being the second item. The default delay is 0.

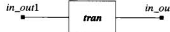


FIGURE A-21 Bidirectional switch (***tran***). Terminal order: *in_out1, in_out2*

FIGURE A-22 Resistive bidirectional switch (***rtran***). Terminal order: *in_out1, in_out2*

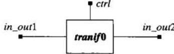


FIGURE A-23 Three-state bidirectional switch (***tranif0***). Terminal order: *in_out1, in_out2, ctrl*

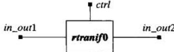


FIGURE A-24 Resistive three-state bidirectional switch (***rtranif0***). Terminal order: *in_out1, in_out2, ctrl*

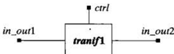


FIGURE A-25 Three-state bidirectional switch (***tranif1***). Terminal order: *in_out1, in_out2, ctrl*

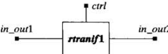


FIGURE A-26 Resistive three-state bidirectional switch (***rtranif1***). Terminal order: *in_out1, in_out2, ctrl*

APPENDIX B

Verilog Keywords

Verilog keywords in Verilog 1364 (1995, 2001, and 2005) are predefined, lower-case, nonescaped identifiers that define the language constructs. An identifier may not be a keyword, and an escaped identifier is not treated as a keyword. In this text, Verilog keywords are printed in boldface.

always
and
assign
automatic
begin
buf
bufif0
bufif1
case
casex
casez
cell
cmos
config
deassign
default
defparam
design
disable
edge
else
end
endcase
endconfig
endfunction
endgenerate
endmodule
endprimitive
endspecify
endtable
endtask
event
for
force
forever
fork
function
generate
genvar
highz0
highz1
if
ifnone
incdir
include
initial
inout
input
instance
integer
join
large
liblist
library
localparam
macromodule
medium
module
nand
negedge
nmos
nor
noshowcancelled
not
notif0
notif1
or
output
parameter
pmos
posedge
primitive
pull0
pull1
pulldown
pullup
pulsestyle_onevent
pulsestyle_ondetect
rcmos
real
realtime
reg
release
repeat
rnmos
rpmos
rtran
rtranif0
rtranif1
scalared
showcancelled
signed
small
specify
specparam
strong0

PDG

strong1	tranif0	trireg	weak0
supply0	tranif1	unsigned	weak1
supply1	tri	use	while
table	tri0	uwire	wire
task	tri1	vectored	wor
time	triand	wait	xnor
tran	trior	wand	xor

APPENDIX C

Verilog Data Types

Verilog has two families of fixed data types: nets and registers. Nets establish structural connectivity. Registers store information.

C.1 Nets

The family of net data types is described in Table C-1.

TABLE C-1 Data types in Verilog

Net Types	
wire	Establishes connectivity, with no logical behavior or functionality implied.
tri	Establishes connectivity, with no logical behavior or functionality implied. This type of net has the same functionality as **wire**, but is identified distinctively to indicate that it will be three-stated in hardware.
wand	A net that is connected to the output of multiple primitives. It models the hardware implementation of a wired-AND, e.g., open collector technology.
wor	A net that is connected to the output of multiple primitives. It models the hardware implementation of a wired-OR, e.g., emitter coupled logic.
triand	A net that is connected to the output of multiple primitives. It models the hardware implementation of a wired-AND, e.g., open collector technology. The physical net is to be three-stated.
trior	A net that is connected to the output of multiple primitives. It models the hardware implementation of a wired-OR, e.g., emitter coupled logic. The physical net is to be three-stated.
supply0	A global net that is connected to the circuit ground.
supply1	A global net that is connected to the power supply.
tri0	A net that is connected to ground by a resistive pull-down connection.
tri1	A net that is connected to the power supply by a resistive pull-up connection.
trireg	A net that models the charge stored on a physical net.

At time $t_{sim} = 0$, nets that are driven by a primitive, module, or continuous assignment have a value determined by their drivers, which defaults to the (ambiguous) logic value, x. The simulator assigns the default logic value z (high impedance) to all nets that are not driven. These initial assigned values remain until they are changed by subsequent events during simulation.

C.2 Register Variables

Register variables are assigned values by procedural statements within an ***always*** or ***initial*** block. Register variables hold their value until an assignment statement changes them. The following are predefined register types: ***reg, integer, real, realtime***, and ***time***.

C.2.1 Data Type: reg

The data type **reg** is an abstraction of a hardware storage element, but it does not correspond directly to physical memory. A ***reg*** variable has a default initial value of x. The default size of a register variable is a single bit. Verilog operators create a **reg** variable as an unsigned value. A register variable may be assigned value only by a procedural statement, a user-defined sequential primitive, a task, or a function. A **reg** variable may never be the output of a predefined primitive gate, an **input** or **inout** port of a module, or the target of a continuous assignment.

C.2.2 Data Type: integer

The data type ***integer*** supports numeric computation in procedural statements. Integers are represented internally to the word length of the host machine (at least 32 bits). A negative integer is stored in 2s complement format. A ***integer*** variable has a default initial value of 0.

Verilog operators operate on integers with 2s complement arithmetic, with the (most significant bit) indicating the sign of the value. For example, the negative integer -4_{10} is stored as 1111_1111_1111_1111_1111_1111_1111_1100. When the size of a number assigned to an integer is less than the length of the word used by the machine to store an integer, the number is padded with 0s to the left. The number assigned to an integer variable must have a decimal equivalent (i.e., x and z are not allowed). Because integers have a fixed word size, they may not be declared to have a range specification. Some examples of valid declarations of integers and arrays of integers are shown below:

Example

```
integer A1, K, Size_of_Memory;
integer Array_of_Ints [1:100];
```

End of Example

An integer will be interpreted as a signed value in 2s complement form if it is assigned a value without a base specifier (e.g., $A = -24$). If the value assigned has a specified base, the integer is interpreted as an unsigned value. For example, if A is an integer, the result of $A = -12/3$ is -4; the result of $A = -$'d12/3 is 1431655761. Both words evaluate to the same bit pattern, but the former is interpreted as a negative value in 2s complement.

C.2.3 Data Type: real

Variables having type ***real*** are stored in double precision, typically a 64-bit value. A ***real*** variable has a default initial value of 0.0. Real variables can be specified in decimal and exponential notation. An object of type ***real*** may not be connected to a port of a module or to a terminal of a primitive. Verilog includes two system tasks that convert data types to permit real data transfer across a port boundary in a hierarchical structure: ***$realtobits*** and ***$bitstoreal***. The language reference manual (LRM) describes limitations on the use of operators with real operands.

C.2.4 Data Type: realtime

Variables having type ***realtime*** are stored in real number format. A ***realtime*** variable has a default initial value of 0.0.

C.2.5 Data Type: time

The data type ***time*** supports time-related computations within procedural code in Verilog models. ***time*** variables are stored as unsigned 64-bit quantities. A variable of type ***time*** may not be used in a module port, nor may it be an input or output of a primitive. A ***time*** variable has a default value of 0.

C.2.6 Common Error: Undeclared Register Variables

Verilog has no mechanism for handling undeclared register variables. An identifier that has not been declared is assumed to reference a net of the default type (e.g., ***wire***). A procedural assignment to an undeclared variable will cause a compiler error.

C.2.7 Addressing Nets and Register Variables

The most significant bit of a part-select of a net or register is always the leftmost array index; the least significant bit is the rightmost array index. A constant or variable expression can be the index of a part-select. If the index of a part-select is out of bounds the value x is returned by a reference to the variable.

Example

If an 8-bit word *vect_word* has a stored value of decimal 4, then *vect_word[2]* has a value of 1; *vect_word[3:0]* has a value of 4; *vect_word[5:1]* has value 2; i.e., *vect_word[7:0]* = 0000_0100_2, and *vect_word[5:1]* = 0_0010_2.

End of Example

C.2.8 Common Error: Passing Variables through Ports

Table C-2 summarizes the rules that apply to nets and registers that are port objects in a Verilog module. For example, a register variable may not be declared to be an **inout** port.

TABLE C-2 Rules for port modes with nets and registers.

	Port Mode		
Variable Type	input	output	inout
Net Variable	Yes	Yes	Yes
Register Variable	No	Yes	No

A variable that is declared as an ***input*** port of a module is implicitly a net variable within the scope of the module, but a variable that is declared to be an ***output*** port may be a net or a register variable. A variable declared to be an ***input*** port of a module may not be declared to be a register variable. An ***inout*** port of a module may not be a register type. A register variable may not be placed in an output port of a primitive gate, and may not be the target (LHS) of a continuous assignment statement.

C.2.9 Arrays

A one-dimension array with elements of type **reg** is called a memory, and represents an array of words. This construct is an extension of the declaration of a register variable to provide a *memory*, i.e., multiple addressable cells of the same word size. An example of the syntax for a memory of register variables is shown below. Bit-select and part-select are not valid with a memory. Reference may be made to only a *word* of a memory. The MSB of a part-select is the leftmost array index; the LSB is the rightmost. If an index is out of bounds the result is the logic value ***x***. A *constant* expression may be used for the LSB and the MSB in a declaration of an array.

Example

The code fragment below shows how the syntax for declaring a ***reg*** memory variable simplifies to the form: ***reg*** *word_size array_name memory_size* for an array of 1024 32-bit words:

```
word size              memory size
     |    memory name        |
     ↓         ↓             ↓
reg [31:0] cache_memory [0:1023];
```

End of Example

Multi-dimensional arrays of elements are formed by appending one or more address ranges to a declaration providing the type, size, and name of an object.

Example

```
reg [15: 0] data   [0: 127][0: 127];
```

End of Example

Verilog 1995 allows a part-select of contiguous bits from a vector if the range indices of the part select are constant. Verilog 2001, 2005 provide two additional part-select operators to provide a indexed variable part-select of fixed width, +*:* and −*:*, having the syntax [<*start_bit*> +: <*width*>] and [<*start_bit*> −: <*width*>], respectively. The parameter *width* specifies the size of the part select, and *start_bit* specifies the rightmost or the leftmost bit in the vector from which the part-select is taken, depending on whether the selection will be made by incrementing (+) or decrementing (−) the index of the bits of the vector.

C.2.10 Scope of a Variable

The scope of a variable is the module, task, function, or named procedural (***begin . . . end***) block in which it is declared. In Figure C-1 a net at the input port of *child_module* can be driven by a net or register in the enclosing *parent_module*, and a net or a register at the output port of *child_module* can drive a net in *parent_module*.

C.2.11 Strings

Verilog does not have a distinct data type for strings. Instead, a string must be stored within a properly sized register by a procedural assignment statement. A properly sized **reg** (array) has 8 bits of storage for each character of the string that it is to hold.

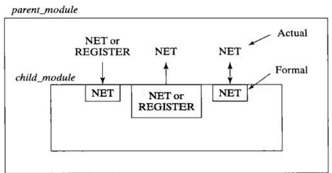


FIGURE C-1 Scope of nets and registers.

Example

A declaration of a **reg**, *string_holder*, that will accommodate a string with *num_char* characters is given below.

```
reg [8*num_char-1: 0] string_holder;
```

End of Example

The declaration in the example implies that 8 bits will encode each of the *num_char* characters. If the string "Hello World" is assigned to *string_holder*, it is necessary that *num_char* be at least 11 to ensure that a minimum of 88 bits are reserved. If an assignment to an array consists of less characters than the array will accommodate, 0s are automatically filled in the unused positions, beginning at the position of the MSB (i.e., the leftmost position).

C.3 Constants

A constant in Verilog is declared with the keyword **parameter**, which declares and assigns value to the constant. The value of a constant may not be changed during simulation. Constant expressions may be used in the declaration of the value of a constant.

Example

```
parameter high_index = 31;                                  // integer
parameter width = 32, depth = 1024;                         // integers
parameter byte_size = 8, byte_max = byte_size-1;            // integer
parameter a_real_value = 6.22;                              // real
parameter av_delay = (min_delay + max_delay)/2;             // real
parameter initial_state = 8'b1001_0110;                     // reg
```

End of Example

C.4 Referencing Arrays of Nets or Regs

A net or a register is referenced by its identifier. A reference to a vector object can include a bit-select (i.e., a single bit, or element) or part-select consisting of a range of contiguous bits enclosed by square brackets (e.g., [7:0]). An expression can be the index of a part-select. If a declared vector identifier has an ascending (descending) order from its LSB to its MSB, a referenced part-select of that identifier must have the same ascending (descending) order from its LSB to its MSB.

APPENDIX D

Verilog Operators

The built-in operators of the Verilog HDL manipulate the various types of data implemented in the language to produce values on nets and registers. Some of the operators are used within expressions on the righthand side of continuous-assignment statements and procedural statements; others are used in Boolean expressions in conditional statements or with conditional operators. Verilog 2001, 2005 implements the classes of operators listed in Table D-1. The meaning of Verilog's operators is fixed; there is no provision for "overloading" an operator. The interpretation of the operators and operands is automatic and transparent to the user. The arithmetic supporting these operators is fully implemented in Verilog for scalar and vector **nets** and **regs**, for 2 complement arithmetic modulo 2^n, where n is the length of the operand word.

D.1 Arithmetic Operators

Arithmetic operators create a numeric value by operating on a pair of operands representing numeric values expressed in binary, decimal, octal, or hexadecimal form. The arithmetic operators supported by Verilog are identified in Table D-2.

When arithmetic operations are performed on vectors (net and registers) the result is determined by modulo 2^n arithmetic, where n is the size of the vector. *The bit*

TABLE D-1 Arguments and results produced by Verilog operators.

Operator	Argument	Result
Arithmetic	Pair of operands	Binary word
Bitwise	Pair of operands	Binary word
Reduction	Single operand	Bit
Logical	Pair of operands	Boolean value
Relational	Pair of operands	Boolean value
Shift	Single operand	Binary word
Conditional	Three operands	Expression

TABLE D-2 Verilog arithmetic operators and symbols.

Symbol	Operator
+	Addition
−	Subtraction
*	Multiplication
/	Division
**	Exponentiation
%	Modulus

pattern that is stored in a register is interpreted as an unsigned value. A negative value is stored in 2s complement format, but it is interpreted as an unsigned quantity when it is used in an expression. For example, if -1 is stored in a 2-bit register, its value, after its 2s complement is formed, is $11_2 = 3_d$. On the other hand, if -1 is stored in a 3-bit register its value $111_2 = 7_d$.

Example

The simulation results shown below illustrate the modulo 2^n arithmetic operations of addition, subtraction, and negation.

```
module arith1 ();
  reg    [3:0]    A, B;
  wire   [4:0]    sum, diff1, diff2, neg;
  assign sum = A + B;
  assign diff1 = A – B;
  assign diff2 = B – A;
  assign neg = –A;

  initial
   begin
    #5 A = 5; B = 2;
    $display ("          t_sim A  B  A+B  A-B  B-A  -A");
    $monitor ($time,"%d   %d  %d   %d    %d     %d", A, B, sum, diff1, diff2, neg);
    #10 $monitoroff;
    $monitor ($time,,"%b %b %b %b %b %b", A, B, sum, diff1, diff2, neg);
  end
endmodule
```

t_sim	A	B	A + B	A−B	B−A	−A
5	5	2	7	3	29	27
15	0101	0010	00111	00011	11101	11011

Note that $A + B$ and $A - B$ have values that are expected. But $B - A$ does not return the decimal equivalent of $2 - 5 = -3$. The actual value of $B - A$ is obtained by adding the 2s complement of A (11011_2) to the value of B (00010_2), using the word length of the result. Verilog descriptions involving operations on nets and registers must anticipate modulo 2^n arithmetic, or implement an arithmetic scheme, such as

signed magnitude arithmetic. Note that the value of $B - A$ is the 2s complement of the correct result, but without the correct sign. The presence of 1 in the MSB indicates that the result is to be interpreted as a negative value expressed in 2s complement format.

End of Example

D.2 Bitwise Operators

The bitwise negation operator negates the individual bits of a word. The other bitwise operators produce a binary word result by operating bitwise on a pair of operands. The operands may be scalar or vector. Table D-3 lists the Verilog bitwise operators.

TABLE D-3 Verilog bitwise operators.

<table>
<tr><th>Symbol</th><th>Operator</th></tr>
<tr><td>~</td><td>Bitwise negation</td></tr>
<tr><td>&</td><td>Bitwise and</td></tr>
<tr><td>|</td><td>Bitwise inclusive or</td></tr>
<tr><td>^</td><td>Bitwise exclusive or</td></tr>
<tr><td>~^, ^~</td><td>Bitwise exclusive nor</td></tr>
</table>

Example

If $y1$ is the binary word 1011_0001 and $y2$ is the binary word 0010_1001, then the bitwise operation $y1$ & $y2$ produces the result: 0010_0001. A 1 is produced in a given position of the result if a 1 is present at that position in both of the operands.

End of Example

D.3 Reduction Operators

Reduction operators are unary operators. They create a single-bit value by operating on a single data word. Table D-4 lists the Verilog reduction operators.

TABLE D-4 Verilog reduction operators.

<table>
<tr><th>Symbol</th><th>Operator</th></tr>
<tr><td>&</td><td>Reduction and</td></tr>
<tr><td>~&</td><td>Reduction nand</td></tr>
<tr><td>|</td><td>Reduction or</td></tr>
<tr><td>~|</td><td>Reduction nor</td></tr>
<tr><td>^</td><td>Reduction exclusive or</td></tr>
<tr><td>~^, ^~</td><td>Reduction xnor</td></tr>
</table>

PDG

Example

If y is the eight-bit binary word 1011_0001, the "reduction and" operation on y produces: $\&y = 0$. The reduction and operation takes the "and" of the bits in the operand. It returns a 1 if all of bits in the operand are 1.

End of Example

Example

Examples of the reduction operator are given below.

<table>
<tr><th>Expression</th><th>Result</th><th>Operator</th></tr>
<tr><td>&(010101)</td><td>0</td><td>Reduction And</td></tr>
<tr><td>|(010101)</td><td>1</td><td>Reduction Or</td></tr>
<tr><td>&(010 × 10)</td><td>0</td><td>Reduction And</td></tr>
<tr><td>|(010 × 10)</td><td>1</td><td>Reduction Or</td></tr>
</table>

End of Example

D.4 Logical Operators

The Verilog logical operators operate on Boolean operands as logical connectives to create a Boolean result. The operand may be a net, a register, or an expression that is evaluated to produce a Boolean result. This family of operators is listed in Table D-5. Logical operators are commonly used with the conditional assignment operator and in conditional (*if*) statements within behaviors, functions, or tasks.

TABLE D-5 Verilog logical operators.

<table>
<tr><th>Symbol</th><th>Operator</th></tr>
<tr><td>!</td><td>Logical negation</td></tr>
<tr><td>&&</td><td>Logical and</td></tr>
<tr><td>||</td><td>Logical or</td></tr>
<tr><td>==</td><td>Logical equality</td></tr>
<tr><td>!=</td><td>Logical inequality</td></tr>
<tr><td>===</td><td>Case equality</td></tr>
<tr><td>!==</td><td>Case inequality</td></tr>
</table>

Example

Examples using logical operators are given below:

a. **if** ((a < size − 1) && (b != c) && (index != last–one) ...
b. **if** (! inword) ...
c. **if** (inword == 0) ...

End of Example

The ***case*** equality operators (e.g., ===) are used to determine whether two words match identically on a bit-by-bit basis, including bits that have values x or z. The logical equality operator (==) is less restrictive—it is used to test whether two words are identical, but it produces an x result when the test is ambiguous. The comparison is made bit by bit, and 0s are filled in as necessary. The result of the test is 1 if the comparison is true, and 0 if the comparison is false. If the operands are nets or registers, their values are treated as unsigned words. If any bit is unknown, the relation is unknown, and the result that is returned is ambiguous (x value). If the operands are integers or reals, they may be signed values, but they are compared as though they are unsigned.

The appropriate use of the logical OR and the logical AND operators is as connectives in a logical expression. Verilog is loosely typed, so the logical operators can be used inappropriately. For example A && B will return a Boolean scalar result. If A and B are scalars, the result will be the same as obtained using A & B. If A and B are vectors, A && B returns Boolean true if both words are positive integers. A & B returns true if the word formed from the bitwise operation is a positive integer. For example, suppose $A = 3'b110$ and $B = 3'b11x$. Then A && $B = 0$, which has a Boolean value of false, because B is false. On the other hand, A & $B = 110$, which has a Boolean value of true.

D.5 Relational Operators

The Verilog relational operators compare operands and produce a Boolean (true or false) result. If the operands are nets or registers, their values are treated as unsigned words. If any bit is unknown, the relation is unknown, and the result that is returned is ambiguous (x value). If the operands are integers or reals, they may be signed. Table D-6 lists the Verilog relational operators.

TABLE D-6 Verilog relational operators.

Symbol	Operator
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

D.6 Shift Operators

Verilog *logical shift* operators (Verilog 1364-1995, 2001, 2005) shift the bits of a single operand left or right by a specified number of position, filling in with zeros the positions that are vacated by the shift. Verilog *arithmetic shift* operators (Verilog 1364-2001, 2005) shift the bits of a single operand left or right by a specified number of positions. If the shift is to the right, the vacated bits are replaced by the MSB of the word; if the shift is to the left, the vacated bits are replaced by 0.

Note: the effect of the left arithmetic shift is identical to that of the left logical shift. Table D.7 identifies the shift operators.

TABLE D-7 Verilog shift operators.

Symbol	Operator
<<	Left logical shift
>>	Right logical shift
<<<	Left arithmetic shift
>>>	Right arithmetic shift

Example

If word A has the bit pattern 1011_0011, the statements A = A << 1 and A <<< 1 create the bit pattern A = 0110_0110. The statement A = A >> 2 creates the bit pattern A = 0010_1100. The statement A = A >>> 3 creates the pattern A = 1111_0110.

End of Example

D.7 Conditional Operator

The Verilog conditional assignment operator selects an expression for evaluation, based on the value of *conditional_expression*. The conditional assignment operator has the syntax: *conditional_expression ? true_expression : false_expression*. If *conditional_expression* evaluates to Boolean true, then *true_expression* is evaluated; otherwise, *false_expression* is evaluated.

Example

The statement below assigns the value of A to Y if $A = B$; otherwise, it assigns the value of B.

$$Y = (A == B)\ ?\ A : B;$$

End of Example

Example

The following Verilog statement uses the conditional operator to assign value to *bus_a*.

```
wire [15:0] bus_a = drive_bus_a ? data : 16'bz;
```

The effect of the assignment is summarized below:

drive_bus_a = 1 sets data on *bus_a*
drive_bus_a = 0 sets *bus_a* to high impedance
drive_bus_a = x sets *x* on *bus_a*

End of Example

The conditional operator can be nested to any depth.

The conditional operator can also be used to control the activity flow of the procedural statements within a Verilog behavior. The following rules determine the value that results from an assignment using the conditional operator: (1) logic value *z* is not allowed in the *conditional_expression*, (2) 0s are automatically filled if the operands have different lengths, (3) if the *conditional_expression* is ambiguous, then both *true_expression* and *false_expression* are evaluated and the result is calculated on a bitwise basis according to Table D-8.

Note that the truth table assigns 0 (1) to the expression when both *true_expression* and *false_expression* have the value 0 (1). In these cases the result of the evaluation does not depend on *conditional_expression*. One restriction that applies to the use of the concatenation operator is that no operand may be an unsized constant.

TABLE D-8 Truth table for the conditional assignment operator.

		true_expression		
	?:	**0**	**1**	***x***
false_expression	**0**	0	*x*	*x*
	1	*x*	1	*x*
	x	*x*	*x*	*x*

D.8 Concatenation Operator

The concatenation operator forms a single word from two or more operands. This operator is particularly useful in forming logical busses. The concatenation result follows the same order in which the words are given. The concatenation operator nests to any depth and repetition.

Example

a. If the operand A is the bit pattern 1011 and the operand B is the bit pattern 0001, then $\{A, B\}$ is the bit pattern 1011_0001.
b. $\{4\{a\}\} = \{a, a, a, a\}$
c. $\{0011, \{\{01\}, \{10\}\}\} = 0011_0110$.

End of Example

D.9 Expressions and Operands

Verilog expressions combine operands with the language's operators to produce a resultant value. A Verilog operand may be composed of nets, registers, constants, numbers, bit-select of a net, bit-select of a register, part-select of a net, part-select of a register, memory element, a function call, or a concatenation of any of these. The result of an expression may be used to determine an assignment to a net or register variable or to choose among alternatives. The value of an expression is determined by performing the indicated operations on its operands. An expression may consist of a single identifier (operand), or some combination of operands and operators that conforms to the allowed syntax of the language. The evaluation of an expression always produces a value that is represented by one or more bits.

Example

Some examples of expressions are given below:
a. **assign** THIS_SIG = A_SIG ^ B_SIG;
b. **assign** y_out = (select) **?** input_a : input_b;

End of Example

D.10 Operator Precedence

Verilog evaluates expressions from left to right, and the evaluation of a Boolean expression is terminated as soon as the expression is determined to be true or false. The precedence of Verilog operators within an expression is given in Table D-9. The operators in the same row have the same precedence, and the rows are ordered from top to bottom.

The result produced by a compiler may not correspond to the intent expressed in an expression. As a precaution, use parentheses to eliminate ambiguity in expressions.

TABLE D-9 Verilog operators and their precedence.

<table>
<tr><th>Operator Symbol</th><th>Function</th><th>Precedence</th></tr>
<tr><td>+ − ! ~ (unary)</td><td>Sign, complement</td><td>Highest</td></tr>
<tr><td>**</td><td>Exponentiation</td><td></td></tr>
<tr><td>* / %</td><td>Multiplication, Division, Modulus</td><td></td></tr>
<tr><td>+ − (binary)</td><td>Addition, Subtraction</td><td></td></tr>
<tr><td><< >> <<< >>></td><td>Shift</td><td></td></tr>
<tr><td>< <= > >=
== != === !==</td><td>Relational</td><td></td></tr>
<tr><td>& ~&
^ ^~ ~^
| ~|</td><td>Reduction</td><td></td></tr>
<tr><td>&&
||</td><td>Logical</td><td></td></tr>
<tr><td>? :</td><td>Conditional</td><td>Lowest</td></tr>
</table>

D.11 Arithmetic with Signed Data Types

Verilog-1995 is limited to signed arithmetic on 32-bit integers. The ***reg*** and net data types are unsigned, and expressions are evaluated as signed arithmetic only if every operand is a signed variable (i.e. has type ***integer***). The data types of the variables in an expression, not the operators, determine whether signed or unsigned arithmetic is performed. Verilog-2001, 2005 use the reserved keyword ***signed*** to declared that a ***reg*** or a net type variable is signed, and supports signed arithmetic on vectors of any size, not just 32-bit values.

Example: Arithmetic with Signed Variables

Figure D-1 declares signed variables in Verilog 2001 and illustrates the results stored from arithmetic operations in Verilog 1995, 2001, and 2005.

Verilog 1995	Verilog 2001, 2005
integer m, n; **reg** [63: 0] v;	**integer** m, n; **reg signed** [63: 0] v;

... // value stored
m = 12; // 0000_..._0000_1100
n = −4; // 1111_..._1111_1100
v = 8; // 0000_..._0000_1000
	... // value stored
	m = 12; // 0000_..._0000_1100
	n = −4; // 1111_..._1111_1100
	v = 8; // 0000_..._0000_1000

m = m / n; // result: −3	m = m / n; // result: −3
v = v / n; // result: 0	v = v / n; // result: −2

FIGURE D-1 Arithmetic with signed data types.

Example End: Arithmetic with Signed Variables

D.12 Signed Literal Integers

Verilog-1995 represents literal integers in three ways: a number (e.g., −10), an unsized radix-specified number (e.g., 'hA), and a sized radix-specified number (e.g., 64'hF). If a radix is specified the number is interpreted as an unsigned value; if a radix is omitted, the number is interpreted as a signed value. In Verilog 2001, a sized literal integer can be declared as signed. The symbol ***s*** is pre-appended to the base specifier to specify that a sized or unsized literal integer is signed.

Example: Declaration of Signed Variables and Literals

The statements below illustrate the declaration of a signed variables and the results of arithmetic with signed variables and literals.

```
reg      signed    [63: 0]   v;          // Signed variable
...
v = 12;                                  // Literal integer
...
v = v / -64'd2;                          // Stored as 0
v = v / -64'sd2;                         // Stored as -6
```

Example End: Declaration of Signed Variables and Literals

D.13 System Functions for Sign Conversion

Verilog 2001 provides two new system functions for converting values to signed or unsigned values. The function **$signed** returns a signed value from the value passed in. The function **$unsigned** returns an unsigned value from the value pass in. The functions are useful because an expression returns a signed value if and only if all of its operands are signed variables. Sign conversion eliminates the need to declare and assigned value to additional variables to circumvent the restrictions of Verilog-1995.

Example: Arithmetic with Sign Conversion Functions

In the statements below the function ***$signed*** returns a signed value from its argument, *sum_diff*, with the result apparent in the value stored for *signed_sum_diff*.

```
integer             v;
reg      [63: 0]    sum_diff;
...
v = -16;
sum_diff = 48;
sum_diff = sum_diff/v;                              // Returns 0
signed_sum_diff = $signed (sum_diff)/v;             // Returns -3
```

Example End: Arithmetic with Sign Conversion Functions

2.1.1 Assignment Width Extension

Verilog-1995 has two rules for extending the bits of a word when the expression on the RHS side of an assignment statement has a smaller width than the expression on the LHS. If the expression on the RHS is *signed*, the sign-bit determines the extension to fill the LHS. If the expression on the RHS is *unsigned* (i.e. ***reg***, ***time***, and all net types), its extension is formed by filling with 0. This can lead to inappropriate extensions when the LHS exceeds 32 bits.

Verilog-2001 has a more elaborate set of rules for extending the width of a word beyond 32-bits, as summarized in Figure D-2. These rules differ from those for Verilog-1995, so a model that adhered to the rules of Verilog-1995 will not work he same as a model employing the rules of Verilog-2001.

	Extended value	
Left-most bit of RHS expression	Unsigned RHS expression	Signed RHS expression
0	0	0
1	0	1
x	*x*	*x*
z	*z*	*z*

FIGURE D-2 Width extension in Verilog 2001, 2005.

APPENDIX E

Verilog Language Formal Syntax

The syntax of the Verilog language conforms to the following Backus-Naur Form (BNF) of formal syntax notation.

1. White space may be used to separate lexical tokens.
2. *Name* ::= starts off the definition of a syntax construction item. Sometimes *Name* contains embedded underscores (_). Also, the symbol::= may be found on the next line.
3. The vertical bar, |, introduces an alternative syntax definition, unless it appears in bold.
4. ***Name*** in bold text is used to denote reserved keywords, operators, and punctuation marks required in the syntax.
5. [item] is an optional item that may appear once or not at all.
6. {item} is an optional item that may appear once, more than once, or not at all. If the braces are in bold, then they are part of the syntax.
7. *Name1_name2* is equivalent to the syntax construct item *name2*. The *name1* (in italics) imparts some extra semantic information to name2. However, the item is defined by the definition of *name2*.
8. The notation | ... is used in the non-appendix text to indicate that there are other alternatives, but that due to space or expediency they are not listed here.

PDG

APPENDIX F

Verilog Language Formal Syntax

This formal syntax specification is provided in Backus-Naur form (BNF). It is reprinted from IEEE 1364-2001, 2005. The IEEE disclaims any responsibility or liability resulting from the placement an use in this publication. This information is reprinted with permission of the IEEE.

The syntax of the Verilog language conforms to the following BNF of formal syntax notation.

F.1 Source text

F.1.1 Library source text

library_text ::= { library_descriptions }
library_descriptions ::=
    library_declaration
    | include_statement
    | config_declaration
library_declaration ::=
    library library_identifier file_path_spec [{ , file_path_spec }]
    [**-incdir** file_path_spec [{ , file_path_spec }] ;
file_path_spec ::= file_path
include_statement ::= **include** < file_path_spec > ;

F.1.2 Configuration source text

config_declaration ::=
    config config_identifier ;
    design_statement

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

{ config_rule_statement }
endconfig
design_statement ::= **design** { [library_identifier.] cell_identifier } ;
config_rule_statement ::=
default_clause liblist_clause
| inst_clause liblist_clause
| inst_clause use_clause
| cell_clause liblist_clause
| cell_clause use_clause
default_clause ::= **default**
inst_clause ::= **instance** inst_name
inst_name ::= topmodule_identifier{ .instance_identifier }
cell_clause ::= **cell** [library_identifier.] cell_identifier
liblist_clause ::= **liblist** [{ library_identifier }]
use_clause ::= **use** [library_identifier .] cell_identifier [**: config**]

F.1.3 Module and primitive source text

source_text ::= { description }
description ::=
module declaration
| udp_declaration
module_declaration ::=
{ attribute_instance } module_keyword module_identifier
[module_parameter_port_list]
[list_of_ports] ; { module_item }
endmodule
| { attribute_instance } module_keyword module_identifier
[module_parameter_port_list]
[list_of_port_declarations] ; { non_port_module_item }
endmodule
module_keyword ::= **module** | **macromodule**

F.1.4 Module parameters and ports

module_parameter_port_list ::= # (parameter_declaration { , parameter_declaration })
list_of_ports ::= (port { , port })
list_of_port_declarations ::=
(port_declaration { , port_declaration})
| ()
port ::=
[port_expression]
| . port_identifier ([port_expression])
port_expression ::=
port_reference
| { port_reference { , port_reference } }

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

```
port_reference ::=
      port_identifier
    | port_identifier [ constant_expression ]
    | port_identifier [ range_expression]
port_declaration ::=
      {attribute_instance } inout_declaration
    | {attribute_instance } input_declaration
    | {attribute_instance } output_declaration
```

F.1.5 Module items

```
module_item ::=
      module_or_generate_item
    | port_declaration;
    | { attribute_instance } generated_instantiation
    | { attribute_instance } local_parameter_declaration
    | { attribute_instance } parameter_declaration
    | { attribute_instance } specify_block
    | { attribute_instance } specparam_declaration
module_or_generate_item ::=
      { attribute_instance } module_or_generate_item_declaration
    | { attribute_instance } parameter_override
    | { attribute_instance } continuous_assign
    | { attribute_instance } gate_instantiation
    | { attribute_instance } udp_instantiation
    | { attribute_instance } module_instantiation
    | { attribute_instance } initial_construct
    | { attribute_instance } always_construct
module_or_generate_item_declaration ::=
      net_declaration
    | reg_declaration
    | integer_declaration
    | real_declaration
    | time_declaration
    | realtime_declaration
    | event_declaration
    | genvar_declaration
    | task_declaration
    | function_declaration
non_port_module_item ::=
      { attribute_instance } generated_instantiation
    | { attribute_instance } local_parameter_instantiation
    | { attribute_instance } module_or_generate_item
    | { attribute_instance } parameter_declaration
    | { attribute_instance } specify_block
    | { attribute_instance } specparam_declaration
parameter_override ::= defparam list_of_param_assignments ;
```

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

F.2 Declarations

F.2.1 Declaration types

F.2.1.1 Module parameter declarations

local_parameter_declaration ::=
  localparam [**signed**] [range] list_of_param_assignments ;
  | **localparam integer** list_of_param_assignments ;
  | **localparam real** list_of_param_assignments ;
  | **localparam realtime** list_of_param_assignments ;
  | **localparam time** list_of_param_assignments ;
parameter_declaration ::=
  parameter [**signed**] [rang e] list_of_param_assignments ;
  | **parameter integer** list_of_param_assignments ;
  | **parameter real** list_of_param_assignments ;
  | **parameter realtime** list_of_param_assignments ;
  | **parameter time** list_of_param_assignments ;
specparam_declaration ::= **specparam** [range] list_of_specparam_assignments ;

F.2.1.2 Port declarations

inout_declaration ::= **inout** [net_type] [**signed**] [range]
  list_of_port_identifiers
input_declaration ::= **input** [net_type] [**signed**] [range]
  list_of_port_identifiers
output_declaration ::=
  output [net_type] [**signed**] [range] list_of_port_identifiers
  | output [**reg**] [**signed**] [range] list_of_port_identifiers
  | **output reg** [**signed**] [range] list_of_variable_port_identifiers
  | **output** [output_variable_type] list_of_port_identifiers
  | **output** output_variable_type list_of_variable_port_identifiers

F.2.1.3 Type declarations

event_declaration ::= **event** list_of_event_identifiers ;
genvar_declaration ::= **genvar** list_of_genvar_identifiers ;
integer_declaration ::= **integer** list_of_variable_identifiers ;
net_declaration ::=
  net_type [**signed**]
    [delay3] list_of_net_identifiers ;
  | net_type [drive_strength] [**signed**]
    [delay3] list_of_net_decl_assignments ;
  | net_type [**vectored** | **scalared**] [**signed**]
    range [delay3] list_of_net_identifiers ;
  | net_type [drive_strength] [**vectored** | **scalared**] [**signed**]
    range [delay3] list_of_net_decl_assignments ;
  | **trireg** [charge_strength] [**signed**]

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

[delay3] list_of_net_identifiers ;
| **trireg** [drive_strength] [**signed**]
[delay3] list_of_net_decl_assignments ;
| **trireg** [charge_strength] [**vectored** | **scalared**] [**signed**]
range [delay3] list_of_net_identifiers ;
| **trireg** [drive_strength] [**vectored** | **scalared**] [**signed**]
range [delay3] list_of_net_decl_assignments ;
real_declaration ::= **real** list_of_real_identifiers ;
realtime_declaration ::= **realtime** list_of_real_identifiers ;
reg_declaration ::= **reg** [signed] [range]
list_of_variable_identifiers ;
time_declaration ::= time list_of_variable_identifiers ;

F.2.2 Declaration data types

F.2.2.1 Net and variable types

net_type ::=
supply0 | **supply1**
| **tri** | **triand** | **trior** | **tri0** | **tri1**
| **wire** | **wand** | **wor**
output_variable_type ::= **integer** | **time**
real_type ::=
real_identifier [= constant_expression]
| real_identifier dimension { dimension }
variable_type ::=
variable_identifier [= constant_expression]
| variable_identifier dimension { dimension }

F.2.2.2 Strengths

drive_strength ::=
(strength0 , strength1)
| (strength1 , strength0)
| (strength0 , **highz1**)
| (strength1 , **highz0**)
| (**highz0** , strength1)
| (**highz1** , strength0)
strength0 ::= **supply0** | **strong0** | **pull0** | **weak0**
strength1 ::= **supply1** | **strong1** | **pull1** | **weak1**
charge_strength ::= (**small**) | (**medium**) | (**large**)

F.2.2.3 Delays

delay3 ::= # delay_value | # (delay_value [, delay_value [, delay_value]])
delay2 ::= # delay_value | # (delay_value [, delay_value])
delay_value ::=
unsigned_number
| parameter_identifier
| specparam_identifier
| mintypmax_expression

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

F.2.3 Declaration lists

list_of_event_identifiers ::= event_identifier [dimension { dimension }]
{ , event_identifier [dimension { dimension }] }
list_of_genvar_identifiers ::= genvar_identifier { , genvar_identifier }
list_of_net_decl_assignments ::= net_decl_assignment { , net_decl_assignment }
list_of_net_identifiers ::= net_identifier [dimension { dimension }]
{ , net_identifier [dimension { dimension }] }
list_of_param_assignments ::= param_assignment { , param_assignment }
list_of_port_identifiers ::= port_identifier { , port_identifier }

list_of_real_identifiers ::= real_type { , real_type }
list_of_specparam_assignments ::= specparam_assignment { , specparam_assignment }
list_of_variable_identifiers ::= variable_type { , variable_type }
list_of_variable_port_identifiers ::= port_identifier [= constant_expression]
{ , port_identifier [= constant_expression] }

F.2.4 Declaration assignments

net_decl_assignment ::= net_identifier = expression
param_assignment ::= parameter_identifier = constant_expression
specparam_assignment ::=
specparam_identifier = constant_mintypmax_expression
| pulse_control_specparam
pulse_control_specparam ::=
PATHPULSE$ = (reject_limit_value [, error_limit_value]) ;
| PATHPULSE$specify_input_terminal_descriptor$specify
output_terminal_descriptor
= (reject_limit_value [, error_limit_value]) ;
error_limit_value ::= limit_value
reject_limit_value ::= limit_value
limit_value ::= constant_mintypmax_expression

F.2.5 Declaration ranges

dimension ::= [dimension_constant_expression : dimension_constant_expression]
range ::= [msb_constant_expression : lsb_constant_expression]

F.2.6 Function declarations

function_declaration ::=
function [**automatic**] [**signed**] [range_of_type] function_identifier;
function_item_declaration { function_item_declaration }
function_statement
endfunction
| **function** [**automatic**] [**signed**] [range_of_type]
function_identifier (function_port_list) ;
block_item_declaration { block_item_declaration }

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

function_statement
endfunction
function_item_declaration ::=
block_item_declaration
| tf_input_declaration ;
function_port_list ::= { attribute_instance } tf_input_declaration { , { attribute_instance } tf_input_declaration }
range_or_type ::= range | **integer** | **real** | **realtime** | **time**

F.2.7 Task declarations

task_declaration ::=
task [**automatic**] task_identifier;
{ task_item_declaration }
statement
endtask
| **task** [**automatic**] task_identifier (task_port_list) ;
{ block_item_declaration }
statement
endtask
task_item_declaration ::=
block_item_declaration
| { attribute_instance } tf_input_declaration ;
| { attribute_instance } tf_output_declaration ;
| { attribute_instance } tf_inout_declaration ;
task_port_list ::= task_port_item { ,task_port_item }
task_port_item ::=
{ attribute_instance } tf_input_declaration
| { attribute_instance } tf_outpur_declaration
| { attribute_instance } tf_inout_declaration
tf_input_declaration ::=
input [**reg**] [**signed**] [range] list_of_port_identifiers
| **input** [task_port_type] list_of_port_identifiers
tf_output_declaration ::=
output [**reg**] [**signed**] [range] list_of_port_identifiers
| **output** [task_port_type] list_of_port_identifiers
tf_inout_declaration ::=
inout [**reg**] [**signed**] [range] list_of_port_identifiers
| **inout** [task_port_type] list_of_port_identifiers
task_port_type ::=
time | **real** | **realtime** | **integer**

F.2.8 Block item declarations

block_item_declaration ::=
{ attribute_instance } block_reg_Ldeclaration
| { attribute_instance } event_declaration
| { attribute_instance } integer_declaration

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

```
        | { attribute_instance } local_parameter_declaration
        | { attribute_instance } parameter_declaration
        | { attribute_instance } real_declaration
        | { attribute_instance } realtime_declaration
        | { attribute_instance } time_declaration
block_reg_declaration ::= reg [signed] [ range ]
        list_of_block_variable_identifiers ;
list_of_block_variable_identifiers ::=
        block_variable_type { , block_variable_type }
block_variable_type ::=
        variable_identifier
        | variable_identifier dimension { dimension }
```

F.3 Primitive instances

F.3.1 Primitive instantiation and instances

```
gate_instantiation ::=
         cmos_switchtype [delay3]
                cmos_switch_instance { " cmos_switch_instance } ;
        | enable gatetype [ drive_strength ] [ delay3 ]
                enable gate_instance { , enable gate_instance } ;
        | mos_switchtype [delay3]
                mos_switch_instance { , mos_switch_instance } ;
        | n_input_gatetype [ drive_strength ] [ delay2 ]
                n_input gate_instance { , n_input_gate_instance } ;
        | n_output_gatetype [ drive_strength ] [ delay2 ]
                n_output_gate_instance { , n_output_gate_instance } ;
        | pass_en_switchtype [ delay2 ]
              pass_enable_switch_instance { , pass_enable_switch_instance } ;
        | pass_switchtype
         pass_switch_instance { , pass_switch_instance } ;
        | pulldown [ pulldown_strength ]
         pull_gate_instance { , pull_gate_instance } ;
        | pullup [ pull up_strength ]
         pull_gate_instance { , pull_gtate_instance } ;
cmos_switch_instance ::= [ name_of_gate_instance ] ( output_terminal , input_terminal ,
                ncontrol_terminal , pcontrol_terminal )
enable_gate_instance ::= [ name_of_gate_instance ] ( output_terminal , input_terminal ,
                enable_terminal )
mos_switch_instance ::= [ name_of_gate_instance ]
                ( output_terminal, input_terminal , enable_terminal)
n_input_gate_instance ::= [ name_of_gate_instance ]
                ( output_terminal , input_terminal { , input_terminal } )
n_output_gate_instance ::= [ name_of_gate_instance ]
                ( output_terminal { , output_terminal } , input_terminal )
pass_switch_instance ::= [ name_of_gate_instance ] ( inout_terminal , inout_terminal )
pass_enable_switch_instance ::= [ name_of_gate_instance ]
```

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

(inout_terminal , inout_terminal , enable_terminal)
pull_gate_instance ::= [name_of_gate_instance] (output_terminal)
name_of_gate_instance ::= gate_instance_identifier [range]

F.3.2 Primitive strengths

pulldown_strength ::=
(strength0 , strength1)
| (strength1 , strength0)
| (strength0)
pullup_strength ::=
(strength0 , strength1)
| (strengthl , strength0)
| (strength1)

F.3.3 Primitive terminals

enable_terminal ::= expression
inou_terminal ::= net_lvalue
input_terminal ::= expression
ncontrol_terminal ::= expression
output_terminal ::= net_lvalue
pcontrol_terminal ::= expression

F.3.4 Primitive gate and switch types

cmos_switch type ::= **cmos** | **rcmos**
enable_gatetype ::= **bufif0** | **bufifl** | **notif0** | **notif1**
mos_switchtype ::= **nmos** | **pmos** | **rnmos** | **rpmos**
n_input_gatetype ::= **and** | **nand** | **or** | **nor** | **xor** | **xnor**
n_output_gatetype ::= **buf** | **not**
pass_en_switchtype ::= **tranif0** | **tranifll rtranif1** | **rtranif0**
pass_switchtype ::= **tran** | **rtran**

F.4 Module and generated instantiation

F.4.1 Module instantiation

module_instantiation ::=
module_identifier [parameter_value_assignment]
module_instance { , module_instance } ;
parameter _value_assignment ::= **#** (list_of_parameter_assignments)
list_of_parameter_assignments ::=
ordered_parameter_assignment { , ordered_parameter_assignment } |
named_parameter_assignment { , named_parameter_assignment }
ordered_parameter_assignment ::= expression
named_parameter_assignment ::= . parameter_identifier ([expression])
module_instance ::= name_of_instance ([list_of_port_connections])

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

name_of_instance ::= module_instance_identifier [range]
list_of_port_connections ::=
    ordered_port_connection { , ordered_port_connection }
    | named_poet_connection { , named_port_connection })
ordered_port_connection ::= (attribute_instance) [expression]
named_port_connection ::= (attribute_instance) .port_identifier ([expression])

F.4.2 Generated instantiation

generated_instantiation ::= **generate** { generate_item } **endgenerate**
generate_item_or_null ::= generate_item | ;
generate_item ::=
    generate_conditional_statement
    | generate_case_statement
    | generate_loop_statement
    | generate_block
    | module_or_generate_item
generate_conditional_statement ::=
    if (constant_expression) generate_item_or_null [**else** generate_item_or_null]
generate_case_statement ::= **case** (constant_expression)
        genvar_case_item { genvar_case_item } **endcase**
genvar_case_item ::= constant_expression { , constant_expression } :
        generate_item_or_null | **default** [:] generate_item_or_null
generate_loop_statement ::= **for** (genvar_assignment ; constant_expression ;
  genvar_assignment)
      begin: generate_block_identifier { generate_item } **end**
genvar_assignment ::= genvar_identifier = constant_expression
generate_block ::= **begin** [: generate_block_identifier] { generate_item } **end**

F.5 UDP declaration and instantiation

F.5.1 UDP declaration

udp_declaration ::=
    { attribute_instance } **primitive** udp_identifier (udp_port_list) ;
    udp_port_declaration { udp_port_declaration }
    udp_body
    endprimitive
    | { attribute_instance } **primitive** udp_identifier (udp_declaration_port_list) ;
    udp_body
    endprimitive

F.5.2 UDP ports

udp_port_list ::= output_identifier , input_identifier { , input_identifier }
udp_declaration_port_list ::=
    udp_output_declaration , udp_input_declaration { , udp_input_ddeclaration }

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

udp_port_declaration ::=
 udp_output_declaration ;
 | udp_input_declaration ;
 | udp_reg_declaration ;
udp_output_declaration ::=
 { attribute_instance } **output** port_identifier
 | { attribute_instance } **output reg** port_identifier [= constant_expression]
udp_input_declaration ::= { attribute_instance } **input** list_of_port_identifiers
udp_reg_declaration ::= { attribute_instance } **reg** variable_identifier

F.5.3 UDP body

udp_body ::= combinational_body | sequential_body
combinational_body ::= **table** combinational_entry { combinational_entry} **endtable**
combinational_entry ::= level_input_list : output_symbol ;
sequential_body ::= [udp_initial_statement] **table** sequential_entry { sequential_entry} **endtable**
udp_initial_statement ::= **initial** output_port_identifier = init_val;
init_val::= **1'b0** | **1'b1** | **1'bx** | **1'bX** | **1'B0** | **1'B1** | **1'Bx** | **1'BX** | **1** | **0**
sequential_entry ::= seq_input_list: current_state: next_state;
seq_input_list ::= level_input_list I edge_input_list
level_input_list ::= level_symbol { level_symbol }
edge_input_list ::= { level_symbol } edge_indicator { level_symbol }
edge_indicator ::= (level_symbol | level_symbol) I edge_symbol
current_state ::= level_symbol
next_state ::= output_symbol | **-**
output_symbol ::= **0** | **1** | **x** | **X**
level_symbol ::= **0** | **1** | **x** | **X** | **?** | **b** | **B**
edge_symbol ::=**r** | **R** | **f** | **F** | **p** | **P** | **n** | **N** | *****

F.5.4 UDP instantiation

udp_instantiation ::= udp_identifier [drive_strength] [delay2]
 udp_instance { , udp_instance } ;
udp_instance ::= [name_of_udp_instance] (output_terminal , input_terminal
 { , input_terminal })
name_of_udp_instance ::= udp_instance_identifier [range]

F.6 Behavioral statements

F.6.1 Continuous assignment statements

continuous_assign ::= **assign** [drive_strength] [delay3] list_of_net_assignments ;
list_of_net_assignments ::= net_assignment { , net_assignment }
net_assignment ::= net_lvalue = expression

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

F.6.2 Procedural blocks and assignments

initial_construct ::= **initial** statement
always_construct ::= **always** statement
blocking_assignment ::= variable_lvalue = [delay_or_event_control] expression
nonblocking_assignment ::= variable_lvalue <= [delay_or_event_control] expression
procedural_continuous_assignments ::=
    assign variable_assignment
    | **deassign** variable_lvalue
    | **force** variable_assignment
    | **force** net_assignment
    | **release** variable_lvalue
    | **release** net_lvalue
function_blocking_assignment ::= variable_lvalue = expression
function_statement_or_null ::= function_statement I { attribute_instance } ;

F.6.3 Parallel and sequential blocks

function_seq_block ::= **begin** [: block_identifier
    { block_item_declaration }] { function_statement } **end**
variable_assignment ::= variable_lvalue = expression
par_block ::= **fork** [: block_identifier { block_item_declaration }] { statement } **join**
seq_block ::= **begin** [: block_identifier { block_item_declaration }] { statement } **end**

F.6.4 Statements

statement ::=
    { attribute_instance } blocking_assignment ;
    | { attribute_instance } case_statement
    | { attribute_instance } conditional_statement
    | { attribute_instance } disable_statement
    | { attribute_instance } event_trigger
    | { attribute_instance } loop_statement
    | { attribute_instance } nonblocking_assignment ;
    | { attribute_instance } par_block
    | { attribute_instance } procedural_continuous_assignments ;
    | { attribute_instance } procedural_timing_control_statement
    | { attribute_instance } seq_block
    | { attribute_instance } system_task_enable
    | { attribute_instance } task_enable
    | { attribute_instance } wait_statement
statement_or_null ::=
    statement
    | { attribute_instance } ;
function_statement ::=
    { attribute_instance } function_blocking_assignment ;
    | { attribute_instance } function_case_statement
    | { attribute_instance } function_conditional_statement
    | { attribute_instance } function_loop_statement

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

| { attribute_instance } function_seq_block
| { attribute_instance } disable_statement
| { attribute_instance } system_task_enable

F.6.5 Timing control statements

delay_control ::=
delay_value
| # (mintypmax_expression)
delay_or_event_control ::=
delay_control
| event_control
| **repeat** (expression) event_control
disable_statement ::=
disable hierarchical_task_identifier ;
| **disable** hierarchical_block_identifier;
event_control ::=
@ event_identifier
| @ (event_expression)
| @*
| @ (*)
event_trigger ::=
-> hierarchical_event_identifier ;
event_expression ::=
expression
| hierarchical_identifier
| **posedge** expression
| **negedge** expression
| event_expression or event_expression
| event_expression , event_expression
procedural_timing_control_statement ::=
delay_or_event_control statement_or_null
wait_statement ::=
wait (expression) statement_or_null

F.6.6 Conditional statements

conditional_statement ::=
if (expression)
statement_or_null [**else** statement_or_null]
| if_else_if_statement
if_**else**_if_statement ::=
if (expression) statement_or_null
{ **else** if (expression) statement_or_null }
[**else** statement_or_null]
function_conditional_statement ::=
if (expression) function_statement_or_null
[**else** function_statement_or_null]
| function_if_else_if_statement

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

```
function_if_else_if_statement ::=
        if ( expression ) function_statement_or_null
        { else if ( expression ) function_statement_or_null }
        [ else function_statement_or_null ]
```

F.6.7 Case statements

```
case_statement ::=
        case ( expression )
              case_item { case_item } endcase
        | casez ( expression )
              case_item { case_item } endcase
        | casex ( expression )
              case_item { case_item } endcase
case_item ::=
        expression { , expression } : statement_or_null
        | default [ : ] statement_or_null
function_case_statement ::=
        case ( expression )
        function_case_item ( function_case_item } endcase
        | casez ( expression )
        function_case_item { function_case_item } endcase
        | casex ( expression )
        function_case_item { function_case_item } endcase
function_case_item ::=
        expression {, expression } : function_statement_or_null
        | default [ : ] function_statement_or_null
```

F.6.8 Looping statements

```
function_loop_statement ::=
        forever function_statement
        | repeat ( expression ) function_statement
        | while ( expression ) function_statement
        | for ( variable_assignment; expression; variable_assignment)
              function_statement
loop_statement ::=
        forever statement
        | repeat ( expression ) statement
        | while ( expression ) statement
        | for ( variable_assignment; expression; variable_assignment ) statement
```

F.6.9 Task enable statements

```
system_task_enable ::= system_task_identifier [ ( expression { , expression } ) ] ;
task_enable ::= hierarchical_task_identifier [ ( expression { , expression } ) ] ;
```

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

F.7 Specify section

F.7.1 Specify block declaration

specify_block ::= **specify** { specify_item } **endspecify**
specify_item ::=
    specparam_declaration
    | pulsestyle_declaration
    | showcancelled_declaration
    | path_declaration
    | system_timing_check
pulsestyle_declaration ::=
    pulsestyle_onevent list_of_path_outputs ;
    | **pulsestyle_ondetect** list_of_path_outputs ;
showcancelled_declaration ::=
    showcancelled list_of_path_outputs ;
    | **noshowcancelled** list_of_path_outputs ;

F.7.2 Specify path declarations

path_declaration ::=
    simple_path_declaration ;
    | edge_sensitive_path_declaration ;
    | state_dependent_path_declaration ;
simple_path_declaration ::=
    parallel_path_description = path_delay _value
    | full_path_description = path_delay_value
parallel_path_description ::=
    (specify _input_terminal_descriptor [polarity_operator] => specify
    _output_terminal_descriptor)
full_path_description ::=
    (list_of_path_inputs [polarity_operator] *> list_of_path_outputs)
list_of_path_inputs ::=
    specify_input_terminal_descriptor { , specify_input_terminal_descriptor }
list_of_path_outputs ::=
    specify_output_terminal_descriptor { , specify_output_terminal_descriptor }

F.7.3 Specify block terminals

specify_input_terminaLdescriptor ::=
    input_dentifier
    | input_identifier [constant_expression]
    | input_dentifier [range_expression]
specify_output_terminal_descriptor ::=
    output_identifier
    | output_identifier [constant_expression]
    | output_identifier [range_expression]
input_identifier ::= input_port_identifier | inout_port_identifier
output_dentifier ::= output_port_identifier | inout_port_identifier

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

F.7.4 Specify path delays

```
path_delay_value ::=
        list_of_path_delay _expressions
       | (list_of_path_delay_expressions)
list_of_path_delay _expressions ::=
       t_path_delay _expression
       | trise_path_delay_expression , tfall_path_delay _expression
       | trise_path_delay_expression , tfall_path_delay_expression ,
        tz_path_delay_expression
       | t01_path_delay_expression , t10_path_delay_expression ,
        t0z_path_delay_expression, tz1_path_delay_expression ,
        t1z_path_delay_expression , tz0_path_delay_expression
       | t01_path_delay_expression , t10_path_delay_expression ,
        t0z_path_delay_expression,
        tz1_path_delay_expression , t1z_path_delay_expression ,
        tz0_path_delay_expression
        t0x_path_delay_expression , tx1_path_delay_expression ,
        t1x_path_delay_expression,
        tx0_path_delay_expression , txz_path_delay_expression ,
        tzx_path_delay_expression
t_path_delay_expression ::= path_delay_expression
trise_path_delay_expression ::= path_delay_expression
tfall_path_delay_expression ::= path_delay_expression
tz_path_delay_expression ::= path_delay_expression
t01_path_delay_expression ::= path_delay_expression
t10_path_delay_expression ::= path_delay_expression
t0z_path_delay _expression ::= path_delay_expression
tz1_path_delay_expression ::= path_delay_expression
t1z_path_delay_expression ::= path_delay_expression
tz0_path_delay_expression ::= path_delay_expression
t0x_path_delay_expression ::= path_delay_expression
tx1_path_delay_expression ::= path_delay_expression
t1x_path_delay_expression ::= path_delay_expression
tx0_path_delay_expression ::= path_delay_expression
txz_path_delay_expression ::= path_delay_expression
tzx_path_delay_expression ::= path_delay_expression
path_delay_expression ::= constant_mintypmax_expression
edge_sensitive_path_declaration ::=
        parallel_edge_sensitive_path_description = path_delay_value
       | full_edge_sensitive_path_description = path_delay _value
parallel_edge_sensitive_path_description ::=
        ( [ edge_identifier ] specify_input_terminal_descriptor =>
       specify_output_terminal_descriptor [ polarity_operator ] :
        data_source_expression )
full_edge_sensitive_path_description ::=
        ( [ edge_identifier ] list_of_path_inputs *>
        list_of_path_outputs [ polarity_operator ] : data_source_expression )
data_source_expression ::= expression
```

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

```
edge_identifier ::= posedge | negedge
state_dependent_path_declaration ::=
        if ( module_path_expression ) simple_path_declaration
        | if ( module_path_expression ) edge_sensitive_path_declaration
        | ifnone simple_path_declaration
polarity_operator ::= + | -
```

F.7.5 System timing checks

F.7.5.1 System timing check commands

```
system_timing_check ::=
        $setup_timing_check
        | $hold _timing_check
        | $setuphold_timing_check
        | $recovery_timing_check
        | $removal_timing_check
        | $recrem_timing_check
        | $skew timing_check
        | $timeskew_timing_check
        | $fullskew_timing_check
        | $period_timing_check
        | $width_timing_check
        | $nochange_timing_check
$setup_timing_check ::=
        $setup (data_event , reference_event , timing_check_limit [ , [ notify_reg ] ] ) ;
$hold _timing_check ::=
        $hold ( reference_event , data_event , timing_check_limit [ , [ notify_reg] ] ) ;
$setuphold_timing_check ::=
        $setuphold ( reference_event , data_event, timing_check_limit ,
         timing_check_limit
                [ , [ notify_reg ] [ , [ stamptime_condition] [, [ checktime_condition]
                 [ , [ delayed_reference] [ , [ delayed_data ] ] ] ] ] ] ) ;
$recovery_timin_check ::=
        $recovery ( reference_event , data_event, timing_check_limit [ ,
         [ notify_reg ] ] ) ;
$removal_timing_check ::=
        $removal ( reference_event , data_event , timing_check_limit [ , [
         notify_reg ] ] ) ;
$recrem_timing_check ::=
        $recrem ( reference_event , data_event , timing_check_limit ,
         timing_check_limit
                [ , [ notify_reg] [ , [ stamptime_condition ] [ , [ checktime_condition ]
                [ , [ delayed_reference ] [ , [ delayed_data] ] ] ] ] ] ) ;
$skew_timing_check ::=
        $skew ( reference_event , data_event , timing_check_limit [ , [ notify_reg ] ] ) ;
$timeskew_timing_check ::=
        $timeskew ( reference_event , data_event , timing_check_limit
                [ , [ notify_reg ] [ , [ event_based_flag ] [ , [ remain_active_flag ] ] ] ] ) ;
```

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

```
$fullskew_timing_check ::=
        $fullskew ( reference_event, data_event , timing_check_limit ,
         timing_check_limit
              [ , [ notify_reg ] [ , [ event_based_flag ] [ , [ remain_active_flag ] ] ] ] ] ) ;
$period_timing_check ::=
        $period (controlled_reference_event , timing_check_limit [ , [ notify_reg ] ] ) ;
$width_timing_check ::=
        $width ( controlled_reference_event , timing_check_limit , threshold [ , [
         notify_reg ] ] ) ;
$nochange_timing_check ::=
        $nochange ( reference_event , data_event , start_edge_offset ,
                 end_edge_offset [ , [ notify_reg ] ] ) ;
```

F.7.5.2 System timing check command arguments

```
checktime_condition ::= mintypmax_expression
controlled_reference_event ::= controlled_timing_check_event
data_event ::= timing_check_event
delayed_data ::=
        terminal_identifier
       | terminal_identifier [ constant_mintypmax_expression ]
delayed_reference ::=
        terminal_identifier
       | terminal_identifier [ constant_mintypmax_expression ]
end_edge_offset ::= mintypmax_expression
event_based_flag ::= constant_expression
notify_reg ::= variable_identifier
reference_event ::= timing_check_event
remain_active_flag ::= constant_mintypmax_expression
stamptime_condition ::= mintypmax_expression
start_edge_offset ::= mintypmax_expression
threshold ::=constant_expression
timing_check_limit ::= expression
```

F.7.5.3 System timing check event definitions

```
timing_check_event ::=
        [timing_check_event_control] specify_terminal_descriptor [ &&&
         timing_check_condition ]
controlled_timing_check_event ::=
        [ timing_check_event_control specify_terminal_descriptor [ &&&
         timing_check_condition ]
timing_check_event_control ::=
        posedge
       | negedge
       | edge_control_specifier
specify_terminal_descriptor ::=
        specify_input_terminal_descriptor
       | specify _output_terminal_descriptor
edge_control_specifier ::= edge [ edge_descriptor [ , edge_descriptor ] ]
```

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

```
edge_descriptor ::=
        01
        | 10
        | z_or_x zero_or_one
        | zero_or_one z_or_x
zero_or_one ::= 0 | 1
z_or_x ::= x | X | z | Z
timing_check_condition ::=
        scalar_timing_check_condition
        | ( scalar_timing_check_condition )
scalar_timing_check_condition ::=
        expression
        | ~expression
        | expression == scalar_constant
        | expression === scalar_constant
        | expression != scalar_constant
        | expression !== scalar_constant
scalar_constant ::=
        1'b0 | 1'b1 | 1'B0 | 1'B1 | 'b0 | 'b1 | 'B0 | 'B1 | 1 | 0
```

F.8 Expressions

F.8.1 Concatenations

```
concatenation ::= { expression { , expression } }
constant_concatenation ::= { constant_expression { , constant_expression } }
constant_multiple_concatenation ::= { constant_expression constant_concatenation }
module_path_concatenation ::= { module_path_expression { ,
  module_path_expression } }
  module_path_multiple_concatenation ::= { constant_expression
  module_path_concatenation }
  multiple_concatenation ::= { constant_expression concatenation }
net_concatenation ::= { net_concatenation_value { , net_concatenation_value } }
net_concatenation_value ::=
        hierarchical_net_identifier
        | hierarchical_net_identifier [ expression ] { [ expression ] }
        | hierarchical_net_dentifier [ expression ] { [ expression] } [ range_expression ]
        | hierarchical_net_identifier [ range_expression ]
        | net_concatenation
variable_concatenation ::= { variable_concatenation_value { ,
  variable_concatenation_value } }
variable_concatenation- value ::=
        hierarchical_variable_identifier
        | hierarchical_variable_identifier [ expression ] { [ expression ] }
        | hierarchical_variable_identifier [ expression ] { [ expression ] } [
          range_expression ]
        | hierarchical_variable_identifier [ range_expression ]
        | variable_concatenation
```

* From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

F.8.2 Function calls

constant_function_call ::= function_identifier { attribute_instance }
(constant_expression { , constant_expression })
function_call ::= hierarchical_function_identifier { attribute_instance }
(expression { , expression })
genvar_function_call ::= genvar_function_identifier { attribute_instance }
(constant_expression { , constant_expression })
system_function_call ::= system_function_identifier
[(expression { , expression })]

F.8.3 Expressions

base_expression ::= expression
conditional_expression ::= expression1 **?** { attribute_instance } expression2 **:** expression3
constant_base_expression ::= constant_expression
constant_expression ::=
constant_primary
| unary_operator { attribute_instance } constant_primary
| constant_expression binary_operator (attribute_instance }
constant_expression
| constant_expression **?** (attribute_instance } constant_expression **:**
constant_expression | string
constant_mintypmax_expression ::=
constant_expression
| constant_expression **:** constant_expression **:** constant_expression
constant_range_expression ::=
constant_expression
| msb_constant_expression : lsb_constant_expression
| constant_base_expression **+:** width_constant_expression
| constant_base_expression **-:** width_constant_expression
dimension_constant_expression ::= constant_expression
expression1 ::= expression
expression2 ::= expression
expression3 ::= expression
expression ::=
primary
| unary_operator { attribute_instance } primary
| expression binary_operator { attribute_instance } expression
| conditional_expression
| string
lsb_constant_expression ::= constant_expression
mintypmax...expression ::=
expression
| expression: expression: expression
module_path_conditional_expression ::= module_path_expression **?** {
attribute_instance }
module_path_expression **:** module_path_expression

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

```
module_path_expression ::=
        module_path_primary
      | unary_module_path_operator { attribute_instance } module_path_primary
      | module_path_expression binary_module_path_operator { attribute_instance }
        module_path_expression
      | module_path_conditional_expression
module_path_mintypmax_expression ::=
        module_path_expression
      | module_path_expression : module_path_expression :
         module_path_expression
msb_constant_expression ::= constant_expression
range_expression ::=
        expression
      | msb_constant_expression : lsb_constant_expression
      | base_expression +: width_constant_expression
      | base_expression -: width_constant_expression
width_constant_expression ::= constant_expression
```

F.8.4 Primaries

```
constant_primary ::=
        constant_concatenation
      | constant_function_call
      | ( constant_mintypmax_expression )
      | constant_multiple_concatenation
      | genvar_identifier
      | number
      | parameter_identifier
      | specparam_identifier
module_path_primary ::=
        number
      | identifier
      | module_path_concatenation
      | module_path_multiple_concatenation
      | function_call
      | system_function_call
      | constant_function_call
      | ( module_path_mintypmax_expression )
primary ::=
        number
      | hierarchical_identifier
      | hierarchical_identifier [ expression ] { [ expression ] }
      | hierarchical_identifier [ expression ] { [expression ] } [ range_expression ]
      | concatenation
      | multiple_concatenation
      | function_call
      | system_function_call
      | constant function_call
      | ( mintypmax_expression )
```

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

F.8.5 Expression left-side values

```
net_lvalue ::=
        hierarchical_net_identifier
      | hierarchical_net_identifier [ constant_expression ] { [constant_expression ] }
      | hierarchical_net_identifier [ constant_expression ] { [ constant_expression ] } [
            constant_range_expression ]
      | hierarchical_net_identifier [ constant_range_expression ]
      | net_concatenation
variable_lvalue ::=
        hierarchical_variable_identifier
      | hierarchical_variable_identifier [ expression ] { [ expression ] }
      | hierarchical_variable_identifier [ expression ] { [expression ] }
        [range_expression ]
      | hierarchical_variable_identifier [ range_expression ]
      | variable_concatenation
```

F.8.6 Operators

```
unary_operator ::=
        +| - | ! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~
binary_operator ::=
        + | - | * | / | % | == | != | === | !== | && | || | **
        | < | <= | > | >= | & | | | ^ | ^~ | ~^ | >> | << | <<< | >>>
unary _module_path_operator ::=
        ! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~
binary _module_path_operator ::=
        == | != | && | || | & | | | ^ | ^~| ~^
```

F.8.7 Numbers

```
number ::=
        decimal_number
      | octal_number
      | binary_number
      | hex_number
      | real_number
real_number ::= unsigned_number . unsigned_number
      | unsigned_number [ . unsigned_number] exp [ sign]
unsigned_number exp ::= e | E
decimal_number: :=
        unsigned_number
      | [ size ] decimal_base unsigned_number
      | [ size ] decimal_base x_digit { _ }
      | [ size ] decimal_base z_digit { _ }
binary_number ::= [ size ] binary_base binary_value
octal_number ::= [ size ] octal_base octal_value
hex_number ::= [ size ] hex_base hex_value
sign ::= + | -
```

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

size ::= non_zero_unsigned_number
non_zero_unsigned_number ::= non_zero_decimal_digit { _ | decimal_digit }
unsigned_number ::= decimal_digit { _ | decimal_digit }
binary_value ::= binary_digit { _ | binary_digit }
octal_value ::= octal_digit { _ | octal_digit }
hex_value ::= hex_digit { _ | hex_digit }
decimal_base ::= **'[s|S]d | '[s|S]D**
binary_base ::= **'[s|S]b | '[s|S]B**
octal_base ::= **'[s|S]o | '[s|S]O**
hex_base ::= **'[s|S]h | '[s|S]H**
non_zero_decimal_digit ::= **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
decimal_digit::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
binary_digit ::= x_digit | z_digit | **0** | **1**
octal_digit ::= x_digit | z_digit | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7**
hex_digit ::=
    x_digit | z_digit | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
    | **a** | **b** | **c** | **d** | **e** | **f** | **A** | **B** | **C** | **D** | **E** | **F**
x_digit ::= **x** | **X**
z_digit ::= **z** | **Z** | **?**

F.8.8 Strings

string ::= " { Any_ASCII_Characters_except_new_line } "

F.9 General

F.9.1 Attributes

attribute_instance ::= (* attr_spec { , attr_spec} *)
attr_spec ::=
    attr_name = constant_expression
    | attr_name
attr_name ::= identifier

F.9.2 Comments

comment ::=
    one_line_comment
    | block_comment
one_line_comment ::= // comment_text \n
block_comment ::= /* comment_text */
comment_text ::= { Any_ASCII_character }

F.9.3 Identifiers

arrayed_identifier ::=
    simple _arrayed _identifier

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

```
        | escaped_arrayed_identifier
block_identifier ::= identifier
cell_identifier ::= identifier
config_identifier ::= identifier
escaped_arrayed_identifier ::= escaped_identifier [ range ]
escaped_hierarchical_identifier ::=
        escaped_hierarchical_branch
            { . simple_hierarchical_branch | . escaped_hierarchical_branch }
escaped_identifier ::= \ {Any_ASCII_character_except_white_space} white_space
event_identifier ::= identifier
function_identifier ::= identifier
gate_instance_identifier ::= arrayed_identifier
generate_block_identifier ::= identifier
genvar_function_identifier ::= identifier /* Hierarchy disallowed */
genvar_identifier ::= identifier
hierarchical_block_identifier ::= hierarchical_identifier
hierarchical_event_identifier ::= hierarchical_identifier
hierarchical_function_identifier ::= hierarchical_identifier
hierarchical_identifier ::=
        simple hierarchical_identifier
        | escaped_hierarchical_identifier
hierarchical_net_identifier ::= hierarchical_identifier
hierarchical_variable_identifier ::= hierarchical_identifier
hierarchical_task_identifier ::= hierarchical_identifier
identifier ::=
        simple_identifier
        | escaped_identifier
inout_port_identifier ::= identifier
input_port_identifier ::= identifier
instance_identifier ::= identifier
library_identifier ::= identifier
memory_identifier ::= identifier
module_identifier ::= identifier
module_instance_identifier ::= arrayed_identifier
net_identifier ::= identifier
output_port_identifier ::= identifier
parameter_identifier ::= identifier
port_identifier ::= identifier
real_identifier ::= identifier
simple_arrayed_identifier ::= simple_identifier [ range ]
simple_hierarchical_identifier ::=
        simple_hierarchical_branch [ .escaped_identifier ]
simple_identifier ::= [ a-zA-Z_ ] { [ a-zA-Z0-9_$ ] }
specparam_identifier ::= identifier
system_function_identifier ::= $[ a-zA-Z0-9_$]{ [a-zA-Z0-9_$] }
system_task_identifier ::= $[ a-zA-Z0-9_$]{ [a-zA-Z0-9_$] }
task_identifier ::= identifier
```

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

terminal_identifier ::= identifier
text_macro_identifier ::= simple_identifier
topmodule_identifier ::= identifier
udp_identifier ::= identifier
udp_instance_identifier ::= arrayed_identifier
variable_identifier ::= identifier

F.9.4 Identifier branches

simple_hierarchical_branch ::=
  simple_identifier [[unsigned_number]]
    [{ .simple_identifier [[unsigned_number]] }]
escaped_hierarchical_branch ::=
  escaped_identifier [[unsigned_number]]
    [{ .escaped_identifier [[unsigned_number]] }]

F.9.5 White space

white_space ::= space | tab | newline | eof

NOTES:

1. Embedded spaces are illegal.
2. A simple_identifier and arrayed_reference shall start with an alpha or underscore (_) character, shall have at least one character, and shall not have any spaces.
3. The period (.) in simple_hierarchical_identifier and simple_hierarchical- branch shall not be preceded or followed by white_space.
4. The period in escaped_hierarchical_identifier and escaped_hierarchical- branch shall be preceded by white_space, but shall not be followed by white_space.
5. The $ character in a system_function_identifier or system_task_identifier shall not be followed by white_space. A system_function_identifier or system_task_identifier shall not be escaped.
6. End of file.

From IEEE Std. 1364-2001, IEEE Std. 1364-2005. Copyright 2005 IEEE. All rights reserved.

APPENDIX G

Additional Features of Verilog

G.1 Arrays of Primitives

Declaring a range between the key word and ports of a primitive forms an array of instances of the primitive.

Example

The description in *array_of_nor* contains a declaration of 8-bit input and output datapaths. The declared instance of the ***nor*** primitive with an 8-bit range specification creates a structure of 8 ***nor*** gates. The individual bits of the datapaths are automatically connected in sequential order to the inputs of the corresponding gate. The array structure is shown in Figure G-1.

```
module array_of_nor (output [0: 7] y, input [0: 7] a, b,);
  nor [0:7] M0 (y, a, b);
endmodule
```

End of Example

G.2 Arrays of Modules

Declaring a range between the instance name of a module and its ports forms an array of instances of the module. (*Note:* The list of ports in an array of instances must be compatible with the structure being instantiated. If the port of an instantiated object is

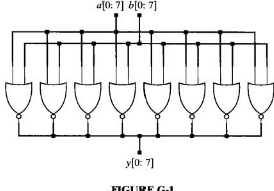


FIGURE G-1

an array, then the size of the port in the instantiated array of instances of the object must be sufficiently large to accommodate all copies of the object.)

Example

An array of full adders connected to form 4-bit slice ripple-carry adders is described in *array_of_adders*.

```
module array_of_adders (output [3: 0] sum, output c_out, input [3: 0] a, b,
  input c_in);
  wire [3:1] carry;
  Add_full M[3:0] (sum, {c_out, carry[3:1]}, a, b, {carry[3:1], c_in});
endmodule
```

End of Example

G.3 Hierarchical Dereferencing

An identifier must be associated with only one object within a scope or domain within which the identifier has unique meaning (i.e., within a module, named procedural block, task, or function). Consequently, a variable may be referenced directly by its identifier within the scope in which it is declared. Verilog also supports hierarchical dereferencing by a variable's hierarchical path name. This feature allows testbenches to monitor the activity of variables at any location within the hierarchical decomposition of the unit under test. If a variable is referenced but not declared locally, Verilog will search upward through the boundaries of named blocks, tasks, and functions to resolve the identifier, but it will not search beyond a module boundary.

G.4 Parameter Substitution

Verilog 1995 supports two methods of changing the values of parameters in a module: direct substitution and indirect referencing. Direct substitution overrides the value of the parameter on a module instance basis. See Appendix I for enhancements for parameter substitution in Verilog 2001, 2005.

Example

The parameters declared within the *G2* instance of *modXnor* are overridden by including #(4,5) in the instantiation of the module. The values given in the instantiation replace the values of *size* and *delay* that were given in the declaration of *modXnor*. The replacement is made in the order that the parameters were originally declared. This method can be cumbersome if the edited value is near the end of a long list.

```
module modXnor #(parameter size = 8, delay = 15) (output [size-1:0]y_out, input
  [size-1:0]a, b);
  assign #delay y_out = a ~^b;          //bitwise xnor
endmodule

module Param (output [7: 0] y1_out, output [3: 0] y2_out, input [7: 0] a1, b1,
  input [3: 0] a2, b2);
  modXnor G1 (y1_out, a1, b1);                //Uses default parameters
  modXnor #(4, 5) G2 (y2_out, a2, b2);        //Overrides default parameters
endmodule
```

End of Example

Indirect substitution uses hierarchical dereferencing to override the value of a parameter in a module. Declaring a separate module in which the ***defparam*** statement is used with the hierarchical path name of the parameters that are to be overridden most conveniently does this. (*Note:* This feature can be misused by annotating from anywhere within a design hierarchy.)

Example

In *hdref_param* the values of *size* and *delay* in instance *G2* of *modXnor* are overridden by the statements in the module *annotate*.

```
module hdref_Param (output [7:0] y1_out, output [3: 0] y2_out, input [7:0] a1, b1,
  input [3:0] a2, b2);
  modXnor          G1 (y1_out, a1, b1),
                   G2 (y2_out, a2, b2);           //instantiation
endmodule
```

```
module annotate:                                    //a separate "annotation"
                                                      module
  defparam
  hdref_Param.G2.size = 4.                          //parameter
                                                      assignment by
  hdref_Param.G2.delay = 5;                         //hierarchical reference
                                                      name
endmodule
module modXnor #(parameter size = 8, delay = 15) (output [size -1: 0] y_out,
  input [size -1: 0] a, b);
   assign #delay y_out = a ~^b;                     //bitwise xnor
endmodule
```

End of Example

G.5 Procedural Continuous Assignment

There are two constructs for procedural continuous assignments, which declare dynamic bindings to nets or registers in a model. Ordinarily, a continuous assignment remains in effect for the duration of a simulation. A ***assign* . . . *deassign*** procedural continuous assignment (PCA) is made by a procedural statement to establish an alternative binding (i.e., dynamically substitute the righthand expression). PCAs using the keyword ***assign*** are used to model the level-sensitive behavior of combinational logic, transparent latches, and asynchronous control of sequential parts. The binding remains in effect until it is removed by the (optional) ***deassign*** key word or until another procedural continuous assignment is made.

Example

The four-channel mux below uses the ***assign* . . . *deassign*** PCA to bind the output to a selected datapath.

```
module mux4_PCA (input a, b, c, d, input [1: 0] select, output reg y_out);
 always @ (select)
   if (select == 0) assign y_out = a; else
   if (select == 1) assign y_out = b; else
   if (select == 2) assign y_out = c; else
   if (select == 3) assign y_out = d; else assign y_out = 1'bx;
endmodule
```

End of Example

The ***force* . . . *release*** form of procedural continuous assignment applies to register variables as well as nets, and overrides ***assign* . . . *deassign*** continuous assignments. The ***force* . . . *release*** construct is used primarily within testbenches to inject logic values or logic into a design. See *t_ASIC_with_JTAG* in Section 11.10.6.

Example

In synchronous operation, the *data* input of a D-type flip-flop is transferred to the *q* output at the synchronizing edge of *clock* (e.g., at the rising edge or the falling edge of the synchronizing signal). If either the *preset* or the *clear* signal is asserted, this (synchronous) clocking action is ignored and the output is held at a constant value. A Verilog model of this behavior is shown below for active-low *preset* and *clear*. The PCA has immediate effect when the statement executes. While *preset* or *clear* is asserted, the ordinary synchronous behavior is ignored. If both *clear* and *reset* are de-assigned, the synchronous activity commences with the next active edge of the clock after the deassignment executes.

```
module FLOP_PCA (output reg q, output qbar, input data, preset, clear, clock);
 assign              qbar = ~q;
 always @ (negedge clock) q <= data;
  always @ (clear, preset) begin
 if (!clear) assign q = 0;
   else if (!preset) assign q = 1;
   else deassign q;
 end
endmodule
```

End of Example

G.6 Intra-Assignment Delay

When a timing control operator (**#** or **@**) appears in front of a procedural statement in a behavioral model the delay is referred to as a "blocking" delay, and the statement that follows the operator is said to be "blocked." The statement that follows a blocked statement in the sequential flow cannot execute until the preceding statement has completed execution. Verilog supports another form of delay in which a timing control is placed to the righthand side of the assignment operator *within* an assignment statement. This type of delay, called *intra-assignment delay*, evaluates the righthand side expression of the assignment and then schedules the assignment to occur in the future, at a time determined by the timing control. Ordinary delay control postpones the execution of a statement; intra-assignment delay postpones the occurrence of the assignment that results from executing a statement. A statement in a list of blocked procedural assignments (i.e. those using the = operator, must complete before the statement after it can execute).

Example

When the first statement is encountered in the sequential activity flow below, the value of B is sampled and scheduled to be assigned to A five time units later. The statement does not complete execution until the assignment occurs. After the assignment to A is

made, the next statement can execute Thus, C gets D 5 time units after the first statement is encountered in simulation.

```
...
A = #5 B;
C = D;
...
```

End of Example

Intra-assignment delay control (#) has the effect of causing the righthand-side expression of an assignment to be evaluated immediately when the procedural statement is encountered in the activity flow. However, the assignment that results from the statement is not executed until the specified delay has elapsed. Thus, referencing and evaluation are separated in time from the actual assignment of value to the target register variable.

Intra-assignment delay can also be implemented with the event-control operator and an event control expression. In this case, the execution of the statement is scheduled subject to the occurrence of the event specified in the expression.

Example

In the description below G gets $ACCUM$ when A_BUS changes. As a result of the intra-assignment delay, the procedural assignment to G cannot complete execution until A_BUS changes. The statement $C = D$ is blocked until G gets value. The value that G gets is the value of $ACCUM$ when the statement is encountered in the activity flow. This may differ from the actual value of $ACCUM$ when A_BUS finally has activity.

```
...
G = @ (A_BUS) ACCUM;
C = D;
...
```

End of Example

G.7 Indeterminate Assignment and Race Conditions

Multiple concurrent behaviors (i.e., ***always*** or ***initial*** blocks) may assign value to the same register variable at the same time step. A simulator must determine the outcome of these multiple assignments and distinguish between blocking (=) and nonblocking (<=) assignments. The activity of a simulator is triggered by an event (i.e., a change in the value

of a net, a register variable, or the triggering of an abstract event). (see Section G.10). The processing steps of the simulator are organized to establish an event queue to determine the order in which assignments to variables occur in simulation. Consequently, the queue manages the assignments to registers when nonblocking and blocking assignments are made simultaneously to the same target variable (i.e., in the same time step). At a given time step, the simulator will (1) evaluate the expressions on the righthand side (RHS) of all the assignments to register variables in statements that are encountered at that time step, (2) execute the blocking assignments to registers, (3) execute nonblocking assignments that do not have intra-assignment timing controls (i.e., they execute in the current time step), (4) execute past procedural assignments whose timing controls have scheduled an assignment for the current simulator time, and (5) advance the simulator time (t_{sim}). The language reference manual (LRM) for Verilog refers to this organization of the simulation activity as a "stratified-event queue." That is, the queue of pending simulation events is organized into five different regions, as shown in Table G-1.

The first region, the active region, consists of events that are scheduled to occur at the current simulation time, and which have top priority for execution. These events result from (1) evaluating the RHS of nonblocking assignments, (2) evaluating the inputs of a primitive and changing the output, (3) executing a procedural (blocked) assignment to a register variable, (4) evaluating the RHS of a continuous assignment and updating the LHS, (5) evaluating the RHS of a procedural continuous assignment and updating the LHS, and (6) evaluating and executing ***$display*** and ***$write*** system tasks. Any procedural assignments blocked by a #0 delay control are placed in the inactive queue, and execute after the active queue is empty, in the next simulation cycle at the current time-step of the simulator. The activity of the active queue is dynamic. When it becomes empty, the contents of the inactive queue are moved to the active queue, and the process continues.

The order of processing events in the active queue is not specified by the LRM and is tool-dependent. For example, if an input to a module at the top level of the design hierarchy has an event at the current simulation time, as prescribed by a testbench, the event would reside in the active area of the queue. Now suppose that the input to the module is connected to a primitive with zero propagation delay and whose output is changed by the event on the input port. This event would also be scheduled to occur at the current simulation time and would also be placed in the active area of the queue. If a behavior is activated by the module input and if the behavior generates an event by means of a nonblocking assignment, that event would be placed in the nonblocking assignment update area of the queue. Events that were scheduled to occur at the current simulation time but that originated in nonblocking assignments at an earlier simulation time would also be placed in the "nonblocking assignment update" area. The monitor area contains events that are to be processed after the active, inactive, and nonblocking assignment update events, such as the ***$monitor*** task. The last area of the stratified event queue consists of events that are to be executed in the future. Given this organization of the event queue, *the simulator executes all of the active events in a single simulation cycle*. As it executes, it may add events to any of the regions of the queue, but it may delete events only from the active region. After the active region is empty, the events in the inactive region are activated (i.e., they are

TABLE G-1

Class of event	Time of occurrence	Order of processing
Active event		
■ Evaluate RHS of nonblocking assignments ■ Evaluate inputs to primitives and update their outputs ■ Execute blocking assignments ■ Update continuous assignments ■ Update procedural continuous assignments ■ Execute **$display** and **$write** statements	Current t_{sim}	In any order
Inactive	Current t_{sim} with #0 blocking assignments	In any order after all active events
Nonblocking assignment update	Evaluated during previous or present t_{sim} to be assigned at t_{sim}	In any order after all active and inactive events
Monitor	Current t_{sim}	After all active, inactive, and nonblocking assignment update events
Future active and nonblocking assignment update	A future simulation time	

Simulation cycle: the processing of all of the events in the active event queue. An explicit #0 delay control requires that the executing process be suspended and added as an inactive event for the current simulation time, so that the process is resumed in the next simulation cycle (IEEE 1364).

added to the active region and a new simulation cycle begins). After the active region and the inactive region are both empty, the events in the nonblocking assignment update area of the queue are activated, and a new simulation cycle begins. After the monitor events have executed, the simulator advances to the next time at which an event is scheduled to occur. Whenever an explicit #0, delay control is encountered in a behavior, the associated process is suspended and added as an inactive event for the current simulation time. The process will be resumed in the next simulation cycle at the current time.

In addition to the structure imposed by the stratified event queue, the simulator must also adhere to the rule that the relative ordering of blocking and nonblocking assignments at the same simulation time will be such that the nonblocking assignments will be scheduled after the blocking assignments, with the exception that blocking assignments that are triggered by nonblocking assignments will be scheduled after the nonblocking assignments that are already scheduled. (*Caution:* The ***$display*** task executes immediately when it is encountered in the sequential activity flow of a behavior.) The ***$monitor*** task executes at the end of the current simulation cycle (i.e., after the nonblocking assignments have been updated). Thus, in the code below, *execute_display* assigns value to a and b, samples the current RHS of a and b, displays the current values of a and b, then updates a and b. The values of a and b at the end of the behavior are not the values that were displayed (i.e., ***$display*** executes before the nonblocking assignments). On the other hand, *execute_monitor* assigns value to c and d, samples c and d, updates c and d, and then prints the values of c and d. The values of c and d when the behavior expires are the same as the values that were printed. The standard output is printed below:

display: $a = 1 \quad b = 0$
monitor: $c = 0 \quad d = 1$

```
initial begin: execute_display
  a = 1;
  b = 0;
  a <= b;
  b <= a;
  $display ("display: a = %b b = %b", a, b);
end
```

```
initial begin: execute_monitor
  c = 1;
  d = 0;
  c <= d;
  d <= c;
  $monitor ("monitor: c = %b d = %b", c, d);
end
```

G.8 wait Statement

The wait construct suspends a thread of an activity flow within a behavior until an expression evaluates true.

Example

The assignment of *register_b* to *register_a* below is suspended until *enable* is asserted true. After the assignment is made activity is again suspended for 10 time steps before assigning *register_d* to *register_c*.

```
wait (enable) register_a = register_b;
#10 register_c = register_d;
```

End of Example

G.9 fork . . . join Statement

The ***fork*** ..***join*** construct within a behavior branches an activity flow into multiple parallel threads, each of which may be a ***begin*** . . . ***end*** block statement. The statements within a ***begin*** . . . ***end*** block statements execute in the ordinary way (i.e., sequentially). ***fork*** . . . ***join*** statements are helpful in modeling complex waveforms in testbenches and abstract (and nonsynthesizable) models of behavior. The activity of a ***fork*** . . . ***join*** statement is complete when all of the statements within it have completed execution.

Example

The assignment to A below is made at $t_{sim} = 5$, and the assignment to C is made at $t_{sim} = 10$;

```
fork
 #5 A = B;
 #10 C = D;
join
```

End of Example

G.10 Named (Abstract) Events

The Verilog *named event* provides a high-level mechanism of communication and synchronization within and between modules. A named event, sometimes referred to as an abstract event, is an abstraction that provides interprocess communication without requiring details about physical implementation. In the early stages of design, this can free the designer from having to pass signals between modules explicitly through their ports. A named event can be declared only in a module; it can then be referenced

within that module directly, or in other modules by hierarchically dereferencing the name of the event. The occurrence of the event itself is determined explicitly by a procedural statement using the *event-trigger* operator, −>.

Example

In the description below, the abstract event *up_edge* is triggered when *clock* has a positive edge transition. A second behavior detects the event of *up_edge* and assigns value to the flip-flop's output, subject to an asynchronous *reset* signal. Hierarchical referencing allows modules to communicate between any locations in a design hierarchy, without requiring structural detail [1].

```
module Flop_event (input clock, reset, data, output reg q, output q_bar);
  event              up_edge;
  assign q_bar = ~q;
  always @ (posedge clock) -> up_edge;
  always @ (up_edge, negedge reset)
    begin
    if (reset == 0) q <= 0; else q <= data;
    end
endmodule
```

End of Example

G.11 Constructs Supported by Synthesis Tools

Synthesis tools support a limited subset of the Verilog language. It is essential that models use only supported constructs. Otherwise a synthesis tool will report an error and fail to synthesize a circuit. Table G-2 lists language constructs that are commonly supported by synthesis tools; Table G-3 lists constructs that are to be avoided. Not all of these are inherently unsynthesizable (e.g., ***repeat*** loops), but they are not supported by vendors because equivalent functionality can be described by other constructs. Verilog has robust delay constructs for modeling inertial delays of primitives, and transport delays of nets, and pin-pin path delays of modules (see Appendix F), but technology-dependent attributes, such as propagation delays, are not to be included in modes for synthesis. The rule is to model only the functionality of the circuit, not its timing. The synthesis tools will implement a design subject to constraints on area, timing, and availability of parts in a cell library or speed grade in an FPGA. For additional details, see [1].

REFERENCES

1. Ciletti MD. *Modeling, Synthesis, and Rapid Prototyping with the Verilog HDL*. Upper Saddle River, NJ: Prentice-Hall, 1999.

TABLE G-2

Module declaration
Port modes: **input, output, inout**
Port binding by name
Port binding by position
Parameter declaration
Connectivity nets: **wire, tri, wand, wor, supply0, supply1**
Register variables: **reg, integer**
Integer types in binary, decimal, octal, hex formats
Scalar and vector nets
Subrange of vector nets on RHS of assignment
Module and macromodule instantiation
Primitive instantiation
Continuous assignments
Shift operator
Conditional operator
Concatenation operator (including nesting)
Arithmetic, bitwise, reduction, logical and relational operators
Procedural block statements (**begin** ... **end**)
case, casex, casez, default
Branching: **if, if** ... **else, if** ... **else** ... **if**
disable (of procedural block)
for loops
Tasks: **task** ... **endtask** (no timing or event control)
Functions: **function** ... **endfunction**

TABLE G-3

Assignment with variable used as bit select on LHS
global variables
case quality, inequality (===, !==)
defparam
event
fork ... **join**
forever
while
wait
initial
pulldown, pullup
force ... **release**
repeat
cmos, rcmos, nmos, rnmos, pmos, rpmos
tran, tranif0, tranif1, rtran, rtranif0, rtranif1
primitive ... **endprimitive**
table ... **endtable**
intra-assignment timing control
delay specifications
scalared, vectored
small, medium, large
specify, endspecify
$time
weak0, weak1, strong0, strong1, pull0, pull1
$keyword

APPENDIX H

Flip-Flop and Latch Types

Some of the examples in the text use a variety of flip-flop and latches from a standard-cell library. The main functional features of those flip-flops are summarized in Table H-1.

TABLE H-1

Symbol	Description
CK D G RB Q *dffrgpqb_a*	The *dffrgpqb_a* is a D-type flip-flop with rising-edge *clock* (CK), internally gated data input (between the output and the external datapath and the output), (D) asynchronous active-low gate control (G), reset (RB), and Q output.
CK D0 D1 RB SL Q *dffrmpqb_a*	The *dffrmpqb_a* is a D-type flip-flop with rising-edge clock (CK), dual, internally multiplexed data inputs, ($D0$ and $D1$), asynchronous active-low reset (RB), data select (SL), and Q output.
CK D SB Q *dffspqb_a*	The *dffspqb_a* is a D-type flip-flop with rising-edge clock (CK), asynchronous active-low set (SB), and Q output.
CK D RB Q QB *dffrpb_a*	The *dffrpb_a* is a D-type flip-flop with rising-edge clock (CK), asynchronous active-low reset (RB), Q and QB outputs.
CK D RB Q *dffrpqb_a*	The *dffrpqb_a* is a D-type flip-flop with rising-edge clock (CK), asynchronous active-low reset (RB), and Q output.
D GB RB Q *latrpqb_a*	The *latrpqb_a* is a D-type transparent latch with active-low latch enable (GB), active-low reset (RB), and Q output.
D G RB Q QB *latrnb_a*	The *latrnb_a* is a D-type transparent latch with active-high latch enable (G), active-low reset (RB), Q and QB output.
D G RB Q *latrnqb_a*	The *latrnqb_a* is a D-type transparent latch with active-high latch enable (G), active-low reset (RB), and Q output.

APPENDIX I

Verilog-2001, 2005

The Verilog HDL underwent its first revision in 2000, and emerged as IEEE Std. 1364-2001, known as Verilog-2001, with significant changes aimed at improving the utility and clarity of the language. The Verilog Standards Committee clarified ambiguous syntax and semantics in IEEE Std. 1364-1995[1] and removed errors in the LRM. The language was enhanced to support higher level modeling and abstract modeling, while maintaining backward compatibility with IEEE 1364-1995. Minor revisions and clarifications were again introduced in 2005. We will discuss a selected set of changes here. For additional topics, see [1, 2]. In this appendix, we will refer to the revised language as Verilog 2001, 2005.

I.1 ANSI C Style Changes

IEEE Std. 1364-2001 introduced ANSI C style syntax for module and UDP declarations.

I.1.1 Module Port Mode and Type Declarations

Verilog-2001, 2005 allows the mode and type of a port to be combined in a single declaration, as shown in Figure I-1. Also, the input ports have default type ***wire***, so the declaration of type ***wire*** for the input ports may be omitted to further simplify the description. The optional description in Figure I-2 places the declaration of the mode, type and vector range of the signals in the port.

I.1.2 Module Declarations

See Figure I-2.

I.1.3 Module Port Parameter List

Parameters are declared as module items in Verilog-IEEE 1364 (i.e. within the body of the module's declaration). In Verilog-2001, 2005, the declarations of parameters may be included between the module name and the port list, as shown in Figure I-3.

[1]Referred to as Verilog-1995.

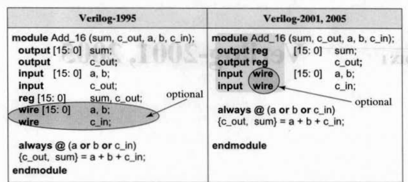


FIGURE I-1

Verilog-1995

```
module Add_16 (sum, c_out, a, b, c_in);
  output [15: 0]  sum;
  output          c_out;
  input  [15: 0]  a, b;
  input           c_in;
  reg [15: 0]     sum;
  reg             c_out;
  wire [15: 0]    a, b;
  wire            c_in;

  always @ (a or b or c_in)
  {c_out, sum} = a + b + c_in;
endmodule
```

Verilog-2001, 2005

```
module Add_16 (
output reg   [15: 0]  sum,
output reg            c_out,
input        [15: 0]  a, b,
input                 c_in);

always @ (a or b or c_in)
{c_out, sum} = a + b + c_in;

endmodule
```

FIGURE I-2

Verilog-1995

```
module Add (sum, c_out, a, b, c_in);
  parameter                  size = 16;
  output   [size -1: 0]      sum;
  output                     c_out;
  input    [size -1: 0]      a, b;
  input                      c_in;
  reg      [size -1: 0]      sum;
  reg                        c_out;

  always @ (a or b or c_in)
  {c_out, sum} = a + b + c_in;
endmodule
```

Verilog-2001, 2005

```
module Add #(parameter size = 16) (
  output  reg   [size -1: 0]   sum,
  output  reg                  c_out,
  input         [size -1: 0]   a, b,
  input                        c_in);

  always @ (a or b or c_in)
  {c_out, sum} = a + b + c_in;

endmodule
```

FIGURE I-3

I.1.4 UDP Declarations

Verilog-2001, 2005 allows ANSI-style declarations combining the mode and/or data type of the elements of a port with the port list (see Figure I-4).

I.1.5 Declarations of Functions and Tasks

The syntax of Verilog-1995 for declaring functions and tasks separates the arguments of a function from its name and associates inputs and outputs with their order in separately made declarations. Verilog-2001, 2005 adopts an ANSI C style that associates the arguments with the name, using the same syntax as that for modules. Examples of the new syntax for functions and tasks are shown in Figure I-5.

Verilog 1995	Verilog 2001, 2005
primitive latch (q_out, enable, data); **output** q_out; **input** enable, data; **reg** q_out; **table** ... **endtable** **endprimitive**	**primitive** latch (**output reg** q_out, **input** enable, **input** data); **table** ... **endtable** **endprimitive**

FIGURE I-4

Verilog 1995	Verilog 2001, 2005
function [16: 0] sum_FA; **input** [16: 0] a, b; **input** c_in; sum = a + b + c_in; **endfunction**	**function** [16: 0] sum_FA (**input** [15:0] a, b, **input** c_in); sum = a + b + c_in; **endfunction**

(a)

Verilog 1995	Verilog 2001, 2005
task sum_FA; **output** [16: 0] sum_FA; **input** [15: 0] a, b; **input** c_in; sum = a + b + c_in; **endtask**	**task** sum_FA (**output** [16: 0] sum, **input** [15:0] a, b, **input** c_in); sum = a + b + c_in; **endtask**

(b)

FIGURE I-5

Verilog 1995	Verilog 2001, 2005

function real sum_Real;
  input real a, b;
 	
sum = a + b;
 	
endfunction
	function real sum_Real (**input real** a, b);
	  sum = a + b;
	
	endfunction

(a)

Verilog 1995	Verilog 2001, 2005

task sum_Real;
  output real sum;
  input real a, b;
 	
sum = a + b;
 	
endtask
	task sum_Real
	  (**output real** sum, **input real** a, b);
	
	sum = a + b;
	
	endtask

(b)

FIGURE I-6

The type of the inputs and outputs of a function or task is ***reg***, unless specified by a declaration within the function of task. Verilog-2001, 2005 allows the type to be declared within the port of the function or task, as shown in Figure I-6.

I.1.6 Initialization of Variables

Variables of type ***reg, integer*** and ***time*** are initialized to a default value[2] of x in the first cycle of simulation. A **wire** is initialized to z and inherits the value of its driver. Variables of type ***real*** and ***realtime*** are initialized to the default value 0.0. In Verilog-1995, a separate declaration can declare an initial value for a ***reg, integer***, or ***time*** variable. Verilog-2001, 2005 combines the declaration of an initial value with the declaration of the type of a ***reg, integer, time, real***, or ***realtime*** variable that is declared at the module level (i.e., a variable declared elsewhere, such as in a task, may not be declared to have an initial value). The value of a ***wire*** remains at the value of its default until the ***wire*** is driven to a different value in simulation. An example of initialization of a variable is given in Figure I-7. A wire may inherit an initial value from a continuous assignment.

An initial value may be assigned to a variable as part of an ANSI C style of a port declaration, as shown in Figure I-8.

I.2 Code Management

Verilog-2001, 2005 expands the capabilities of tasks and functions to include re-entrant tasks and recursive functions.

[2]The default net type can be overridden by a compiler directive

Verilog 1995	Verilog 2001, 2005
module Clk_gen (clock); **parameter** delay = 5; **output** clock; **reg** clock; **initial begin** clock = 0; **forever** #delay clock = ~clock; **end** **endmodule**	**module** Clk_Gen **#(parameter** delay = 5) **(output** clock); **reg** clock = 0; **initial forever** #delay clock = ~clock; **endmodule**

FIGURE I-7

Verilog 1995	Verilog 2001, 2005
module Clk_gen (clock); **parameter** delay = 5; **output** clock; **reg** clock; **initial begin** clock = 0; **forever** #delay clock = ~clock; **end** **endmodule**	**module** Clk_Gen **#(parameter** delay = 5) **(output reg** clock = 0); **initial forever** #delay clock = ~clock; **endmodule**

FIGURE I-8

I.2.1 Re-Entrant Tasks

Tasks in Verilog-1995 are allocated static memory that persists for the duration of simulation. The memory space of a task is shared by all calls to the task. Task variables retain their value between calls. Tasks may be called from multiple concurrent behaviors, setting up the possibility that data may be overwritten and compromised before a given call to a task is complete. Designers work around this problem by placing the same task in multiple modules and isolating their memory space, but this wastes resources and complicates maintenance of the code [1, 2].

Verilog-2001, 2005 supports *re-entrant tasks* with dynamic allocation and deallocation of memory during simulation, each time a task is called. The keyword **automatic** designates a task with dynamic memory allocation. Such tasks are not static, and their allocated memory is not shared. Because the memory allocated for an automatic task is released when the task completes execution, models that use such

tasks must not reference data generated by the task after the task exits. This imposes restrictions on the style of code that may use automatic tasks [1, 2].

I.2.2 Recursive Functions

Functions in Verilog-1995 are static too, and may not include delay constructs (i.e., #, @, ***wait***). Functions effectively implement combinational logic equivalent to an expression. Because a function executes in zero time, there is no possibility for concurrent calls to the same function. However, subsequent calls to a function overwrite its memory space. If a function calls itself recursively, each call will overwrite the memory of the previous call. In Verilog-2001, 2005 a function can be declared ***automatic***, which causes distinct memory to be allocated each time a function is called. The memory is released when the function exits. The classical example of recursion in Figure I-9 compares a recursive implementation in Verilog-2001, 2005 with an illegal description in Verilog-1995.

I.2.3 Constant Functions

Functions may be used in Verilog-1995 only where a nonconstant expression can be used. For example, the widths and depths of arrays can be hardwired by fixed numbers defined by parameters, which are constants. Although parameters can be defined in terms of other parameters, this mechanism for scaling a design can be cumbersome.

Verilog-2001, 2005 supports *constant functions*, which can be called wherever a constant is required. Constant functions are evaluated at elaboration time and do not depend on the values of variables at simulation run-time. Only constant expressions may be passed to a constant function, not the value of a net or register variable. Consequently, a constant function may reference only parameters, localparams, locally declared variables, and other constant functions. The parameters that are used by a function must be declared before the function is called, and the memory used by a function is released after the function has been elaborated.

Avoid using ***defparam*** statements to redefine the parameters within a function, because the value returned may differ between simulators. The parameters within an instance of a module can be redefined unambiguously by the # construct. Constant functions may not call system functions and tasks, and may not use hierarchical path references.

Verilog-1995	Verilog-2001, 2005
function [63: 0] Bogus input [31: 0] N; if (N == 1) Bogus = 1; else Bogus = N*Bogus (N–1); endfunction	function automatic [63: 0] factorial input [31: 0] N; if (N == 1) factorial = 1; else factorial = N*factorial (N–1); endfunction

FIGURE I-9

I.3 Support for Logic Modeling

I.3.1 Implicit Nets

In Verilog-1995, an undeclared identifier will be an implicit net data type if it (1) appears in the port of an instantiated module, (2) is connected to an instance of a primitive, or (3) appears on the LHS of a continuous-assignment statement and is also declared as a port of the module containing the assignment. If the implicit net is connected to a vector port, it will inherit the size of the port; otherwise, it will be a scalar net. The default data type of an implicit net is ***wire***, which can be modified by a compiler directive. If the type of the LHS of a continuous assignment is not declared explicitly and is not determined implicitly by the above rules, an error will result. In Verilog-2001, 2005 an undeclared identifier that is not a port of a module will be inferred as an implicit scalar net. Figure I-10 shows a module that complies with Verilog-2001, 2005, but not with Verilog-1995.

I.3.2 Disabled Implicit Nets

The mechanism for implicitly declaring a net can be disabled by including a new argument, ***none*** with the ***'default_nettype,*** compiler directive. This argument requires all nets to be explicitly declared. Disabling the default assignment of type to identifiers will reveal compilation errors that arise from misspelled identifiers but that would be otherwise undetected.

I.3.3 Variable Part Selects

Verilog-1995 allows a part select of contiguous bits from a vector if the range indexes of the part select are constant. Verilog-2001, 2005 provides two new part-select operators to support a variable part select of fixed width, +: and −:, having the syntax <starting_bit> +: <width> and <starting_bit> −: <width>, respectively. The parameter width specifies the size of the part select, and *start_bit* specifies the leftmost or rightmost bit in the vector from which the part select is taken, depending on whether the selection will be made by

Verilog 1995	Verilog 2001, 2005
`module Adder (sum, c_out, a, b, c_in);`	`module Adder (`
`output [15:0] sum;`	`output [15: 0] sum,`
`output c_out;`	`output c_out,`
`input [15:0] a, b;`	`input [15: 0] a, b,`
`input c_in;`	`input c_in);`
`always @ (a or b or c_in)`	`always @ (a or b or c_in)`
`{c_out, sum} = a + b + c_in;`	`{c_out, sum} = a + b + c_in;`
`assign match = a & b; // Invalid`	`assign match = a & b // Valid`
`endmodule`	`endmodule`

FIGURE I-10

Verilog 1995	Verilog 2001, 2005
reg [15: 0] sum; **reg** [2: 0] K;	**reg** [15: 0] sum; **reg** [2: 0] K;
// Valid; **wire** [7: 0] a_byte = sum [15: 8];	// Valid; **wire** [7: 0] a_byte = sum[15: 8];
// Error: **wire** [3: 0] b_byte = sum[K + 3: K];	// Valid: **wire** [3: 0] b_byte = sum[K +: 3];

FIGURE I-11

Verilog 2001, 2005					
wire	[31: 0]	d_paths	[15: 0];		// 1-dimensional array of words
reg	[15: 0]	data	[0: 127] [0: 127];		// 2-dimensional array of words
real		time_array	[0: 15] [0: 15] [0: 15];		// 3-dimensional array

FIGURE I-12

incrementing or decrementing the index of the bits in the vector. See Figure I-11 for an example.

I.3.4 Arrays

Verilog-1995 supports only one-dimensional arrays of type ***reg***, ***integer***, and ***time***. Verilog-2001, 2005 supports arrays of ***real*** and ***realtime*** variables, in addition to arrays of type ***reg***, ***integer***, and ***time***. Arrays can be of any number of dimensions in Verilog-2001, 2005. The range specifications for the indexes of the dimensions of an array follow the declared name of the array. Examples are shown in Figure I-12.

A bit or a part select of a word in an array cannot be selected directly in Verilog-1995, but Verilog-2001, 2005 allows selection of a bit or a part select from an array of any number of dimensions. To select a word, reference the array with an index for each dimension. To select a bit or a part, reference the array with an index for each dimension plus a bit or range specification. See Figure I-13.

I.4 Support for Arithmetic

I.4.1 Signed Data Types

Verilog-1995 is limited to signed arithmetic on 32-bit integers. The ***reg***, ***time***, and all net data types are unsigned, and expressions are evaluated as signed arithmetic only if every operand is a signed variable (i.e., has type ***integer***). The data types of the variables in an

Verilog 1995

```
reg   [15: 0]   data        [0: 127][0: 127];          // 2-dimensional array of words
real  time_array            [0: 15][0: 15][0: 15];     // 3-dimensional array
wire  [31: 0] d_paths       [15: 0];                   // 1-dimensional array of words

wire  [15: 0] a_data_word = data [4][21];              // references a word
wire  a_time_sample = time_array [7][7][7];            // references a word
wire  [7: 0] a_byte = data [64][32][12: 5];            // references a byte
wire  a_bit = data [31][8][3];                         // references a bit
```

FIGURE I-13

expression, not the operators, determine whether signed or unsigned arithmetic is performed. Verilog-2001, 2005 uses the reserved key word ***signed*** to declared that a ***reg*** or a net type variable is signed, and supports signed arithmetic on vectors of any size, not just 32-bit values. See Figure I-14.

I.4.2 Signed Ports

Ports may be declared to be signed in two ways: by declaration with the mode of the port or by declaration of the type of the associated port variable. Figure I-15 shows examples of declarations of signed variables without and with ANSI C style syntax.

I.4.3 Signed Literal Integers

Verilog-1995 represents literal integers in three ways: a number (e.g., −10), an unsized radix-specified number (e.g., 'hA), and a sized radix-specified number (e.g., 64'hF). If a

Verilog 1995	Verilog 2001, 2005

<pre>integer   m, n;
reg   [63: 0]   v;
 	
...   // value stored
m = 12; // 0000_..._0000_1100
n = –4; // 1111_..._1111_1100
v = 8; // 0000_..._0000_1000
 	
m = m / n; // result –3
 	
v = v / n; // result 0</pre>
	<pre>integer m, n;
	reg signed [63: 0] v;
	
	... // value stored
	m = 12; // 0000_..._0000_1100
	n = –4; // 1111_..._1111_1100
	v = 8; // 0000_..._0000_1000
	
	m = m / n; // result –3
	
	v = v / n; // result –2</pre>

FIGURE I-14

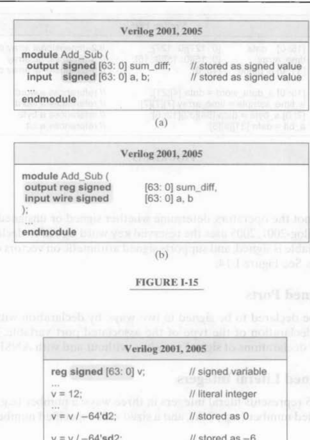


FIGURE I-15

FIGURE I-16

radix is specified the number is interpreted as an unsigned value; if a radix is omitted, the number is interpreted as a signed value. In Verilog-2001, 2005, a sized literal integer can be declared as an integer. The symbol ***s*** is used to specify that a sized or unsized literal integer is signed, as illustrated in Figure I-16.

I.4.4 Signed Functions

Functions in Verilog-1995 may be called any place that an expression can be used. The value returned by a function is signed if and only if the function is declared to be an integer. With the reserved keyword ***signed***, Verilog-2001, 2005 allows signed arithmetic to be performed on returned values of a vector size. Figure I-17 identifies the possible

Example	Returned Value	Verilog 1995	Verilog 2001, 2005
function sum	single bit	x	x
function [31: 0] sum	unsigned vector of 32 bits	x	x
function integer sum	signed vector of 32 bits	x	x
function real sum	64-bit double-precision	x	x
function time sum	unsigned 64-bit vector	x	x
function signed [63: 0] sum	signed 64-bit vector		x

FIGURE I-17

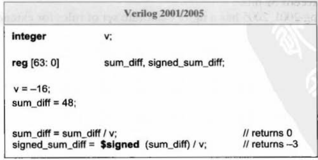
types of a function in Verilog-1995 and Verilog-2001, 2005. Remember that the data types of the variables in an expression, not the operators, determine whether signed or unsigned arithmetic is performed. Signed arithmetic is performed only when all of the operands are signed variables.

I.4.5 System Functions for Sign Conversion

Verilog-2001, 2005 provides two new system functions for converting values to signed or unsigned values. The function **$signed** returns a signed value from the value passed in. The function **$unsigned** returns an unsigned value from the value passed in. The functions are useful because an expression returns a signed value if and only if all of its operands are signed variables. Sign conversion eliminates the need to declare and assign value to additional variables to circumvent the restrictions of Verilog-1995. See Figure I-18.

I.4.6 Arithmetic Shift Operator

Logical shift operators (<<, >>) are supported by Verilog-1995. These operators have two operands: an expression (operand) whose value is to be shifted, and an expression



Verilog 2001/2005

```
integer         v;

reg [63: 0]     sum_diff, signed_sum_diff;

v = –16;
sum_diff = 48;

sum_diff = sum_diff / v;                            // returns 0
signed_sum_diff = $signed (sum_diff) / v;           // returns –3
```

FIGURE I-18

Verilog 2001, 2005

```
integer data_value, data_value_1995, data_value_2001; // signed datatype
...
data_value = -9;                           // stored as 1111..._1111_0111
...
data_value_1995 = data_value >> 3;         // stored as 0001..._1111_1110
data_value_2001 = data_value >>> 3;        // stored as 1111..._1111_1110
data_value_1995 = data_value << 3;         // stored as 1111..._1011_1000
data_value_2001 = data_value <<< 3;        // stored as 1111..._1011_1000
```

FIGURE I-19

(operand) that determines the number of shifts. The bitwise-shift operators insert a 0 into the cell vacated by a shift. Verilog-2001, 2005 includes the operators >>> to shift arithmetically to the right and <<< to shift bitwise to the left. The arithmetic right-shift operator (>>>) implemented in Verilog 2001, 2005 inserts the MSB (sign bit) back into the MSB at each shift if the expression that determines the value of the shifted word is signed; otherwise it will insert a 0. The expression forming the second operand of the shift operator may be signed or unsigned; all other expressions are interpreted to be signed if and only if every operand is signed. The arithmetic left-shift operator (<<<) is functionally equivalent to the logical left-shift operator (<<). The examples in Figure I-19 show the distinctions between the logical and arithmetic shift operators.[3]

I.4.7 Assignment Width Extension

Verilog-1995 has two rules for extending the bits of a word when the expression on the RHS side of an assignment statement has a smaller width than the expression on the LHS. If the expression on the RHS is signed, the sign-bit determines the extension to fill the LHS. If the expression on the RHS is unsigned (i.e., ***reg***, ***time***, and all net types), its extension is formed by filling with 0. This can lead to inappropriate extensions when the LHS exceeds 32 bits.

Verilog-2001, 2005 has a more elaborate set of rules for extending the width of a word beyond 32 bits, as summarized in Figure I-20. These rules differ from those for Verilog-1995, so a model that adhered to the rules of Verilog-1995 will not work the same as a model employing the rules of Verilog-2001, 2005.

I.4.8 Exponentiation

Exponentiation is not implemented conveniently in Verilog-1995—it requires repeated multiplication within a loop statement. Verilog-2001, 2005 includes the operator **, which

[3]The shift-left operator implemented in Verilog-2001 fills with 0, which corresponds to multiplication by a power of 2. An arithmetic shift-left that fills with the LSB produces different results.

Left-most bit of RHS expression	Extended value	
	Unsigned RHS expression	Signed RHS expression
0	0	0
1	0	1
x	*x*	*x*
z	*z*	*z*

FIGURE I-20

Verilog 2001, 2005		
Returned value	Base	Exponent
double-precision floating point	real, integer, or signed value	real, integer, or signed value
ambiguous	0	not a positive number
ambiguous	negative number	not an integer

FIGURE I-21

Verilog 2001, 2005

```
reg [7: 0] base;
reg [2: 0] exponent;
reg [15: 0] value;

value = base ** exponent;
```

FIGURE I-22

implements exponentiation directly. The operator has two operands: a base and an exponent. The type of the returned value depends on the operands, as shown in Figure I-21. The example in Figure I-22 illustrates the syntax for using **. (*Note*: The operator for exponentiation has higher precedence than the operator for multiplication.)

I.5 Sensitivity List for Event Control

The event-control expression in Verilog-1995 uses the ***or*** operator to compose an expression that is sensitive to multiple variables. Verilog-2001, 2005 allows a comma-separated list, as shown by the example in Figure I-23.[4]

I.6 Sensitivity List for Combinational Logic

Level-sensitive cyclic behaviors (***always***) will simulate and synthesize combinational logic if the event-control expression of the behavior is complete (i.e., it contains every signal that is referenced implicitly or explicitly in the behavior), and if every variable is assigned value in every execution thread of the behavior (e.g., **if** statements are complete). If the event-control expression is incomplete, a synthesis tool will infer latched logic, rather than combinational logic. Unintentional omission of a signal from an event-control expression is problematic, so Verilog-2001, 2005 uses a wildcard token (*) to indicate a level-sensitive event-control expression that is sensitive to every variable that is reference within the behavior, thereby eliminating the need to explicitly identify them and avoiding the consequences of an incomplete event-control expression. See Figure I-24.

Verilog-1995	Verilog-2001, 2005
module Adder (sum, c_out, a, b, c_in); **output** [15: 0] sum; **output** c_out; **input** [15: 0] a, b; **input** c_in; **always @** (a **or** b **or** c_in) {c_out, sum} = a + b + c_in; **endmodule**	**module** Adder (sum, c_out, a, b, c_in); **output** [15: 0] sum; **output** c_out; **input** [15: 0] a, b; **input** c_in; **always @**(a, b, c_in) {c_out, sum} = a + b + c_in; **endmodule**

FIGURE I-23

[4]The new syntax permits mixing separated by ***or*** with items separated by commas, but this usage makes the code less readable.

Verilog-1995	Verilog-2001, 2005
module Adder (sum, c_out, a, b, c_in); **output** [15: 0] sum; **output** c_out; **input** [15: 0] a, b; **input** c_in; **reg** [15: 0] sum; **reg** c_out; **always @** (a **or** b **or** c_in) {c_out, sum} = a + b + c_in; **endmodule**	**module** Adder (**output reg** [15: 0] sum; **output reg** c_out; **input** [15: 0] a, b; **input** c_in); **always @** (*) // alternative: **always @ *** {c_out, sum} = a + b + c_in; **endmodule**

FIGURE I-24

Caution: The @ operator is associated with the single statement or ***begin*** ... ***end*** block of statements that immediately follow it. Careless use of the operator will lead to models that do not represent the functionality of combinational logic, and that do not synthesize into combinational logic. In Figure I-25, the cyclic behavior within *Bogus* misuses the event control operator @, making the behavior sensitive to only *a* and *b*, but not to *c_in*.

Verilog-1995	Verilog-2001, 2005
module Adder (sum, diff, c_out, a, b, c_in); **output** [15: 0] sum, diff; **output** c_out **input** [15: 0] a, b; **input** c_in; **reg** [15: 0] sum, diff; **reg** c_out; **always @** (a **or** b **or** c_in) **begin** {c_out, sum} = a + b + c_in; diff = a – b; **end** **endmodule**	**module** Adder (**output reg** [15: 0] sum, diff, **output reg** c_out **input** [15: 0] a, b, **input** c_in); sensitive to only a and b **always begin** @ diff = a – b; {c_out, sum} = a + b + c_in; **end** **endmodule**

FIGURE I-25

I.7 Parameters

Parameters make models more configurable, readable, extendable, and portable. Declared by the keyword ***parameter***, parameters in Verilog-1995 are run-time constants, and their value can be changed before simulation and during elaboration. There are two mechanisms for redefining the value of a parameter: remotely, using the keyword ***defparam***, and implicitly, by *in-line redefinition*. The declaration redefining a parameter with the ***defparam*** keyword can be placed anywhere in the design hierarchy, and it redefines the value of a parameter at any location in the design hierarchy via hierarchical dereferencing of path names. This poses the risk that parameters can be changed inadvertently from any location in the design, since parameters are not fixed constants. In-line redefinition requires that text adhering to the syntax *#(value_1, value_2, . . . , value_m)* be inserted after the instance name of a module to redefine parameters declared within the module. The order of the sequence of *value_1, value_2*... must correspond to the order of the sequence in which the parameters are declared within the module. This is cumbersome when the modules contain several parameters, not all of which are to be redefined. Because the parameters are not explicitly named in this syntax for redefinition, the practice is prone to error, and renders the model less readable. Verilog-1995 also supports specparams (*specify parameters*) that may be declared and used only within ***specify***... ***endspecify*** blocks[5] within a module. A ***specparam*** is local to the block in which it is declared, and may be used only within the block. A standard delay format (SDF) file can redefine the value of a ***specparam***. The risk, again, is that specparams could be mistakenly redefined.

A parameter inherits its size and type from the final value assigned to it during elaboration, before simulation, which need not be the same type that was assigned to the parameter when it was declared in its parent module. A parameter can be an unsized integer (at least 32 bits), a sized and unsigned integer, a real (floating point) number, or a string. Other parameters can be operands in the expression that declares the value of a parameter. Thus, in Verilog-1995 the size and type of a parameter can be changed when the parameter is redefined, which could produce undesirable side effects, because the operations performed in an expression depend on the size and type of its operands. Figure I-26 displays the rules used by Verilog-1995 to determine the arithmetic performed by the operands in an expression.

[5]Specify blocks are used to declare input–output paths across a module, assign delay to those paths, and declare time checks to be performed on signals at the module inputs.

PDG

Verilog 1995	
Operands	Operation
All operands are signed integers	Signed arithmetic
An operand is unsigned	Unsigned integer arithmetic
At least one operand is a real value	Floating point arithmetic

FIGURE I-26

I.7.1 Parameter Constants

Verilog-2001, 2005 provides explicit definition of the size and data type of a parameter. Figure I-27 shows the rules for determining how the size and type of a parameter are redefined in Verilog-2001, 2005 by an expression having arithmetic operators. When the sign, size, or type of a parameter is explicitly declared it cannot be overridden by a subsequent parameter value redefinition.

I.7.2 Parameter Redefinition

Verilog-2001, 2005 provides *explicit in-line redefinition* of parameters on a module-instance basis. The syntax for redefining the parameters of an instantiated module is shown below.

```
module_name instance_name #(.parameter_name (parameter_value), ...)
  (port_connections);
```

This feature of Verilog-2001, 2005 explicitly identifies the redefined parameters. The redefinition does not depend on the order in which the parameters are defined in the associated module. Consequently, the code is self-documenting and more readable than its counterpart in Verilog-1995.

Verilog 2001, 2005					
Specified by declaration			Subject to redefinition	Redefinition rule	
Sign	Range	Type			
No	No	No	Yes	Same as Verilog-1995[1]	
Yes	Yes	No	No	Parameter is signed	Size is specified by the range
Yes	No	No	Inherits size	Parameter is signed	Size is inherited from last redefinition
No	Yes	No	No	Parameter is unsigned	Size is fixed by range
		Yes	Yes	Parameter retains type	

[1]In Verilog-1995 a parameter inherits the vector size and type of the last parameter redefinition.

FIGURE I-27

I.7.3 Local Parameters

Verilog-2001, 2005 introduces local parameters (keyword: ***localparam***), whose value cannot be directly redefined from outside the module in which they are declared.[6] Figure I-28 compares the parameters, specify parameters, and local parameters. Although a ***localparam*** cannot be directly redefined, it can be indirectly redefined by assigning it the value of a ***parameter***, which can be changed by the methods described above.

I.8 Instance Generation

Verilog-1995 supports structural modeling with declarations of arrays of instances of primitives and modules. The ***generate*** ... **endgenerate** construct in Verilog-2001, 2005 extends this feature to replicate distinct copies of net declarations, register variable declarations, parameter redefinitions, continuous assignments, ***always*** behaviors,

[6]It might be desirable to protect, for example, the state-assignment codes from inadvertent change.

		Verilog 2001, 2005		
		Verilog 1995		
		parameter	specparam	localparam
Location of declaration	Module item	Yes	No	Yes
	Task item	Yes	No	Yes
	Function item	Yes	No	Yes
	Specify block	No	Yes	No
Method of direct redefinition	By a defparam	Yes	No	No
	Redefined in-line	Yes	No	No
	SDF files	No	Yes	No
Method of indirect redefinition by assignment of value	From another parameter	Yes	Yes	Yes
	From a localparam	Yes	Yes	No
	From a specparam	No	Yes	No
Allowed reference	Within a module	Yes	No	Yes
	Within a specify block	No	Yes	No

FIGURE I-28

initial behaviors, tasks, and functions.[7] Verilog-2001, 2005 introduces a new kind of variable, denoted by the keyword ***genvar***, which declares a nonnegative integer[8] that is used as an index in the replicating ***for*** loop associated with a ***generate . . . endgenerate*** block. The index of the ***for*** loop of a ***generate . . . endgenerate*** block must be a ***genvar*** variable, and the initializing statement and the loop update statement must both assign value to the same ***genvar*** variable. The contents of the ***for*** loop of a ***generate . . . endgenerate*** statement must be within a named ***begin . . . end*** block. Note: *The name of the block is used to build a unique name for each generated item.*

---

[7]A ***generate . . . endgenerate*** block may not include port declarations, constant declarations, and specify blocks.

[8]A variable of type ***genvar*** may be declared within a module or within the ***generate . . . endgenerate*** block; it may not be assigned a negative value, an x value, or a z value.

```
Verilog 2001, 2005

module Adder_CLA (parameter size = 32)(
 output [size – 1: 0]     sum,
 output                   c_out,
 input [size – 1: 0]      a, b,
 input                    c_in);

 wire   [size/8 – 1: 0]   c_o, c_i;
 assign                   c_i[0] = c_in;
 assign                   c_out = c_o[size/8 – 1];

 generate
  genvar j;
  for (j = 1; j <= 3; j = j + 1) begin: j
   assign c_i[j] = c_o[j – 1];
  end
 endgenerate

 generate
  genvar k;
  for (k = 0; k <= size/8 – 1; k = k + 1) begin: M
    Add_cla_8 ADD (sum[((k+1)*8 – 1) –: 8], c_o[k], a[((k+1)*8 – 1) –:8], b[((k+1)*8 – 1) –:8], c_i[k]);
  end
 endgenerate
endmodule
```

FIGURE I-29

The model in Figure I-29 generates a 32-bit adder from copies of an 8-bit adder, with instance names *M[0].ADD, M[1].ADD, M[2].ADD*, and *M[3].ADD*, A separate generate statement connects the internal carry chain of the adder by generating continuous assignments. Note that the entire model is parameterized, so redefining the value of size to 64 will generate and connect 8 copies of the 8-bit slice adder, *Add_cla_8*. This leads to a more compact description than instantiating 8 individual 8-bit adders. Also, manual replication of structural items does not lead to a parameterized model, which ultimately limits the utility of the model.

In Figure I-30, ***generate*** statements are used to generate a parameterized pipeline of words.

The replication of items by a generate block can be controlled by ***if*** statements and ***case*** statements. Figure I-31 uses an ***if*** statement to determine whether a ripple-carry adder or a look-ahead adder is instantiated, depending on the width of the datapath. Figure I-32 uses a ***case*** statement to determine the instantiation.

Verilog 2001, 2005

```
module generated array_pipeline #(parameter width = 8, length = 16)(
  output [width – 1: 0] data_out,
  input [width – 1: 0] data_in,
  inputclk, reset);

  reg[width – 1: 0] pipe[0: length – 1];
  wire [width – 1: 0] d_in[0: length – 1];

  assign d_in [0] = data_in;
  assign data_out = pipe[size – 1];

  generate
    genvar k;
    for (k = 1; k <= length – 1; k = k + 1) begin: W
      assign d_in[k] = pipe[k – 1];
    end
  endgenerate

  generate
    genvar j;
    for (j = 0; j <= length – 1; j = j + 1) begin: stage
      always @ (posedge clk, negedge reset)
      if (reset == 0) pipe[j] <= 0; else pipe[j] <= d_in[j];
    end
  endgenerate
endmodule
```

FIGURE I-30

Verilog 2001, 2005

```
module Add_RCA_or_CLA#(parameter size = 8)(
  output [size – 1: 0]       sum;
  output                     c_out,
  input   [size – 1: 0]      a, b,
  input                      c_in);

  generate
    if (size < 9) Add_rca #(size) M1 (sum, c_out, a, b, c_in);
    else Add_cla #(size) M1 (sum, c_out, a, b, c_in);
  endgenerate
endmodule
```

FIGURE I-31

Verilog 2001, 2005

```
module Add_RCA_or_CLA (parameter size = 8)(
  output [size – 1: 0]      sum,
  output                    c_out,
  input  [size – 1: 0]      a, b,
  input                     c_in);

  generate
    case (1)
      size < 9:     Add_rca #(size) M1 (sum, c_out, a, b, c_in);
     default:       Add_cla #(size) M1 (sum, c_out, a, b, c_in);
    endcase
  endgenerate
endmodule
```

FIGURE I-32

REFERENCES

1. Sutherland S. *Verilog 2001*. Boston, MA: Kluwer, 2002.
2. *IEEE Standard for Verilog Hardware Description Language 2001*, IEEE Std. 1364–2005. Piscataway, NJ: Institute of Electrical and Electronics Engineers, 2005.

APPENDIX J

Programming Language Interface

The Verilog hardware description language has a built-in programming language interface (PLI) that allows the user to create a "super-Verilog" language with user-defined system tasks that are implemented by the user in the C programming language. These user-defined system tasks are versatile because they are global to the language environment, rather than local to a particular module. This capability greatly expands the utility of the language.

A simulator creates a set of data structures when it compiles a Verilog description. These data structures contain topological and other information about the design. PLI includes a library of C-language functions that can directly access the data structures of a design, giving the user access to a vast amount of information that can support other applications. For example, the data structures that describe the structural connectivity of a description would enable a timing analysis algorithm to enumerate all of the paths from the input ports to the data input of a given flip-flop. The Language Reference Manual lists some of the applications possible with PLI:

- Dynamically scan the data structures of the design and annotate the delays of model instances. (Back-annotation is done after layout to ensure that the models used in timing verification accurately account for the layout-specific parasitic delays induced by the loading of metal interconnect and fanout.)
- Dynamically read test vectors from a file and pass the information to another software tool.
- Create custom graphical environments for user interfaces and displays.
- Create custom debugging environments.
- Decompile the source code to create a Verilog source code description from the data structures of the design.

- Link a C-language simulation model into the design during simulation.
- Interface a hardware unit to the design during simulation.

These are but a few of the uses for PLI. The interested reader is referred to the Language Reference Manual, of which over one-half of the content is dedicated to PLI, and to Sutherland [1], which provides a comprehensive treatment of PLI.

REFERENCE

1. Sutherland S. *The Verilog PLI Handbook*. Boston: Kluwer, 1999.

APPENDIX K

Web sites

Additional resources can be obtained at the various Web sites are listed below. Other sites will be posted our companion Web site.

Industry Organization

www.accellera.org	Accellera
www.opencores.org	Opencores
www.systemc.org	System C

FPGA and Semiconductor Manufacturers

www.actel.com	Actel Corp.
www.altera.com	Altera, Inc.
www.atmel.com	Atmel Corp.
www.latticesemiconductor.com	Lattice Semiconductor Corporation:
www.xilinx.com	Xilinx, Inc.

Media and Archives

www.eetimes.com	EE Times
www.isdmag.com	*Integrated System Design* magazine
http://xup.msu.edu	Xilinx University Resource Center[1]
http://www.mrc.uidaho.edu/vlsi/	See this site for additional links

PDG

EDA Tools, Resources, and Training

www.cadence.com	Cadence Design Systems, Inc.
www.co-design.com	Co-Design Automation, Inc.
www.mentorg.com	Mentor Graphics corp.
www.model.com/verilog	Model Technology
www.simucad.com	Simucad, Inc.
www.silvaco.com	Silvaco, Inc.
www.synopsys.com	Synopsys, Inc.
www.synplicity.com	Synplicity, Inc.
www.tm-associates.com	TM Associates

Consultants

www.sunburst-design.com	Sunburst Design, Inc.
www.sutherland.com	Sutherland HDL, Inc.
www.whdl.com	Willamette HDL, Inc.

[1]The Xilinx University Resource Center Web site, maintained and hosted by the Department of Electrical and Computer Engineering at Michigan State University, provides a collection of resources already located on the web, as well as original content. A robust online support system, consisting of a mailing list, discussion board, and email is in place and monitored to answers any questions that you may have.

APPENDIX L Web-Based Resources

The companion Web site for the book, located at www.prenhall.com, contains tutorials and other resources, including a limited number of additional problems and solutions. A solution manual and classroom slides for the text are available to instructors upon request to the publisher.

Index

$100k, 415
$500K, 10
$bitstoreal, 867
$display, 138, 746, 919, 921
$finish, 125
$hold, 771
$monitor, 138, 746, 919, 921
$period, 774
$pulsewidth, 773
$readmemb, 424, 460
$readmemh, 424
$realtobits, 867
$recovery, 774
$setup, 770
$setuphold, 771
$signed, 740, 882, 937
$skew, 773, 774
$unsigned, 740, 882, 937
$width, 773
$write, 919
2-bit binary comparator, 116
  hierarchical structure of, 118
2s complement
  division, 740
3:8 decoder, 169
3-bit-down counter, 201
4-bit comparator, 117
  block diagram symbol of, 117
  hierarchical structure of, 118
  simulation results for, 118
4-bit serial shift register, 157
  incorrect model of, 158
4-bit shift register, 202
  with parallel load, 203
4-bit universal shift register, 207
8:3 priority encoder, 168
  block diagram and circuit synthesized for, 168
8-bit adder, 946
8-bit barrel shifter, 205
8-bit boundary scan register, 809
8-bit ring counter, 198
8-bit UART transmitter, 381
16-bit, ripple-carry, 112
  design hierarchy, 113
  hierarchical decomposition of, 112
32-bit adder, 946
32-bit comparator, 155
  block diagram symbol for, 155
32-word register file, 208

A

accumulator, 308, 566–569
add-and-shift algorithm, 715
adder
  carry look-ahead, 633–638
  ripple-carry, 632, 946
adder cell, 633
  data input–output relationships, 634
address lines, 418
algorithm, 635
algorithmic state machine (ASM), 4, 82, 236, 335
algorithmic state machine (ASM) charts, 141, 358, 823
  halftone image converter, 536
  up–down counter, 196
  vehicle speed controller, 190
algorithmic state machine and datapath (ASMD), 517
  based sequential binary multiplier
    HDL models, 658
    STG, 658
    verilog modules, 658
algorithmic state machine and datapath (ASMD) charts, 191, 517, 602, 736
  4-bit binary counter, 197
  sequential binary multiplier, 658–669
  serial-to-parallel converter, 604
  up–down counter, 196
algorithms, 515–519
ALU, *See* arithmetic and logic unit (ALU)
always (keyword), 151
American Standard Code for Information Interchange (ASCII), 376
analog-to-digital converter, 556
AND gate, 640
and-or-invert (AOI) circuit, 143
  equivalent circuit modeled by, 145
ANSI C style syntax, 927
AOI circuit, 108
application-specific integrated circuits (ASICs), 81, 103, 141, 415, 515, 749
  based implementation, 515
  chip, 795
  library cells, 14
  market, role of FPGAS , 467–469
  timing violations, elimination of, 766–767
architectural synthesis, 246
archives
  web sites, 951
arithmetic and logic unit (ALU), 104
  RISC stored-program machine, 355
arithmetic operators, 873–875
arithmetic processors, architectures for, 627–741
  addition, functional units for, 632–638
  division, functional units for, 715–742
  fractions, multiplication of, 711–715
  multiplication, 638–710
    ASMD-based sequential binary multiplier, 658–669
    bit-pair encoding, 702–710
    Booth's algorithm sequential multiplier, 687–702
    combinational binary multiplier, 639–642
    efficient STG based sequential binary multiplier, 652–657
    hierarchical decomposition, 644–646
    implicit-state-machine binary multiplier, 675–687
    reduced-register sequential multiplier, 670–675
    sequential binary multiplier, 642–644
    STG-based controller design, 646–652

number representation, 627–632
signed binary numbers, multiplication of, 710–711
subtraction, functional units for, 632–638
arithmetic shift operator, 937–938
arrays, 934
modules, 913
primitives, 913
ASCII, *See* American standard code for information interchange (ASCII)
ASIC, *See* application-specific integrated circuit (ASIC)
ASM, *See* algorithmic-state machine (ASM)
ASMD, *See* algorithmic state machine and datapath (ASMD)
assign (keyword), 144
assign deassign (keywords), 247
assignment width extension, 938
asynchronous arrays, 459
asynchronous input, 211
asynchronous reset signals, 309
asynchronous signals, 208–223
metastability for, 208–213
switch debounce for, 208–213
synchronizers for, 208–213
automatic test pattern generation (ATPG), 786

B

Backus-Naur form (BNF), 885
barrel shifter, 203
BCD, *See* binary-coded decimal (BCD)
BCD-to-excess-3 code converter
synthesis of, 276–280
behavioral algorithms, 154
behavioral description
4-bit shift register synthesis, 202
ring counter synthesis, 199
behavioral modeling, 4
algorithmic state machine (ASM) charts, 187–191
combinational logic, Boolean equation based, 143–146
data types for, 143
decoders, 162–171
encoders, 162–171
multiplexers, 162–171
styles comparison, 154–162
a matter of style, 161–162
algorithm-based models, 160–161
continuous assignment models, 154–156
dataflow/RTL models, 156–160
simulation with behavioral models, 162
behavioral synthesis, 246
parse trees, 247
bidirectional switch, 861
binary counter, 347–353
ASM chart, 350
ASMD chart, 350
for UART_receiver, 391
binary-coded decimal (BCD), 40, 84, 170, 427
excess-3 code converter, 84–89
binary decision diagram (BDD), 21
binary decoder, 65
binary divider, 716
simulation, 721
binary multiplier, 638
BIST, *See* built-in self-test (BIST)
bit lines, 418, 420
bit-pair encoding (BPE), 702–710
rules for, 702
bitwise-and gate, 851
bitwise buffer, 853
bitwise exclusive-nor gate, 853
bitwise exclusive-or gate, 853
bitwise inverter, 854
bitwise-nand gate, 852
bitwise-nor gate, 852
bitwise-or gate, 852
block statement, 152
BNF, *See* backus-naur form (BNF)
Boolean algebra, 16–17
canonic SOP, 24
cube, 17
DeMorgan's laws, 18
essential prime implicant, 32
exclusive-or, with, 36
implicant, 24
irredundant expression, 28
laws of, 17
minimal cover, 33
minimization theorems, 18–21
prime implicant, 31
simplification, 27–28
SOP representation, 23
Boolean equation, 143–146
Boolean function, 24
Boolean logic, 73
Booth recoding, 690–691, 702
Booth's algorithm sequential multiplier, 687–702
ASMD chart, 697
block diagram, 697
Booth's algorithm, 710
boundary scan, 795–796
bypass register (BR) cell for, 798
instruction register (IR) cell for, 799
boundary scan cells (BSCs), 795
BPE, *See* bit-pair encoding (BPE)
bridging faults, 777
bubble sort machine, 548
ASMD chart, 549, 553 block diagram, 549
built-in self-test, 830–844
architecture, 833
built-in verilog operators, 145
bus
unidirectional interface to, 271
bypass register (BR), 798
boundary scan, 798
BYPASS, 800
byte-sequential integrator, 567

C

canonical form, 24, 26
carry look-ahead adder, 633–638
arithmetic implementation of carry bit, 634
carry-in bit, 632
carry-out bit, 632
case (keyword), 163, 252
case-sensitive language, 108
casex statement, 166
child module, 109
circuit's input–output algorithm, 160
circular buffers, 586–589
N-cell, 586
classical design methods, 1
CLBs, *See* configurable logic blocks (CLBs)
clock enable, 313
clock generator, 180–181
clock period, 774
clock skew, 762
effect of, 764
clock tree, 765
CMOS, *See* complementary metaloxide semiconductor (CMOS)
code, 670
code converters, 90
combinational circuit, 638
automatic test pattern generation, 786–788
combinational logic, 143–146, 669
common descriptions of, 148
ROM based implementation, 423
sensitivity List for, 940–942
structural models of, 104
design encapsulation, 104–107
design hierarchy, 111–113
module ports, 108
nested modules, 109–111
some language rules, 108–109
source-code organization, 111–113
structural connectivity, 114
top-down design, 109–111
vectors in verilog, 113
verilog primitives, 104–107
truth table models with verilog, 131
combinational logic circuit, 756
automatic test pattern generation, 786
combinational logic design, 13
ASIC library cells, 13–16
block diagram symbol, 14
Boolean algebra, 16–17
decoder, 64–66
DeMorgan's laws, 18
demultiplexer, 61–62
encoders, 62–63
Glitches, 42
Hazards, 42–55
multiplexer, 60–61
NAND–NOR structures, 55–59
priority decoder, 66–67
priority encoder, 63–64
product-of-sums, 26
representation of, 21–26
sum-of-products, 23–26
combinational logic, synthesis of, 247–260
2-bit comparator circuit, 251
ASIC cells, 258–260 resource sharing, 258–260
continuous assignment, from a, 250
logical don't-care conditions, exploiting, 253–257
priority structures, of, 252–253
combinational (parallel) binary multiplier, 639–642
comparator, 250
complementary metal-oxide semiconductor (CMOS), 74, 128, 418, 776
inverter, 16
doping regions, 16
master–slave circuit of a D-type flip-flop, 75
signal paths, 76

NAND gate, 15
transistor-level schematics, 15
transmission gate, 74
transmission gate, 857
complex programmable logic devices (CPLDs), 416, 465, 795
high-level architecture, 465
computational algorithm, 516
computational wavefront, 533
concatenation operator, 924
concurrent fault simulation, 794
conditional box, 189
conditional operator, 144
configurable logic blocks (CLBs), 474
conjunction operator, 17
consensus theorem, 18, 20
constant functions, 932
constants, 870–871
continuous assignment models, 154–156
continuous assignment statement, 144, 146–148
controller design, 669–670
controller, 355
counters, 195–201
counters synthesis, 304–305
CPLDs, *See* complex programmable logic devices (CPLDs)
CRC, *See* cyclic-redundancy check (CRC)
critical path, 753
cutset, 577
cycle time (period) constraint, 758
cyclic behavior, 152–154
cyclic-redundancy check (CRC), 171

D

D-notation, 782–786
data buffer, 587
dataflow graph, 346, 516–517
feedback, with
architectures for, 547–554
dataflow/RTL models, 156–160
datapath allocation, 518
datapath controllers, 345–399
binary counter, 347–353
RISC stored-program machine, 353–375
sequential machines, partitioned, 345–347
state-machine controller for, 346
UART, 375–399
datapath multiplexer, 368
data register cells, 795
data type, 865–871
common error, 867
passing variables through ports, 868
undeclared register variables, 867
constants, 870–871
nets, 865–866
register variables, 866–870
decimation, 570–575
decimation filter, 570–575
decision box, 189
declarations, 890–894
assignments, 892
block item, 893–894
data types, 891
delays, 891
net, 891
strengths, 891
variable, 891
function, 892–893
lists, 892
ranges, 892
task, 893
types, 890–891
module parameter, 890
port, 890
type, 890–891
UDP, 896–897
decoder, 64–67
binary, 65
block diagram symbols, 64
eight-client, 65
priority, 66
decomposition, 239
circuit after, 241
circuit before, 240
default (keyword), 163
default assignments, 254–255
delay
inertial, 129–130
transport, 131
delay control operator (#), 162
DeMorgan's laws, 18
Venn diagram, 19
demultiplexer, 61–62
dereferencing, hierarchical, 914
design documentation, 183–187
functions, 185–187
tasks, 183–185
design entry, 4
design for testability (DFT), 794–844
design tradeoffs, 547
differentiators, 570
digital design methodology, 1–12
design entry, 4–5
design integration and verification, 6
design partition, 4
design sign-off, 9
design specification, 4
electrical design rule checks, 9
fault simulation, 8
gate-level synthesis, 6
IC technology options, 9–10
parasitic extraction, 9
physical design rule checks, 9
placement, 8
postsynthesis design validation, 7
postsynthesis timing verification, 8
presynthesis sign-off, 6
routing, 8
simulation and functional verification, 5–6
model verification, 6
test execution, 6
test plan development, 5–6
testbench development, 6
technology mapping, 6
test generation, 8
digital filters, 556
design flow, 560
design process, 558–562
finite impulse response, 557
infinite impulse response, 557
digital integrators, 566
digital processors, 515–619
algorithms, 516–519
asynchronous FIFOs, 589–619
circular buffers, 586–589
data flow graphs, 516–519
digital filters, 554–566
halftone pixel image converter, 519–554
nested-loop programs, 516–519
pipelined architectures, 576–586
adder, 579–583
FIR filter, 583–586
signal processers, 554–566
digital filter design process, 558–562
finite-duration impulse response filter, 557–558
infinite-duration impulse response filter, 563–566
signal processors, building blocks for, 566–575
digital signal processing (DSP), 515, 554–566
applications, 555
characteristics, 555
constraints, 556
FIR filter, 562
I/O sample rates, 555
implementation, 555
instruction sets, 555–556
diminished radix complement, 630
direct form II (DF-II), 565
directed acyclic graph (DAG), 240–243, 517, 754
after extraction, 242
after substitution, 244
before extraction, 242
before factoring, 243
before substitution, 244
direct substitution, 915
disable (keyword), 180
disabled implicit nets, 933
discrete-time signal, 556
disjunction operator, 17
division, 715–742
reduced-register sequential, 734–740
signed (2s complement) binary numbers, 740
signed arithmetic, 740–741
STG based, 717, 724
unsigned binary numbers, 716–723
efficient division of, 724–733
don't-care, 17, 253–257
don't-care-set, 17
DRAMs, *See* dynamic random access memories (DRAMs)
DSP, *See* digital signal processing (DSP)
D-type flip-flop, 71–73, 209
negative-edge-triggered, 74
positive-edge-triggered, 73
dynamic hazard, 43, 52–55
effects of, 53
dynamic random access memories (DRAMs), 431, 469
dynamic timing analysis (DTA), 754

E

EDA, *See* electronic design automation (EDA)
edge detection, 152–154
edge-sensitive behavior, 368, 669
edge-sensitive storage elements, 70
edge-sensitive synchronous, 651
eight-client decoder, 65
eight-client priority decoder, 66–67

electronic design automation (EDA), 171
  synthesis tools, 180
    web sites, 952
electronic systems, 69
electrostatic-discharge circuitry, 860
elimination, 244
embeddable IP core, 470–472
embedded fault, 779
embedded timing controls, 321–324
encapsulation, 184
encoder, 62–64, 166
  priority, 63
  schematic symbols, 63
endpoint, 756
endprimitive, 132
equivalent faults, 793
erasable ROMs, 421–422
  floating-gate EEPROM, 422
  floating-gate transistor, 421
escaped identifier, 863
espresso, 239
event (keyword), 184
event control
  operator @, 162
  sensitivity list for, 940
event-driven simulation, 125
excess-3 code, 84–89
exclusive-or, 36
expand, 239
explicit state machines, 275–280
  BCD-to-excess-3 code converter, synthesis of, 276–280
exponentiation, 938–940
expressions, 905–909
expression substitution, 315–317
extended Karnaugh maps, 41–42
EXTEST (external test), 800
extraction, 240

F

factoring, 241
factorization, 243
false path, 767–769
FAN, 786
fatigue, 423
fault
  collapsing, 793
  coverage, 788, 792
  defect levels, 788
  detection, 780–782
  embedded, 799
  equivalent, 793
  grading, 792
  simulation, 775–794
  site, 776
fault-free circuit, 780
fault simulation, 8, 775–794
  automatic test pattern generation, 786–788
  concurrent, 794
  manufacturing tests and, 775–792
    circuit defects, 776–779
    detection, 780
    D-notation, 782–786
  parallel, 794
  probabilistic, 794
  serial, 793–794
feedback coefficients, 563
feedback-free netlists, 261
feedforward coefficients, 563
feed-forward difference equation, 557
ferroelectric nonvolatile memory, 452–454
field-programmable gate array (FPGA), 128, 142, 236, 415, 466–470, 515, 766
  role in ASIC market, 467–469
  synthesis with, 473–475
  technologies, 469–470
  verilog based design flow, 472–473
  volatile, 466
  web sites, 951
  XILINX Virtex, 470
field programmable logic devices (FPLDs), 417
FIFO. *See* first-in, first-out memory (FIFO)
filter
  causal, 557
  FIR, 556–558
  IIR, 556
  linear phase, 557
finite-duration impulse response (FIR) filter, 556–558
  Mth-order digital filter, 558
finite-state machine (FSM), 82, 345
finite-state machine datapath paradigm (FSMD), 191, 346
FIR. *See* finite-duration impulse response (FIR) filter
first-in, first-out memory (FIFO)
  asynchronous, 589–619
    clock domains synchronization, 599
    code converters for, 600
    simplified, 590–599
    status units, 601
  block diagram of, 590
  buffer, 591
  buffered clock domain interface, 603
flash memory
  cell phones, 430
  digital cameras, 430
  digital TV, 430
  microcontrollers, 430
  nonvolatile data storage, 430
  set-top boxes, 430
  telecommunications, 430
flattening, circuit, 244
flip-flop, 136, 925
  cyclic behavioral models of, 150–151
  D-type, 71–73, 209
  J-K, 75
  master–slave, 73–75
  sequential logic synthesis, 272–275
  T type, 75–76
  types of, 398
Floyd–Steinberg algorithm, 519
  halftone image converter, 520
  nearest neighbors, 520
  pixel image converter, 520
  pixel's roundoff error, 520
for (keyword), 176
forever (keyword), 180
fork join, 922
four-channel 32-bit multiplexer, 163
four-input OR gate, 263
four-value logic system, 119
FPGA. *See* field-programmable gate arrays (FPGA)
FPLDs. *See* fieldprogrammable logic devices (FPLDs)
fractions, 711–715
FSM. *See* finite-state machine (FSM)
FSMD. *See* finite-state machine datapath paradigm (FSMD)
full adder, 22
  Venn diagram, 23
functional units, 516
  addition, 632–638
  division, 715–742
  multiplication, 638–710
  subtraction, 632–638
functions
  constant, 932
  declarations of, 929–930
  recursive, 932
  signed, 936–937

G

gated clocks, 313
gate-level circuits, 636
gate-level models, 749–750
gate-level synthesis, 6
general-purpose machine, 515
generate bit, 633
generated instantiation, 895–896
glitches, 42–43
glue logic, 74, 642
gray-code, 600
Grayhill 072 hexadecimal, 218

H

half adder, 21
halftone pixel image converter, 519–554
  baseline design for a, 522–526
  dataflow graphs with feedback, architectures for, 547–554
  design tradeoffs, 547
  minimum concurrent processor architecture, 532–546
  NLP-based architectures, 526–532
  pixel updation using Floyd–Steinberg algorithm, 520
hardware description language (HDL), 103, 142, 235, 472, 516
  design flow, 3
  model, 640
hardware-performance spectrum, 526
hazard cover, 43
hazards, 43–55
  dynamic, 43, 52–55
  static, 43–52
HDL. *See* hardware description language (HDL)
hexadecimal scanner, 216
hierarchical decomposition, 110
hierarchical dereferencing, 914
hierarchical design, 4
high-level synthesis, 246
high-resistance
  cmos transmission gate, 859
  nmos pass transistor switch, 858
  pmos pass transistor switch, 858
hold time, 770
H-trees, 765
hysteresis effect, 452

I

identifier, 108, 863
IEEE standard 1149.1, 800
IEEE standard 1364-2001, 887, 927
if (keyword), 164
implicant, 29
implicit combinational logic, 144
implicit nets, 933
implicit-state-machine binary multiplier, 675–687
implicit state machine synthesis, 302–304
indeterminate assignment, 918–921
indirect substitution, 915
industry organization
  web sites, 951
inertial delay, 129–130
infinite-duration impulse response filter, 556, 563–566
  direct form II (DF-II), 565
  Nth-order filter, 564
  transposed direct form II (TDF-II), 565
  type-1, 563
initial (keyword), 122
in-line redefinition, 942
inout, 108
input, 108
input delay constraint, 757
input–output (pad to pad) constraint, 758
input–output latency, 578
instance generation, 944–948
instruction register (IR), 799
  boundary scan, 799
instruction set, 356–358
integrated circuits (ICs), 415
integrators, 566–569
intellectual property (IP), 470, 755
  reuse, 178–179
interpolation filters, 570–575
INTEST (internal test), 801
intra-assignment delay, 917
IP, *See* intellectual property (IP)
irredundant, 239

J

J-K flip-flop, 75, 77
Johnson code, 301
Johnson counter, 349
JTAG port, 794–844
  boundary scan, 795–796
  built-in self-test, 830–844
  design for testability, 794–795
  instructions, 800–801
  modes of operation, 796–797
  registers, 798–800
  TAP architecture, 801–807
  testing with, 807–830

K

Karnaugh map, 93, 115, 457
  don't-cares and, 40
  extended Karnaugh maps, 41–42
  POS form, 39–40
  SOS form, 36–39
Keypad scanner, 213–223
  ASM chart of, 217
  grayhill 072
  hexadecimal keypad, 215
  hexadecimal, 218
keyword, 863–864
K-maps, *See* Karnaugh maps

L

language formal syntax, 885, 887–911
  behavioral statements, 897–901
    case, 900
    conditional, 899–900
    continuous assignment, 897
    looping, 900
    parallel, 898
    procedural blocks, 89
    sequential blocks, 898
    statements, 898–899
    task enable, 900
    timing control, 899
  declarations, 890–894
    assignments, 892
    block item, 893–894
    data types, 891
    function, 892–893
    lists, 892
    ranges, 892
    task, 893
    types, 890–891
  expressions, 905–909
    concatenations, 905
    function calls, 906
    left-side values, 908
    numbers, 908–909
    operators, 908
    primaries, 907
    strings, 909
  general, 909–911
    attributes, 909
    comments, 909
    identifier branches, 911
    identifiers, 909–911
    white space, 911
  generated instantiation, 895–896
  module instantiation, 895–896
  primitive instances, 894–895
    gate, 895
    strengths, 895
    switch types, 895
    terminals, 895
  source text, 887–890
    configuration, 887–888
    library, 887
    module items, 889
    module parameters, 888–889
    module, 888
    ports, 888–889
    primitive, 888
  specify section, 901–905
    block declaration, 901
    block terminals, 901
    path declarations, 901
    path delays, 902–903
    system timing checks, 903–905
  UDP, 896–897
    body, 897
    declaration, 896
    instantiation, 896–897
    ports, 896–897
language reference manual (LRM), 867, 919
language rules, 108–109
latch
  cyclic behavioral models of, 150–151
  feedback circuit structures implementing, 70
  sequential logic synthesis, 260–269
    accidental synthesis, 262–266
    intentional synthesis, 266
  S-R (set–reset), 70
  transparent, 71
  types, 925
  in verilog, 148–150
leading-edge devices, 466
least significant bit (LSB), 84, 376, 627
LED, *See* light-emitting diode (LED) display
level-sensitive behavior, 368, 669
level-sensitive cyclic behavior, 249
level-sensitive storage elements, 70
LFSR, *See* linear feedback shift register (LFSR)
light-emitting diode (LED) display, 170
  seven-segment, 171
linear feedback shift register (LFSR), 171–173, 830
  data movement in, 173
  with modulo-2 addition, 172
line converter, 298
line justification, 784
local parameter, 944
logical adjacency theorem, 19
logic design, 141–223
  ASM charts, 187–191
  ASMD charts, 191–195
  asynchronous signals, 208–223
    metastability for, 208–213
    switch debounce for, 208–213
    synchronizers, 208–213
  behavioral modeling, 141–143
  behavioral models of, 195–208
    arrays of registers, 206–208
    counters, 195–201
    register files, 206–208
    shift registers, 202–206
  continuous assignments, 146–148
  cyclic behavior, 152–154
  decoders, 162–171
  design documentation, 183–187
  functions, 185–187
    tasks, 184–185
  edge detection, 152–154
  encoders, 162–171
  flip-flops, cyclic behavioral models of, 150–151
  keypad scanner, 213–223
  latches, cyclic behavioral models of, 150–151
  linear-feedback shift register, dataflow models of, 171–173
  modeling digital machines, with repetitive algorithms, 173–181
    clock generators, 180–181
    intellectual property reuse, 178–180
    parameterized models, 178–180
  multicycle operations, machines with, 182–183
  multiplexers, 162–171
  propagation delay, 146–148
logic modeling, 933–934

logic synthesis, 237–245, 248
logic system, 118–126
  design verification, 118
  event-driven simulation, 125
  four-value logic, 119–123
  signal generators for testbenches, 123–125
  signal resolution, 119–123
  sized numbers, 126
  test methodology, 120–123
  testbench template, 125
logic value 0,1, x, z, 119
lookup table (LUT), 433
loops, synthesis of, 318–334
  nonstatic loops
    with embedded timing controls, 326–329
    without embedded timing controls, 324–326
  static loops
    of bitwise-and operations, 319
    with embedded timing controls, 321–324
    without embedded timing controls, 318–321
  unsynthesizable loops, state-machine replacements for, 329–334
LSB, *See* least significant bit (LSB)

M

machine language, 356
Manchester encoders, 90
Manchester line code converter, 281–284
  Mealy-type NRZ-to, 281–282
  Moore-type NRZ-to, 283–284
mask programmable gate array (MPGA), 416
mask-programmable logic device (MPLD), 417
master–slave flip-flop, 73–75
maxterm, 26
Mealy machine, 83, 88
  registered output, 296
Mealy sequence recognizer, 291, 293
Mealy-type NRZ
  Manchester line code converter, to synthesis, 281–282
media, web sites, 951
memory allocation, 931
metastability, 208
minimal cover, 33
minimum concurrent processor architecture, 532–546
minterm, 24
misII, 239
MISR, *See* multiple-input signature register (MISR)
modeling tips, 105, 109, 110, 114, 119, 123, 125, 126, 132, 135
modem, 375
modem clock, 376
module, 106, 888
  arrays of, 913
  control unit, 359
  declarations, 927
  instantiation, 895–896
  memory unit, 359
  parameters, 888
  port mode, 927
  port parameter list, 927
  ports, 108
  processing unit, 359
monitoring mechanism, 146
Moore machine, 83
  registered output, 296
Moore sequence recognizer, 291, 293
Moore-type NRZ
  Manchester line code converter, to synthesis, 283–284
MOS bidirectional switches, 860–861
MOS pull-down gates, 860
MOS pull-up gates, 860
MOS transistor switches, 855–859
most significant bit (MSB), 85, 173, 376, 627
MPGAs, *See* mask programmable gate array (MPGA)
MPLDs, *See* mask-programmable logic device (MPLD)
MSB, *See* most significant bit (MSB)
multicycle operations, 182–183
multiinput combinational logic gates, 851–854
multilevel combinational logic, 240
multiple-input signature register (MISR), 830, 835
multiplexer, 60–61
  behavioral modeling, 162–171
  n-channel, 61
  registered output, with, 294
  two-channel circuit, 60
multiplicand, 670
multiplicand register, 644
multiplication, 711–715
  fractions, 711–715
  negative multiplicand, 714–715
    negative multiplier, 715
    positive multiplier, 714
  positive multiplicand, 714
    negative multiplier, 714–715
    positive multiplier, 714
  signed binary numbers, 710–711
multiplier, 670
multiply and accumulate (MAC), 556
mux, 252

N

named (abstract) events, 922–923
NAND
  circuit transformations for, 58
  latch configuration, 210
  NOR structures, and, 55–59
n-channel MOS (nMOS), 418
negative integers
  ones complement, 629–630
  signed magnitude of, 628–629
  twos complement, 630–632
negative multiplicand, 710–711, 714–715
negative multiplier, 710–711, 714–715
negedge (keyword), 135
nested-loop program (NLP), 516–517
  equivalent combinational logic, 531
  halftone pixel image converter, architectures for, 526–532
nested modules, 109
nets, 865–866
  addressing, 867–868
  referencing arrays of, 871
  scope of, 870
net variables, 143
next state (NS), 80
n-input primitives, 105
NLP, *See* Nested-loop program (NLP)
nmos pass transistor switch, 856
nonblocking assignment, 151
nonrecurring engineering (NRE), 420
non-return-to-zero (NRZ) code, 90
non-return-to-zero invert-on-ones (NRZI) code, 90
nonstatic loop, 324–326
NOR, 70
  circuit transformations for, 59
  NAND structures, and, 55–59
n-output primitives, 105
number representation
  fractions, 632
  negative integers, 628–632
  positive integers, 630–632
number wheel
  1s complement numbers, 629
  2s complement numbers, 630
  signed-magnitude numbers, 629

O

off-set, 37
one-hot encoding, 301
on-set, 37
operator, 873–883
  #, 162, 186
  ?, 144
  @, 152, 186
  <=, 151
  =, 157
  arithmetic, 873–875
  assignment width extension, 883
  bitwise, 875
  built-in verilog operators, 145
  conditional, 144, 878–879
  delaycontrol operator, 162
  event-control operator, 162
  expressions, 880
  logical, 876–877
  operands, 880
  precedence, 880–881
  reduction, 875–876
  relational, 877
  shift, 878
  sign conversion, system functions for, 882
  signed data types, arithmetic with, 881
  signed literal integers, 882
operator grouping, 314–315
oscilloscope, 107
output delay constraint, 757
overflow, 638

P

PAL, *See* programmable array logic (PAL)
parallel fault simulation, 794
parallelism, 518
parameter, 927
  constants, 943
  keyword, 184
  local, 944
  redefinition, 943
  substitution, 915
parameterized models, 178
parasitic capacitance, 9
parent module, 109

parse trees, 247
partial product accumulation, 641
partitioned sequential machine, 345
path-oriented decision making algorithm (PODEM), 786
p-channel MOS (pMOS), 418
phase lock loop (PLL), 90
pipelined adder, 579–583
 16-bit adder structure, 581
 data movement, 580–582
pipelined architectures, 576–586
pipelined finite impulse response filter, 583–586
pipeline registers, 576–577
 benefits, 578
 cutset-based placement of, 577
 disadvantage, 578
pipelining, 518, 632
pixel array, 522
pixel converter, 520
pixel processor datapath unit (PPDU), 522
PLA, *See* programmable logic array (PLA)
place-and-route engine, 475
placement, 8
PLAs, *See* programmable logic arrays (PLAs)
PLDs, *See* programmable logic devices (PLDs)
pmos pass transistor switch, 856
pMOS, *See* p-channel MOS (pMOS)
port, 868, 888, 929
POS, *See* product-of-sums (POS)
posedge (keyword), 135
positive integers
 twos complement, 630–632
positive multiplicand, 710, 714–715
positive multiplier, 710, 714
postsynthesis design tasks, 749–844
 ASIC timing violations, elimination of, 766–767
  options for, 766
 false paths, 767–769
 fault simulation, 775–794
 JTAG ports, 794–844
 manufacturing tests, 775–792
 timing verification, 753–766
  factors that affect, 760–766
  methods comparison, 754
  static timing analysis, 755–757
  timing specifications, 757–759
 timing verification, system tasks for, 769–775
 validation, 749–753
PPDU, *See* pixel processor datapath unit (PPDU)
present state (PS), 80
primary input, 758
primary output, 758
prime implicant, 31
primitive, 104, 851–861, 888
 arrays of, 913
 instances, 894–895
 modeling combinational logic gates for, 105
priority decoder, 66–67
priority encoder, 63–64
probabilistic fault simulation, 794
product operator, 17
product-of-sums (POS), 26
 NOR/inverter realization, 57
programmable array logic (PAL), 416, 463–464
 applied micro devices (AMD), 463
 dual-array structure, 463
 floating-gate link transistors, 463
 microprocessor systems, 463
 PLD-based latches, 463
 Readmemb task, 460
programmable IP core, 470–472
programmable logic array (PLA), 416, 454–463
 minimization, 457–459
 modeling, 459–463
 NOR-NOR logic, 457
 OR-AND logic, 457
 wired-OR logic, 456
programmable logic devices, 415–475
 complex, 465
 field-programmable gate arrays, 466–470
  synthesis with, 473–475
  verilog-based design flows, 472–473
 programmability of, 464
 programmable array logic (PAL), 463–464
 programmable logic array (PLA), 454–463
  PLA minimization, 457–459
  PLA modeling, 459–463
 system-on-a-chip (SoC), 470–472
  embeddable IP cores for, 470–472
  programmable IP cores for, 470–472
programmable ROM (PROM), 420–421
 fusible-link bipolar PROM, 421
 OR-plane, 420
 pull-down device, 420
 pull-up device, 420
programming language interface, 949–950
 applications, 949
PROM, *See* programmable ROM (PROM)
propagate bit, 633
propagation delay, 126–137, 146–148
 inertial delay, 129–130
 transport delay, 131
PRPG, *See* pseudo-random pattern generator (PRPG)
pseudo-random pattern generator (PRPG), 830
Pull-down device, 420, 860
Pull-down resistor, 215
Pull-up device, 420, 860
Pulsewidth constraint, 773
Push-button input device, 209

Q

Q-format, 712
Quine–McCluskey minimization algorithm, 27–28

R

race conditions, 918–921
radix, 628, 713, 936
radix-4 recoding, 687
random-access memory (RAM), 417, 555, 587
read-only memory (ROM), 416–420
 comparison of ROMs, 426
 erasable ROMs, 421–423
 implementation of combinational logic, 423
 programmable ROM (PROM), 420–421
 state machines, 426–430
 verilog system tasks, 423–426
recomposition, 518
reconvergent fanout, 44
recovery time, 774–775
recursive function, 932
reduced-register sequential divider, 734
reduced-register sequential multiplier, 670–675
redundant cube, 51, 53
re-entrant task, 931–932
reg (data type), 866
register bank, 79
registered logic, 292–300
register file, 206–208
registers synthesis, 305–309
register transfer level (RTL), 104, 154, 237, 350, 518, 749
 synthesis, 245–246
register transfer notation (RTN), 194
register variables, 143, 865–871
 arrays, 868–869
 data type, 865–867
 referencing arrays of, 871
 scope of a variable, 869
 strings, 869–870
 undeclared, 867
repeat (keyword), 154
replication, 518
reservation table, 535
resets, 309–313
resistive bidirectional switch, 861
resistive three-state bidirectional switch, 861
resistor-capacitor lowpass filter, 209
resource allocation, 246
resource scheduling, 246, 518
resource sharing, 258–260
 datapath with, implementation of, 260
return-to-zero (RZ) code, 90
ring counter, 305
ripple-carry adder, 632, 946
ripple counter, 304
RISC stored-program machine
 design, 353–375
 synthesis, 353–375
  ALU, 355
  controller design, 358–372
  controller, 355–356
  instruction set, 356–358
  processor, 355
  program execution, 372–375
ROM, *See* read-only memory (ROM)
routing, 8
RTL, *See* register transfer level (RTL)

S

SAMPLE/PRELOAD, 801
scalar, 875
scan path, 8
scan register, 791
Schneider's circuit, 784
self-aligning divider, 726
semiconductor manufacturers
 web sites, 951
sensitivity list
 combinational logic, 940–942
 event control, 940

sequence recognizer
for detecting three successive 1s, 285, 287
Mealy-type 291, 293
Moore-type, 291, 293
synthesis of, 284–292
sequential binary multiplier, 642–646
ASMD-Based, 658–664
hierarchical decomposition, 644–646
register transfers in, 645
state-transition graphs (STGs) based, 652–657
structural units of, 646
sequential circuits
test generation for, 788–792
sequential decimator, 573
sequential logic
flip-flops, synthesis with, 272–275
latches, synthesis with, 260–269
truth table models with verilog, 131
sequential logic design, 69–99
BCD to excess-3 code converter, 84–89
busses, 76–80
data transmission, serial-line code converter for, 90–95
Mealy-type FSM for, 92–93
Moore-type FSM, 93–95
flip -flops, 71–76
D-type, 71–73
J-K, 75
master-slave, 73–75
T type, 75–76
state reduction, 95–99
equivalent states, 95–99
state-transition graphs, 82–84
storage elements, 69–70
latches, 70–71
transparent latches, 71
three-state devices, 76–80
sequential machine, 69
algorithm-based synthesis of, 519
design of, 80–82
partitioned, 345
sequential multiplier, 638, 652, 715
Booth's algorithm, 687–702
reduced-register, 670–675
serial fault simulation, 793–794
serializer deserializer (SerDes), 375
serial-line code converter, 90–95
Mealy-type FSM, 92–93
Moore-type FSM, 93–95
serial-to-parallel converter, 604
setup and hold constraint, 771
setup constraint, 770
setup time, 770
seven-segment LED display, 256
Shannon expansion, 20
Shannon's sampling theorem, 554, 570
shift operator, 179
shift register, 202–206
with registered combinational logic, 306
with separate, unregistered combinational logic, 307
sigma-delta modulator, 566
signal resolution, 119–120
signal processors
building blocks for, 566–575
decimation filters, 570–575
differentiators, 570
integrators, 566–569
interpolation filters, 570–575
signal skew constraint, 773–774
sign conversion, 937
signed binary numbers, multiplication of, 710–711
negative multiplicand, 710
negative multiplier, 710–711
positive multiplier, 710
positive multiplicand, 710
negative multiplier, 710
signed data type, 934–935
signed function, 936–937
signed literal integer, 935–936
signed port, 935
sign-extended multiplicand, 714
sign-extended multiplier, 712
sign-off, 6
simulation cycle, 920
simulator, 918–919
single pass behavior, 123
single stuck fault model, 783
sized radix-specified number, 882
skew, 773
skew-free circuit, 761
skew-free clock, 762
effect of, 764
SoC, *See* systems on a chip (SoC)
SOP, *See* sum-of-products (SOP)
Spice, 2
S-R (set-reset) latch, 70
with an enabling input signal, 72
SRAM, *See* static RAM (SRAM)
SSOP, *See* standard sum-of-product (SSOP)
standard-cell library, 127
standard delay format (SDF), 942
standard sum-of-product (SSOP), 24
startpoint, 756
state assignment, 300–302
commonly used codes, 301
state box, 192
state encoding, 300–302
assignment guidelines, 300
state machine
explicit, 275–280
implicit, 302–305
ROM-based, 426–430
state-per-bit encoding, 475
state reduction, 95–99
state transition graph (STG), 82–84, 187, 346, 717
controller design, 646–652
multiplication with bit-pair recoding, 705
sequential binary multiplier, 652–657
synchronous 4-bit binary counter, 349
static 0-hazard, 43
static 1-hazard, 43–44
static hazard, 43
effects of, 53
elimination of(SOP Form), 45–48
modified circuit, 45
multilevel circuits, in, 49–52
static loop, 318–324
static random access memory (SRAM), 430–452
block diagram symbol, 432
circuit structure, 431
read cycle, parameters for, 452
transistor-level SRAM cell, 432
verilog functional models, 432
with bidirectional data port, 434–436
static timing analysis (STA), 754–757
STG, *See* state-transition graph (STG)
storage devices, 417–454
ferroelectric nonvolatile memory, 452–454
flash memory, 430
programmable ROM (PROM), 420–421
read-only memory (ROM), 418–430
static random access memory (SRAM), 430–452
storage elements, 69–70
stored-program machine, 353
strings, 909
structural model, 107
structural modeling, 141
stuck at 0 faults, 778
stuck at 1faults, 778
stuck faults, 776
substitution, 243
subtract-and-shift algorithm, 715
sum-of-products (SOP), 23, 417
NOR/inverter realization, 57
static hazard, elimination of, 45–48
sum operator, 17
Superlog, 2
switch debounce, 208
synchronized gray code, 602
synchronizer, 208
across clock domains, 214
for asynchronous input signals, 212
synchronous circuit, 69, 756
path groups for, 758
timing of, factors affecting, 760
synchronous implementation, 527
synchronous machine, 81, 83, 156
throughput, 576
syntax (Verilog), 929
synthesis, 236–247
bus interfaces, 269–272
clock enables, 313
combinational logic, 247–260
counters, 302
design traps to avoid, 334–335
explicit state machines, 275–280
gated clocks, 313
high-level synthesis, 246
implicit state machines, 302–305
logic, 237–245
loops, 318–334
Mealy-type NRZ-to-Manchester line code converter, 281–282
Moore-Type NRZ-to-Manchester line code converter, 283–284
partitioning a design, 335–336
registered logic, 292–300
registers, 304, 305–309
resets, 309–313
results of, anticipating the, 314–317
data types, 314
expression substitution, 315–317
operator grouping, 314–315
RTL, 245–246
sequence recognizer, 284–292
sequential logic with latches, 260–269
flip-flops, 272–275
state encoding, 300–302
three-state devices, 269–272
synthesis-tool organization, 238
system timing checks, 903–905

commands, 903–904
arguments, 904
event definitions, 904–905
SystemC, 2
system-on-chip (SoC), 109, 470
systolic array, 641

T

tabular format, 459
TAP, *See* test access port (TAP)
task (key word), 184
Tasks, 931
technology options, 9–10
test access port (TAP), 797
architecture, 801–803
coefficients, 558
controller state machine, 803
testbench, 6
testbench template, 125
test clock (TCK), 801
test-data input (TDI), 798
test-data output (TDO), 798
test-data registers (TDRs), 798
testing
fault simulation, 8
test execution, 6
test plan, 5–6
testbench development, 6
test mode select (TMS), 801
T flip-flop, 75–76, 78
three-state bidirectional switch, 861
three-state buffer, 854
three-state devices, synthess of, 269–272
three-state inverter, 855
three-state logic gates, 854–855
timing analysis
dynamic, 754
signal paths for, 756
static, 754
timing check, 770–775
clock period, 774
hold condition, 770–772
pulsewidth constraint, 773
recovery time, 774–775
setup condition, 770–772
signal skew constraint, 773–774
timing closure, 754
timing constraint
cycle time, 758
input delay, 757
input-output, 758
output delay, 757
skew, 773
timing diagram, 82
timing margin, 750
timing parameter, 450
timing verification, 7, 749
TMS, *See* test mode select (TMS)
Top-down design, 4, 109
transistor-capacitor storage, 431
translation engine, 238
transparent latches, 71
transparent latch model, 149
with active-high enable, 150
with active-low reset, 150
by a continuous assignment statement, 149
transport delay, 131
transposed direct form II (TDF-II), 565
two-channel mux, 146, 265
two-stage pipeline register, 192

U

UART, *See* universal asynchronous receiver and transmitter (UART)
UDPs, *See* user-defined primitives (UDPs)
ultraviolet (UV), 422
undeclared register variables, 867
underflow, 638
unit under test (UUT), 121, 217, 830
universal asynchronous receiver and transmitter (UART), 375–399
ASCII text transmitted by, 376
block diagram of, 377
block diagram of, 377
communication, 376
operation, 376–377
receiver, 387–399
ASMD chart for, 391
block diagram of, 390
circuits synthesized from, 398–399
for clock regeneration, sampling format for, 389
transmitter, 377–387
unsized radix-specified number, 882
Untestable faults, 786
user-defined primitive (UDP), 131, 239, 896–897
declaration, 896–897, 929
instantiation, 896–897
ports, 896–897
UUT, *See* unit under test (UUT)

V

variable part selects, 933–934
vector, 875
Venn diagram
adder cell, 633
consensus, 20
DeMorgan's laws, 19
full adder, 23
logical adjacency, 20
Verilog 1364, 863
Verilog-2001, 2005, 927–948
ANSI C style changes, 927–930
functions, declarations of, 929–930
module declarations, 927
module port mode, 927
module port parameter list, 927–928
tasks, 929–930
type declarations, 927
UDP declarations, 929
variables, initialization of, 930
arithmetic, support for, 934–940
arithmetic shift operator, 937–938
assignment width extension, 938
exponentiation, 938–940
sign conversion, system functions for, 937
signed data types, 934–935
signed functions, 936–937
signed literal integers, 935–936
signed ports, 935
code management, 930–932
constant functions, 932
recursive functions, 932
re-entrant tasks, 931–932
combinational logic, sensitivity list for, 940–942
event control, sensitivity list for, 940
instance generation, 944–948
logic modeling, 933–934
arrays, 934
disabled implicit nets, 933
implicit nets, 933
variable part selects, 933–934
parameters, 942–944
local parameters, 944
parameter constants, 943
parameter redefinition, 943–944
Verilog, additional features of, 913–923
constructs supported by synthesis tools, 923
fork join statement, 922
hierarchical dereferencing, 914
indeterminate assignment, 918–921
intra-assignment delay, 917
named (abstract) events, 922–923
parameter substitution, 915–916
primitives, arrays of, 913–914
procedural continuous assignment, 916–917
race conditions, 918–921
wait statement, 921–922
Verilog standards committee, 927
vertex, 17
very large-scale integrated (VLSI), 15
volatility, 423

W

wait (keyword), 921–922
wait construct, 162
wavefront index, 533
web-based resources, 953
web-based resources, 953
web sites
EDA tools, 952
FPGA, 951
industry organization, 951
media archives, 951
resources and training, 952
semiconductor manufacturers, 951
while, 175
white space, 911
wire, 933
word line, 418
worst-case delay, 760

X

Xilinx
Virtex FPGAs, 470
Virtex-ii, 471

Y

Y-chart, 237

Index of Verilog Modules and User-Defined Primitives

Add (sum, a, b, c_in), 941
Add (sum, c_out, a, b, c_in), 928
Add_16, 928
add_16_pipe, 582
add_4cycle, 182
Add_Accum_1, 309
Add_Accum_2, 309
Add_full, 110, 111
Add_full_ASIC, 128
Add_full_unit_delay, 127
Add_half, 104, 111
Add_half_ASIC, 128
Add_half_unit_delay, 127
Add_prop_gen, 637
Add_rca_16, 111
Add_rca_4, 111
Add_RCA_or_CLA, 947, 948
Add_Sub, 741, 936
Adder (sum, c_out, a, b, c_in), 933, 940, 941
Adder (sum, diff, a, b, c_in), 940, 942
Adder, 933, 941, 942
Adder_CLA, 946
adder_task, 184
Address_Register, 366
Alu_RISC, 368
alu_with_z1, 257
annotate, 916
AOI_5 _CA3, 147
AOI_5_CA0, 143
AOI_5_CA1, 144
AOI_5_CA2, 145
AOI_str_1995, 108
AOI_str_2005, 108
arith1, 874
arithmetic_unit, 187
array_of_adders, 914
array_of_nor, 913
ASIC, 815, 838
ASIC_with_BIST, 837
ASIC_with_TAP, 813
asynch_df_behav, 152
Auto_LFSR, 178
Auto_LFSR_ALGO, 174
Auto_LFSR_RTL, 172
B2G_Conv, 610
B2G_Reg, 610
badd_4, 258
barrel_shifter, 204
BCD_to_Excess_3_ROM, 429
BCD_to_Excess_3b, 279
best_gated_clock, 313
Bi_dir_bus, 271
Binary_Counter_Part_RTL, 351
BIST_Control_Unit, 839
Bogus (sum, diff, a, b, c_in), 940
boole_opt, 249
Boundary_Scan_Register, 810
BR_Cell, 799
BSC_Cell, 796
Bubble_Sort, 549
Bypass_Register, 815
Circular_Buffer_1, 588
Circular_Buffer_2, 588
Clk_gen (clock), 931
Clk_Gen, 931
Clock_Unit, 375
Comp_2_algo, 160
Comp_2_CA0, 147
comp_2_CA1, 154
Comp_2_CA2, 155
Comp_2_RTL, 157
Comp_2_str, 116
Comp_4_str, 117
comparator, 250
compare_32_CA, 156
Control_Unit, 331, 368, 382, 393, 550, 605, 648, 668, 673, 683, 700, 728, 737
Control_Unit_by_3, 352
controller, 193, 656
Controller_Booth_STG_0, 692
Controller_Radix_4_STG_0, 706
count_ones_a, 321
count_ones_b0, 323
count_ones_b1, 323
count_ones_b2, 324
count_ones_c, 325
count_ones_d, 326
count_ones_IMP, 334
count_ones_SD, 328
count_ones_SM, 331
D_flop, 366
D_reg4_a, 274
Datapath, 657
Datapath_Booth_STG_0, 693
Datapath_Radix_4_STG_0, 708
Datapath_Unit, 332, 352, 550, 606, 668, 673, 685, 720, 723, 729, 738
Decimator_1, 570
Decimator_2, 571
Decimator_3, 573
Decoder, 168
df_behav, 151
Differentiator, 570
Divider_RR_STG, 735
Divider_STG_0, 719
Divider_STG_1, 727
empty_circuit, 275
Encoder, 166

expression_sub, 317
FIFO_Channel, 607
FIFO_Control_Unit, 596
FIFO_Datapath_Unit, 596
FIFO_Dual_Port, 595
FIFO_Status_Unit, 596, 608
find_first_one, 181
FIR_Gaussian_Lowpass, 561
Flop_event, 923
FLOP_PCA, 917
for_and_loop_comb, 319
G2B_Conv, 610
generated array_pipeline, 947
hdref_Param, 915
Hex_Keypad_Grayhill_072, 220
IIR_Filter_8, 564
Image_Converter_0, 527
Image_Converter_1, 528
Image_Converter_Baseline, 524
Image_Converter_Concurrent_Processors, 536
Instruction_Decoder, 815
Instruction_Register, 366, 810
Integrator_Par, 567
Integrator_Seq, 568
IR_Cell, 799
Latch_CA, 149
latch_if1, 267
latch_if2, 268
Latch_Race_1, 752
Latch_Race_2, 752
Latch_Race_3,752
Latch_Race_4, 752
Latch_Rbar_CA, 150
Latched_Seven_Seg_Display, 254
Majority, 176
Majority_4b, 176
Memory_Unit, 372
modXnor, 915, 916
multiple_reg_assign, 317
Multiplexer_3ch, 367
Multiplexer_5ch, 367
Multiplier_ASM_0, 660
Multiplier_ASM_1, 667
Multiplier_Booth_ASMD, 700
Multiplier_Booth_STG_0, 691
Multiplier_IMP_1, 678
Multiplier_IMP_2, 682
Multiplier_Radix_4_STG_0, 706
Multiplier_RR_ASM, 672
Multiplier_STG_0, 648
Multiplier_STG_1, 655
Mux_2_32_CA, 146
Mux_4_32_CA, 165
Mux_4_32_case, 163
Mux_4_32_if, 164
mux_4pri, 253
mux_latch, 266
mux_logic, 252
mux_reg, 297
mux4_PCA, 916
my_design, 107
NRZ_2_Manchester_Mealy, 281, 282
NRZ_2_Manchester_Moore, 283
NRZI_Mealy, 299
operator_group, 315
or_nand, 249
or4_behav, 262
or4_behav_latch, 263
Par_load_reg4, 203
Param, 915
Pattern_Generator, 839
PLA_array, 462
PLA_plane, 463
PP_Control_Unit, 538
PP_Datapath_Unit, 539
PP_Memory_Unit, 539
PPDU, 524
Priority, 167
Processing_Unit, 365
Program_Counter, 367
RAM_2048_8, 446
RAM_static, 433
RAM_static_BD, 435
rd_cntr_Unit, 597, 610
Register_File, 208
Register_Unit, 366
res_share, 259
Response_Analyzer, 838
ring_counter, 200
ripple_counter, 304
RISC_SPM, 365
ROM_16_x_4, 424
ROM_BCD_to_Excess_3, 429
Row_Signal, 221
Seq_Rec_3_1s_Mealy, 287
Seq_Rec_3_1s_Mealy_Shft_Reg, 292
Seq_Rec_3_1s_Moore, 288
Seq_Rec_Moore_imp, 311
Ser_Par_Conv_32, 605
Seven_Seg_Display, 170
Shift_reg4, 202
shifter_1, 306
shifter_2, 307
shiftreg_nb_V05, 159
shiftreg_PA, 157
shiftreg_PA_rev, 158
sn54170, 269
source_text, 888
swap_synch, 273
Synchro_Long_Asynch_in_to_Short_Period_Clock, 611
Synchro_Short_Asynch_in_to_Long_Period_Clock, 610
Synchronizer, 221
t_Add_half, 122
t_ASIC_with_BIST, 840
t_ASIC_with_TAP, 817
t_Bubble_Sort, 551
t_DUTB_name, 125
t_FIFO_Channel, 611
t_FIFO_Dual_Port, 597
t_Image_Converter_1, 529
t_Image_Converter_Baseline, 525
t_Image_Converter_Concurrent_Processor, 544
t_NRZI_Mealy, 299
t_Seq_Rec_3_1s, 289
TAP_Controller, 811
TDI_Generator, 822
TDO_Monitor, 822
test_BCD_to_Excess_3b_Converter, 429
test_Divider_STG_1, 730
test_Multiplier_STG_0, 649, 694
test_RAM_2048_8, 448
test_RAM_static_BD, 437
test_RISC_SPM, 374
UART_RCVR, 392
UART_XMTR, 382
Uni_dir_bus, 271
Universal_Shift_Reg, 206
up_down_counter, 200
up_down_counter, 201
Up_Down_Implicit1, 198
word_aligner, 186
wr_cntr_Unit, 597, 609

Summary of Key Verilog Features (IEEE 1364)

(Examples include 1995, 2001, and 2005 syntax)

Design Encapsulation:

A Verilog module encapsulates functionality; modules may be nested to any depth.

module name_1995 (list of ports);
//Declarations
Port modes: **input, output, inout** identifier;
Nets (e.g., **wire** A[3:0];)
Register variable (e.g., **reg** B[31: 0];)
Constants: (e.g. **parameter** size = 8;)
Named events (e.g., **event** trigger;)
Continuous assignments
(e.g. **assign** sum = A + B;)
Behaviors **always** (cyclic), **initial** (single-pass)
specify ... endspecify
function ... endfunction
task ... endtask
generate ... endgenerate
//Instantiations
primitives
modules
endmodule
module name_2001_2005 **#(parameter**
par_1 = val1, ...)(
output reg [size -1: 0] out_port1, ...
input [size_in -1: 0] in_port1,...
);
//Declarations
//Instantiations
endmodule

Multi-(Scalar) Input Primitives

(Each input is a scalar)

and (out, in_1, in_2, ... in_N);
nand (out, in_1, in_2, ... in_N);
or (out, in_1, in_2, ... in_N);
nor (out, in_1, in_2, ... in_N);
xor (out, in_1, in_2, ... in_N);
xnor (out, in_1, in_2, ... in_N);

Multi-Output Primitives

buf (out_1, out_2, ..., out_N, in); // buffer
not (out_1, out_2, ..., out_N, in); // inverter

Three-State Logic Primitives

bufif0 (out, in, control);
bufif1 (out, in, control);
notif0 (out, in, control);
notif1 (out, in, control);

Pullups and Pulldowns

pullup (out_y); **pulldown** (out_y);

Propagation Delays

Single delay: **and** #3 G1 (y, a, b, c);
Rise/fall: **and** #(3, 6) G2 (y, a, b, c);
Rise/fall/turnoff: **bufif0** #(3, 6, 5) (y, x_in, En);
Min:typ:Max: **bufif1** #(3:4:5, 4:5:6, 7,8:9) (y, x_in, en);

Simulator command line options for single delay value simulation:
+maxdelays, +typdelays, +mindelays

Example: verilog +mindelays testbench.v

Concurrent Behavioral Statements

May execute a level-sensitive assignment of value to a net (keyword: **assign**), or may execute the statements of a cyclic (keyword: **always**) or single-pass (keyword: **initial**) behavior. The statements in a behavior execute sequentially or concurrently, depending on the assignment operator, subject to level-sensitive or edge-sensitive event control expressions.

Syntax: **Continuous assignment**
assign net_name = [expression];

Syntax: **Cyclic behavior**
always begin [procedural statements] **end**

Syntax: **Single-pass behavior**
initial begin [procedural statements] **end**

Cyclic (**always**) and single-pass (**initial**) behaviors may have level sensitive and/or edge sensitive timing control.

Edge sensitive: **always @(posedge** clock) q <= data;

Level sensitive: **always @** (enable, data)
if (enable) q <= data

Sensitivity Lists and Event Control Expressions

Verilog 1995: @ (a **or** b)
Verilog 2001: @ (a, b)

Data Types: Nets and Registers

Nets: Used to establish structural connectivity between instantiated primitives and/or modules; may be target of a continuous assignment; (e.g., **wire**, **tri**, **wand**, **wor**).

Value is determined during simulation by the driver of the net, e.g., a primitive or a continuous assignment (Example: **wire** Y = A + B;).

Register variables: Store information and retain value until reassigned.

Value is established by an assignment made by a procedural statement.

Value is retained until a new assignment is made e.g., **reg**, **integer**, **real**, **realtime**, **time**

Example: **always @ (posedge** clock)
if (reset) q_out <=0;
else q_out <= data_in;

Procedural Statements

Describe logic abstractly; statements execute sequentially to assign value to variables

if (expression_is_true) statement_1; **else** statement_2;
case (case_expression)
case_item: statement ;
...
default: statement;
endcase
for (conditions) statement;
repeat constant_expression statement;
while (expression_is_true) statement;
forever statement;
fork statements **join** // execute in parallel

Assignments

Continuous: Continuously assigns the value of an expression to a net.

Procedural (Blocked): uses the = operator; statements execute sequentially; a statement cannot execute until the preceding statement completes execution. Value is assigned immediately.

Procedural (Nonblocking): uses the <= operator; executes statements concurrently, independent of the order in which they are listed. Values are assigned concurrently.

Procedural (Continuous):

assign ... deassign Overrides procedural assignments to a net.

force ... release Overrides all other assignments to a net or a register.

Operators

Operator	Description
{ }, {{ }}	concatenation
**	Exponentiation
+ − * /	arithmetic
%	modulus
> >= < <=	relational
!	logical negation
&&	logical and
||	logical or
==	logical equality (identity)
!=	logical inequality
===	case equality
!==	case inequality
~	bitwise negation
&	bitwise and
|	bitwise or
^	bitwise exclusive-or
^~ or ~^	bitwise equivalence
&	reduction and
~&	reduction nand
|	reduction or
~|	reduction nor
^	reduction exclusive-or
~^ or ^~	reduction xnor
<<	left logical shift
>>	right logical shift
<<<	left arithmetic shift
>>>	right arithmetic shift
?:	conditional
or	Event or

Specify Block

Example: Module Path Delays

```
specify
// specparam declarations (min: typ: max)
specparam t_r = 3;4:5, t_f = 4:5:6;
  (A, B) *> Y) = (t_r, t_f);        // full
  (Bus_1 => Bus_1) = (t_r, t_f); // parallel
  if (state == S0) (a, b *> y) = 2; // state dep
  (posedge clk => (y -: d_in)) = (3. 4);
endspecify
```

PDG

Example: Timing Checks

```
specify
specparam t_setup = 3:4:5, t_hold = 4:5:6;
$setup (data, posedge clock, t_setup);
$hold (posedge clock, data, t_hold);
endspecify
```

Memory: Declares an Array of Words

Example: Memory Declaration and Readout

```
module memory_read_display();
 reg [31: 0] mem_array [1: 1024];
 integer k;
 initial begin
 // read contents of mem_array from a file
   in hex format
 $readmemh ("mem_contents.dat"),
   mem_array);
 // display contents of mem_array
 for (k = 1; k<= 1024; k = k+1)
   $display ("word [%d] =
     %h",k,mem_array[k]);
 end
endmodule
```

Arrays

```
reg [7: 0] name [dim1 : 0] [dim2: 0] [dim3 : 0] ... name.
```

Multidimensional Arrays (Examples)

```
[15: 0] reg data [0: 27][0: 127];          // 2-dimensional
real time_array [0: 15][0: 15][0: 15];  // 3-dimensional
integer indices [7: 0][63: 0];             // 2-dimensional
```

Variable Part Select (See Appendix F)

Concurrency and Race Conditions

Indeterminate outcomes result from race conditions when multiple edge-sensitive behaviors use the procedural assignment operator (=) to reference (read) and assign value to the same variable at the same time.

Example (with race):

```
always @ (posedge clock) a = b;
always @ (posedge clock) b = a;
```

Use nonblocking assignment operator (<=) to eliminate race conditions; assignments are independent of the order of the behaviors and the order of the statements.

Example:

```
always @ (posedge clock) a <= b;
always @ (posedge clock) b <= a;
```

Use the procedural assignment operator (=) in level-sensitive behaviors and the nonblocking assignment operator (<=) in edge-sensitive behaviors to avoid race conditions in sequential machines.

Example (without race):

```
always @ (posedge clock)
 state <= next_state;
always @ (state, inputs)
 case (state)
   state_0: begin next_state = state_5; ...
            end
   ...
endcase
```

Functions

May invoke another function; may not invoke a task
Executes with zero delay time
May not contain delay control (**#**), event control (**@**), nonblocking assignment operator (<=), or **wait**.
Must have at least one input argument.
May not have **output** or **inout** arguments.
Function name serves as placeholder a single returned value

Example:

```
value = adder (a, b);
...
function [4:0] adder;
 input [3:0] a, b;

 adder = a + b;

endfunction
```

Tasks

May invoke other tasks and other functions
May contain delay control(#), event control (@), and wait
May have zero or more arguments having mode input, output, inout
Passes values by its arguments

Example:

```
adder (sum, a, b);
...
task adder;
 output [4: 0]        sum;
 input [3:0]          a, b;

 sum = a + b;
endtask
```

Selected Compiler Directives

```
'define    width = 16;
'include .../project_header.v
'timescale = 100 ns/1ns  // time_units / precision
'ifdef ... 'else ... 'endif
```

Example:

```
module or_model (y, a, b);
 output y;
 input a, b;
 'ifdef cont_assign  // uses continuous
   assignment
 assign y = a | b;
 'else
 or G1 (y, a, b);      // uses primitive
 'endif
endmodule
```

Simulation Output

```
$display ("string_of_info %d", variable);
integer K;
initial K = $fopen("output_file");
always @ (event_control_expression) // dump data
begin
 $fdisplay (K, "%h", data[7: 0]);
...
 $fclose ("output_file");
 $monitor ($time, "out_1 = %b  out_2 =
   %b", out_1, out_2);
 $monitor (K, "some_value = %h",
   address[15: 0]);
 $monitoron;
 $monitoroff;
end
```

Simulation Data Control

Example:

```
initial begin
// dump simulation data into my_data.dump
$dumpfile ("my_data.dump");
// dump all signals
$dumpvars;
// dump variables in module top.
$dumpvars (1,top);
// dump variables 2 levels below top.mod1;
$dumpvars (2,top.mod1);
// stop dump
#1000 $dumpoff;
// start or restart dump #500
$dumpon;
// suspend simulation
$stop;
// terminate simulation
#1000 $finish;
end
```

Parameter Redefinition

In-line (instance-based):

```
module Something ( );
 parameter size = 4;
 parameter width = 8;
 ...
endmodule
 ...
 Something #(8, 32) M1 ( );
 ...
```

Indirect (hierarchical dereferencing):

```
module Annotate ();
 ...
 defparpam .Something.width = 16;
...
```

Local Parameters (See Appendix F)

Value cannot be directly redefined from outside the module in which they are declared.
Keyword: **localparam**

Constructs to Avoid in Synthesis

time, **real**, and **realtime** variables
named **event**
triand, trior, tri0, tri1 nets
vector ranges for integers
single-pass (**initial**) behavior
assign ... **deassign** procedural continuous assignment
force ... **release** procedural continuous assignment
fork .. join block (parallel activity flow)
defparam parameter substitution
Operators: Modulus (%) and division(/)*,
===, !===
Primitives: **pullup**, **pulldown**, **tranif0**, **tranif1**,
rtran, **rtranif0**, **rtranif1**
Hierarchical pathnames
Compiler directives

*Except for division by a power of 2.