THE COMPLETE VERILOG BOOK

THE COMPLETE VERILOG BOOK

by

Vivek Sagdeo
Sun Micro Systems, Inc.

KLUWER ACADEMIC PUBLISHERS
NEW YORK, BOSTON, DORDRECHT, LONDON, MOSCOW

CD-ROM available only in print edition.
eBook ISBN: 0-306-47658-4
Print ISBN: 0-7923-8188-2

©2002 Kluwer Academic Publishers
New York, Boston, Dordrecht, London, Moscow

Print ©1998 Kluwer Academic Publishers
Dordrecht

All rights reserved

No part of this eBook may be reproduced or transmitted in any form or by any means, electronic, mechanical, recording, or otherwise, without written consent from the Publisher

Created in the United States of America

Visit Kluwer Online at: http://kluweronline.com
and Kluwer's eBookstore at: http://ebooks.kluweronline.com

To
My Parents,
Sons-Parth and Nakul,
Anjali,
Friends

LIST OF FIGURES

Figure 1-1. Block Diagram of a System with Processor, Main Memory, and Cache....... 8
Figure 1-2. Bottom-up Methodology and Verilog Language Features Support............. 14
Figure 1-3. Top-Down Methodology and Equivalent Verilog Language Features Support........ 14
Figure 1-4. Typical Design Flow with Verilog........ 15
Figure 1-5. Verilog Keywords........ 17

Figure 2-1. Tables of Net Types and Resolution Functions........ 25

Figure 3-1. Tables of Operators in Verilog Used for Evaluating Expressions........ 45
Figure 3-2. Schematics for the Adder in Example 3-28........ 64
Figure 3-3. Top Level Block Diagram of r4200........ 68
Figure 3-4. UltraSPARC-IIi Block Diagram........ 82

Figure 4-1. Schematics for Example 4-1........ 86
Figure 4-2. Network Data Structure for the andor Verilog Model in Example 4-1........ 91
Figure 4-3. Schedule of Events for a Verilog Model........ 93
Figure 4-4. Algorithm for Verilog Model Execution........ 93
Figure 4-5. Algorithm for Processing an Event........ 94
Figure 4-6. Order of Events at a Time and Event Structure Diagrams........ 94
Figure 4-7. Algorithm for Scheduling an Event........ 95

Figure 6-1. Tables for Each Built-in Gate in Verilog........ 136

Figure 11-1. State Diagram for Cache Controller with Write-Back Policy........ 217
Figure 11-2. Block Diagram for the Cache Controller with Write-Back Policy Containing Dirty Bits........ 222
Figure 11-3. State Transition Diagram for the Cache Controller with Write-Back Policy........ 223
Figure 11-4. Block Diagram for a Cache System........ 230

Figure 12-1. Typical Design Flow with Verilog Including Synthesis........ 245
Figure 12-2. Logic Synthesis Components of Verilog Based Synthesis........ 246
Figure 12-3. Components of Behavioral Synthesis with Verilog........ 247
Figure 12-4. Traditional View (Class A or Mealy Machine) of a Sequential Design........ 248
Figure 12-5. Modern View (Class A – Sagdeo Machine) of a Sequential Design........ 248

Figure 14-1. Sharings Adders Amongst Different Operations in Example 14-1........ 291

Figure 16-1. C Interface Components for Verilog HDL........ 313

Figure 17-1. Schematics for a Static RAM Cell with Bidirectionals and Strengths........ 317

PREFACE

The Verilog hardware description language (HDL) provides the ability to describe digital and analog systems. This ability spans the range from descriptions that express conceptual and architectural design to detailed descriptions of implementations in gates and transistors. Verilog was developed originally at Gateway Design Automation Corporation during the mid-eighties. Tools to verify designs expressed in Verilog were implemented at the same time and marketed. Now Verilog is an open standard of IEEE with the number 1364. Verilog HDL is now used universally for digital designs in ASIC, FPGA, microprocessor, DSP and many other kinds of design-centers and is supported by most of the EDA companies. The research and education that is conducted in many universities is also using Verilog. This book introduces the Verilog hardware description language and describes it in a comprehensive manner.

Verilog HDL was originally developed and specified with the intent of use with a simulator. Semantics of the language had not been fully described until now. In this book, each feature of the language is described using semantic introduction, syntax and examples. Chapter 4 leads to the full semantics of the language by providing definitions of terms, and explaining data structures and algorithms.

The book is written with the approach that Verilog is not only a simulation or synthesis language, or a formal method of describing design, but a complete language addressing all of these aspects. This book covers many aspects of Verilog HDL that are essential parts of any design process. It has the view of original development, and also encompasses changes and additions in subsequent revisions. The book starts with a tutorial introduction in chapter 1, then explains the data types of Verilog HDL in chapter 2. Today´s object-oriented world knows that the language-constructs and data-types are equally important parts of a programming language. Chapter 3 explains the three views of a design object: behavioral, RTL and structural. Each view is then described in detail, including the semantic introduction, example and syntax for each feature, in chapters 3, 5 and 6.

Verilog takes the divide and conquer approach to the language design by separating various types of constructs using different syntax and semantics. The syntax and semantics include features to describe design using the three levels of abstractions, features for simulation control and debug, preprocessor features, timing descriptions, programming language interface and miscellaneous system tasks and functions.

System tasks and functions that are useful for non-design descriptions, such as input-output, are described in chapters 8 and 10. The preprocessor enables one to define text substitutions and to include files, which are defined in chapter 9. The building of systems using all features is explained in chapter 11. Synthesis is an essential part of today´s design process, and Verilog HDL usage for synthesis requires special language understanding. The understanding needed is provided in chapters 11 to 13. Timing descriptions form a separate class of features in Verilog and are described in chapter 15. Chapter 17 describes how programming language interface (PLI) provides access to Verilog data structures and simulation information via common data definitions and routines. Standard Delay Format, which is discussed in chapter 18, extends capabilities of timing descriptions of specific blocks in Verilog, and is used in ASIC designs extensively. Chapter 19 enunciates the analog extensions to Verilog in the form of Verilog-A and Verilog-MS. Simulation speed is an important part of Verilog HDL usage, and a large part of the design cycle is spent in design verification and simulation. Some techniques to enhance this speed are discussed in chapter 20.

The book keeps the reader abreast of current developments in the Verilog world, such as Verilog-A, cycle simulation, SDF, DCL and uses IEEE 1364 syntax.

I hope that this book will be useful to all of those who are new to Verilog HDL, to those who want to learn additional facets, and to those who would like a reference book during the development of a hardware design or software tool with Verilog HDL. I wish for you to design and implement some interesting designs of ASICs, FPGAs, microprocessors, caches, memories, boards, systems and/or tools like simulators, synthesizers, timing analyzers, formal verifiers with Verilog HDL, and to have a lot of fun doing so.

-- Vivek Sagdeo

ACKNOWLEDGEMENTS

A book of this size takes many different things to come together . I would like to acknowledge Carl Harris of Kluwer for encouragement and for facilitating the creation of manuscript. Jackie Denfeld handled the creation of final manuscript in a short time well. Tedd Corman provided the editorial review and my experience of working with him in the past on simulation and HDLs has been valuable. Satish Soman provided feedback from the design perspective. UC Berkeley extension provided the teaching environment for me that has added the academic dimension to this book. Dr Richard Tsina, Joan Siau and Roxanne Giovanetti from UCB deserve mention for their support. Students of the class “Digital Design of Verilog HDL” from UCB and PerformancAE kept the book-writing interesting and live. My coworkers from SUN microsystems have been very cooperative and accomodating and have really good insight into digital design and microprocessors.

While working at Gateway Design where Verilog was designed and implemented, a terrific team was in place. Prabhu Goel, Barry Rosales, Manoj Gandhi, Phil Moorby, Ronna Alintuck and many from Marketing and Sales made this work on Verilog and well-rounded.

Over the several years, experiences of working at Gateway(Cadence), Viewlogic, Silicon Graphics, Meta Software, Philips Semi and SUN Microsystems and IEEE 1364 have provided the background to cover many aspects of Verilog including language, digital and analog, system and microprocessors and have given a perspective that has made this work possible.
I acknowledge all those whose names can’t be mentioned for lack of space but have been part of various projects with me.

DISCLAIMER

This DISK (CD ROM) is distributed by Kluwer Academic Publishers with *ABSOLUTELY NO SUPPORT* and *NO WARRANTY* from Kluwer Academic Publishers.

Use or reproduction of the information provided on this DISK (CD ROM) for commercial gain is strictly prohibited. Explicit permission is given for the reproduction and use of this information in an instructional setting provided proper reference is given to the original source.

Kluwer Academic Publishers shall not be liable for damage in connection with, or arising out of, the furnishing, performance or use of this DISK (CD ROM).

TABLE OF CONTENTS

1. INTRODUCTION TO VERILOG HDL 1

1.1 Language Motivation 1
1.1.1 Language Design 1
1.1.2 Verilog World 1
1.1.3 Accessory Specifications 2
1.2 Tutorial Via Examples 2
1.2.1 Counter Design 2
1.2.2 Factorial Generator 7
1.2.3 System Design with Processor, Memory, and Cache 8
1.2.4 Cache System - Behavioral Model 10
1.3 Overview of Verilog HDL 13
1.3.1 Correspondence To Digital Hardware 13
1.3.2 Typical Design Flow with Verilog 15
1.3.3 List of Keywords 17
1.3.4 Comment Syntax 17
1.4 Syntax Conventions 18
1.5 Exercises 19

2. DATA TYPES IN VERILOG 21

2.1 Overview 21
2.2 Value Systems 21
2.3 Data Declarations 22
2.3.1 Introduction 22
2.3.2 Examples 23
2.3.3 Syntax 23
2.4 Reg Declaration 23
2.4.1 Introduction 23
2.4.2 Examples 24
2.4.3 Syntax 24
2.5 Net Declaration 24
2.5.1 Introduction 24
2.5.2 Syntax 28
2.5.3 Examples 29
2.6 Port Types 29
2.6.1 Introduction 29
2.6.2 Examples 30
2.6.3 Syntax 30
2.7 Aggregates – 1 and 2 Dimensional Arrays (Vectors and Memories) 31
2.7.1 Introduction 31
2.7.2 Examples 31
2.7.3 Syntax 32

2.8 Delays on Nets 32
2.8.1 Introduction 32
2.8.2 Examples 32
2.8.3 Syntax 33
2.9 Integer and Time 33
2.9.1 Introduction 33
2.9.2 Examples 33
2.9.3 Syntax 33
2.10 Real Declaration 33
2.10.1 Introduction 33
2.10.2 Example 33
2.10.3 Syntax 34
2.11 Event Declaration 34
2.12 Parameter Declarations 34
2.13 Examples 34
2.14 Syntax 34
2.15 Hierarchical Names 35
2.15.1 Introduction 35
2.15.2 Examples 35
2.15.3 Syntax 35
2.16 Exercises 35

3. ABSTRACTION LEVELS IN VERILOG: BEHAVIORAL, RTL, AND STRUCTURAL 37

3.1 OVERVIEW 37
3.1.1 Introduction 37
3.1.2 Examples 37
3.1.3 Syntax 38
3.2 Behavioral Abstractions In Verilog 39
3.2.1 Introduction 39
3.2.2 Examples 39
3.2.3 Syntax 40
3.3 Register Transfer Level Abstractions in Verilog 40
3.3.1 Introduction 40
3.3.2 Example 41
3.3.3 Syntax 43
3.3.4 RTL Descriptions – Other Definitions 43
3.4 Expressions 43
3.4.1 Overview 43
3.5 Operators in Expressions 43
3.5.1 Introduction 43
3.5.2 Examples and Explanations 45
3.5.3 Operators in Expressions – Syntax 50
3.6 Operands in Expressions 51
3.6.1 Introduction 51
3.6.2 Examples and Explanations 51
3.6.3 Syntax of Operands in Expressions 53
3.7 Special Considerations in Expressions 55
3.7.1 Constant-Valued Expressions 55
3.7.2 Operators on Reals 55

3.7.3 Operator Precedence 55
3.7.4 Examples of Various Operator Usage 56
3.7.5 Comparisons With X's and Z's 56
3.7.6 Expression Bit Lengths 57
3.8 Syntax for Expressions 57
3.9 Example of Register Transfer Level of Abstraction 59
3.10 Structural Descriptions In Verilog 63
3.10.1 Structural Constructs – Overview 63
3.10.2 Structural Constructs - Module Definitions 64
3.10.3 Structural Constructs – Module Instantiation 67
3.11 Exercises 83

4. SEMANTIC MODEL FOR VERILOG HDL 85

4.1 Introduction 85
4.2 Example 86
4.3 Simulation with Full Analysis 87
4.4 Log of a Typical Simulator 87
4.5 Log of an Ideal Simulator 88
4.6 Analysis and Concepts in Event-Driven Simulation in Verilog 90
4.7 Internal Data Structure Representation 90
4.8 Update and Evaluate Events 90
4.9 Order of Execution of Events in Verilog 91
4.10 Algorithm for Event-Driven Model for Verilog HDL 92
4.10.1 Definitions 92
4.10.2 Algorithm 93
4.11 Highlights of the Algorithm – Concurrent Processes and Event Cancelations 95
4.11.1 Concurrent Processes 95
4.11.2 Event Cancelations 96
4.12 Exercises 98

5. BEHAVIORAL MODELING 99

5.1 Overview of Behavioral Modeling 99
5.1.1 Introduction 99
5.1.2 Examples 100
5.1.3 Syntax 100
5.2 Procedural Assignments 101
5.2.1 Overview 101
5.2.2 Blocking (Immediate) Procedural Assignments 101
5.2.3 Non-Blocking Procedural Assignments 102
5.2.4 Examples Comparing Blocking and Non-Blocking Assignments 104
5.3 Conditional Statement 107
5.3.1 Overview 107
5.3.2 Examples 107
5.3.3 Syntax 107
5.3.4 Special Considerations 107
5.4 Case Statement 108
54.1 Overview 108
5.4.2 Examples 108
5.4.3 Syntax 108

5.4.4 Don't Cares and Case Statements – Casex and Casez 109
5.4.5 Examples Comparing Case, Casex, and Casez 109
5.5 Loops 111
5.5.1 Overview 111
5.5.2 Examples 111
5.5.3 Syntax 112
5.6 Begin-End Blocks 112
5.6.1 Introduction 112
5.6.2 Example 113
5.6.3 Syntax 113
5.7 Wait Statements 113
5.7.1 Introduction 113
5.7.2 Example 113
5.7.3 Syntax 113
5.8 Event and Delay Controls 114
5.8.1 Overview 114
5.8.2 Event Declarations 114
5.8.3 Multi-Event Event – Event OR 115
5.8.4 Event Usage 115
5.8.5 Event Generalization 116
5.8.6 Generalized Event Transitions 116
5.9 Fork-Join Blocks 117
5.9.1 Introduction 117
5.9.2 Examples 117
5.9.3 Syntax 118
5.9.4 Special Considerations – Modeling Pipelines 118
5.10 Functions and Tasks 119
5.10.1 Functions 119
5.10.2 Tasks 121
5.11 Task Disabling 122
5.11.1 Introduction 122
5.11.2 Examples 122
5.11.3 Syntax 123
5.12 Assign-Deassign Statements 123
5.12.1 Introduction 123
5.12.2 Example 123
5.12.3 Syntax 124
5.13 Force-Release Statements 124
5.13.1 Introduction 124
5.13.2 Examples 124
5.13.3 Syntax 125
5.14 A Behavioral Modeling Example – An Essential Microprocessor 125
5.15 Exercises 132

6. RTL AND STRUCTURAL MODELING 135

6.1 Introduction 135
6.2 Gates 135
6.2.1 Introduction 135
6.2.2 Example 136
6.2.3 List of Gates and Their Functions 136

6.2.4 Syntax for Gates and Switch Declarations 136
6.3 Switches 138
6.3.1 Introduction 138
6.3.2 Syntax 138
6.4 User-Defined Primitives 138
6.4.1 Introduction 138
6.4.2 Examples 139
6.4.3 Syntax 139
6.5 Combinational UDPs 141
6.5.1 Introduction 141
6.5.2 Example 141
6.5.3 Syntax 142
6.6 Level-Sensitive Sequential UDP 142
6.6.1 Introduction 142
6.6.2 Example 143
6.6.3 Syntax 143
6.7 Edge Sensitive Sequential UDPs 143
6.8 Mixed Level and Edge Sensitive Sequential UDPs 144
6.9 UDP Instances 145
6.9.1 Introduction 145
6.9.2 Example 146
6.9.3 Syntax for Primitive Instance 146
6.10 Exercises 146

7. MIXED STRUCTURAL, RTL, AND BEHAVIORAL DESIGN 151

7.1 Introduction 151
7.2 Examples and Scenarios: 1 – Comparing Structural Adder Design with Behavioral Model 151
7.3 Examples and Scenarios: 2 – System Modeling 152
7.4 Examples and Scenarios: 3 – Adding Behavioral Code to a Design for Checking 153
7.5 Examples and Scenarios: 4 – Design Cycle and Project Planning Flexibility 154
7.6 Exercises 154

8. SYSTEM TASKS AND FUNCTIONS 155

8.1 Introduction 155
8.2 Display System Tasks 156
8.2.1 Overview 156
8.2.2 Examples 156
8.2.3 Syntax and Format Details 157
8.3 Monitor System Tasks 158
8.3.1 Overview 158
8.3.2 Examples 159
8.3.3 Syntax 160
8.4 File Management 160
8.4.1 Overview 160
8.4.2 Examples 160
8.4.3 Syntax 161

8.5 File Input Into Memories 161
8.5.1 Overview 161
8.5.2 Example 161
8.5.3 Syntax 162
8.6 Simulation Time Functions 162
8.7 Simulation Control Tasks 162
8.7.1 Overview 162
8.7.2 Examples 162
8.8 Waveform Interface (VCD Files) 163
8.8.1 Overview 163
8.8.2 Examples 163
8.8.3 Syntax 165
8.9 Exercises 165

9. COMPILER DIRECTIVES 167

9.1 Introduction 167
9.2 'include 168
9.2.1 Introduction 168
9.2.2 Example 168
9.2.3 Syntax 168
9.3 `define and `undef 168
9.3.1 Introduction 168
9.3.2 Examples 168
9.3.3 Syntax 169
9.4 `ifdef, `else, `endif 169
9.4.1 Example 169
9.4.2 Syntax 170
9.5 `default_nettype 170
9.5.1 Example 170
9.5.2 Syntax 170
9.6 `timescale 170
9.6.1 Example 170
9.6.2 Syntax 171
9.7 `resetall 171
9.8 Exercises 171

10. INTERACTIVE SIMULATION AND DEBUGGING 173

10.1 Introduction 173
10.2 System Tasks and Functions 173
10.2.1 Previously Covered 173
10.2.2 Additional Tasks 174
10.3 Commands 174
10.4 Browser Tools 174
10.5 Code Coverage 174

11. SYSTEM EXAMPLES 177

11.1 Introduction 177
11.2 Example 1: 8085 Based System: Sio85.V 177
11.3 Example 2: R4200 183
11.4 Example 3: Cache Design 215

11.4.1 Cache System: Architecture with State Diagram ... 215
11.4.2 Cache System: Behavioral Model with Write-Through Policy ... 217
11.4.3 Cache System: Behavioral Model Modified for Write-Back Policy ... 222
11.4.4 Cache System: Implementation: Write-Through Policy ... 230
11.5 Memory Model with Bus Cycle Timing and with Timing Checks ... 236
11.6 Exercises ... 242

12. SYNTHESIS WITH VERILOG ... 243

12.1 Logic Synthesis and Behavioral Synthesis ... 243
12.2 Design Flow with Synthesis ... 243
12.2.1 Typical Design Flow with Verilog ... 243
12.3 Logic Synthesis View ... 247
12.4 Examples ... 249
12.5 Exercises ... 253

13. VERILOG SUBSET FOR LOGIC SYNTHESIS ... 255

13.1 Introduction ... 255
13.2 Structural Descriptions – Modules ... 256
13.3 Declarations – Overview ... 257
13.4 Module Items – Overview ... 257
13.5 Synthesizing Net Types ... 257
13.6 Continuous Assignments ... 258
13.7 Module Instantiations – Parametrized Designs ... 258
13.8 Structural Descriptions – Gate-Level Modeling ... 259
13.9 Expressions ... 261
13.10 Behavioral Modeling for Synthesis ... 262
13.11 Function Declarations ... 264
13.11.1 Data Declarations in Functions – Reg, Input, Memory, Parameter, and Integers ... 264
13.12 Behavioral Statements Support for Logic Synthesis – Overview ... 265
13.13 Procedural Assignments ... 266
13.14 if Statement ... 267
13.15 Conditional Assignments ... 268
13.16 Case Statements ... 268
13.17 For Loops ... 270
13.18 Forever Loops ... 272
13.19 Disable Statements ... 272
13.20 Task Statements ... 273
13.21 Always Blocks ... 274
13.22 Incomplete Event (Sensitivity) Specification ... 276
13.23 Exercises ... 277

14. SPECIAL CONSIDERATIONS IN SYNTHESIZING VERILOG ... 279

14.1 Inferring Registers ... 279
14.1.1 Introduction ... 279
14. 1.2 Latch Inference ... 279
14. 1.3 Simple Flip-Flop Inference ... 281
14. 1.4 Modeling Flip-Flops with Resets ... 282
14. 1.5 Synthesis Checks During Register Inference ... 283
14. 1.6 Bus Latch ... 286

14.2 Multiplexers 287
14.3 Three-State Inference 287
14.3.1 Modeling a Tri-State Gate for Synthesis 287
14.3.2 Modeling Two Three-State Gates 288
14.3.3 Registered or Latched Three-State 288
14.3.4 Three-State with Registered Enable 289
14.4 Designs via Resource Sharing 290
14.4.1 Introduction 290
14.4.2 Sharable Resources 290
14.5 Control Flow and Data Flow with Sharing 292
14.5.1 Data Flow Conflicts 292
14.5.2 Resource Area 293
14.5.3 Resource Sharing Methods 293

15. SPECIFY BLOCKS — TIMING DESCRIPTIONS 295

15.1 Overview 295
15.2 Example 295
15.3 Specify Blocks – Syntax 296
15.4 Timing Checks in Specify Blocks 296
15.4.1 $Setup Timing Check 297
15.4.2 $Hold 297
15.4.3 $Width 297
15.4.4 $Period 298
15.4.5 $Skew 298
15.4.6 $Recovery 298
15.4.7 $Setuphold 298
15.4.8 Example of Timing Checks 299
15.5 Module Path (Delay) Declarations 300
15.5.1 Introduction 300
15.5.2 Simple Paths 300
15.5.3 Edge-Sensitive Paths 301
15.5.4 State-Dependent Paths 301
15.5.5 Edge-Sensitive State-Dependent Paths 303
15.5.6 State-Dependent Paths – ifnone Condition 303
15.6 Delay Specifications 304
15.7 Mixing Distributed and Specified Delays 305
15.8 Multi-Driver Nets 305
15.9 Pulse Specification 305
15.10 Exercises 305

16. PROGRAMMING LANGUAGE INTERFACE 307

16.1 Overview and Examples 307
16.2 PLI Origin and Use 308
16.3 PLI Function Types 308
16.4 PLI Interface Classes 309
16.5 Interface Definitions 309
16.5.1 veriuser.h 309
16.5.2 acc_user.h 310
16.5.3 vpi_user.h 311
16.6 User Tasks and Functions 312

16.7 Steps Involved in Defining User Tasks/Functions 312
16.8 C Interface Components 313
16.9 Verilog Callbacks – Utility Routines 313
16.10 Verilog Callbacks – Access Routines 313
16.11 Verilog Callbacks – VPI Routines 313
16.12 Exercises 313

17. STRENGTH MODELING WITH TRANSISTORS 315

17.1 Overview 315
17.2 Modeling with Unidirectional Switches – Example 316
17.3 Modeling with Bidirectionals and Strengths – Example 317
17.4 Strength-Levels in Verilog 319
17.4.1 Overview 319
17.4.2 Examples of Strength Algebra 319
17.4.3 Strength Specifications On Gates 320
17.5 Exercises 320

18. STANDARD DELAY FORMAT 321

18.1 Introduction 321
18.2 SDF Description 324
18.2.1 Introduction 324
18.2.2 Syntax 324
18.2.3 Example 325
18.3 Header 325
18.3.1 Introduction 325
18.3.2 Syntax 325
18.4 Header SubParts 325
18.5 Cell Entry 328
18.5.1 The CELLTYPE entry 329
18.5.2 The Cell Instance Entry 329
18.6 Timing Specifications 330
18.6.1 Delay Type – Absolute 331
18.6.2 The INCREMENT keyword 331
18.6.3 The PATHPULSE Entry 332
18.6.4 The PATHPULSEPERCENT Entry 333
18.7 Delay Definitions 333
18.7.1 The Delay Data 334
18.7.2 Delay Value 335
18.7.3 The IOPATH Entry 336
18.7.4 Conditionals 337
18.7.5 The RETAIN Entry 338
18.7.6 The PORT Entry 339
18.7.7 The INTERCONNECT Entry 339
18.7.8 The DEVICE Entry 340
18.8 Timing Check Entries 341
18.8.1 The SETUP Entry 344
18.8.2 The HOLD Entry 344
18.8.3 The SETUPHOLD Entry 345
18.8.4 The RECOVERY Entry 345
18.8.5 The REMOVAL Entry 346

18.8.6 The RECREM Construct ... 347
18.8.7 The SKEW Entry ... 347
18.8.8 The WIDTH Entry ... 348
18.8.9 The PERIOD Entry ... 348
18.8.10 The NOCHANGE Entry ... 349
18.9 Timing Environment and Constraints ... 350
18.9.1 The PATHCONSTRAINT Entry ... 350
18.9.2 The PERIODCONSTRAINT Construct ... 351
18.9.3 The SUM Entry ... 352
18.9.4 The DIFF Constraint ... 352
18.9.5 The SKEWCONSTRAINT Entry ... 353
18.10 Timing Environment – Information Entries ... 353
18.10.1 The ARRIVAL Construct ... 354
18.10.2 The DEPARTURE Construct ... 354
18.10.3 The SLACK Construct ... 355
18.10.4 The WAVEFORM Construct ... 357
18.11 SDF File Examples ... 359
18.12 Delay Model ... 360
18.12.1 Introduction ... 360
18.12.2 The List of Delay Models ... 361
18.12.3 Rules for the Delay Model ... 361
18.13 DCL – New Emerging Standard ... 362
18.14 OMI Standard ... 363

19. VERILOG-A AND VERILOG-MS ... 365

19.1 Analog Module ... 365
19.1.1 Introduction ... 365
19.1.2 Examples ... 365
19.1.3 Syntax ... 366
19.2 Analog Data Declarations ... 366
19.2.1 Introduction ... 366
19.2.2 Examples ... 366
19.2.3 Syntax ... 366
19.3 Analog Behavioral Descriptions ... 367
19.3.1 Introduction ... 367
19.3.2 Examples ... 367
19.3.3 Syntax ... 367
19.4 Expressions in Analog Assignments ... 368
19.5 Mixed Signal Designs in Verilog ... 368

20. SIMULATION SPEEDUP TECHNIQUES ... 371

20.1 Cycle-Based Simulation ... 371
20.2 2-State Versus 4-State Simulations ... 372
20.3 Compiled, Native Code, and Interpretive Simulations ... 372
20.4 Parallel Processors and Multi-Threaded Simulators ... 372
20.5 Usage of Caches and Other Memory to Achieve Speedup ... 373
20.6 Distributed Simulations Over a Network of Workstations ... 373
20.7 C Code Versus HDL Code ... 373
20.8 File Management in Simulation ... 374

A. FORMAL SYNTAX DEFINITION FOR VERILOG HDL 375

A.1 Source Text 376
A.2 Declarations 377
A.3 Primitive Instances 379
A.4 Module Instantiations 380
A.5 UDP Declaration and Instantiation 381
A.6 Behavioral Statements 383
A.7 Specify Section 385
A.8 Expressions 390
A.9 General 392

B. VERILOG SUBSET FOR LOGIC SYNTHESIS 395

B.1 Introduction 395
B.2 Syntax for Verilog for Logic Synthesis 395
B.3 Ignored Constructs for Logic Synthesis 403
B.3.1 Compiler Directives 403
B.3.2 Verilog System Functions 403
B.4 Unsupported Verilog Language Constructs 403
B.4.1 Unsupported Definitions and Declarations 403
B.5 Verilog Keywords Set for Logic Synthesis 404

C. PROGRAMMING LANGUAGE INTERFACE (PLI), HEADER FILE – veriuser.h 407

D. PROGRAMMING LANGUAGE INTERFACE (PLI) HEADER FILE – acc_user.h 419

E. PROGRAMMING LANGUAGE INTERFACE (PLI), HEADER FILE – vpi_user.h 431

F. FORMAL SYNTAX DEFINITION OF SDF 445

G. LIST OF EXAMPLES 451

H. REFERENCES 457

INDEX 459

1 INTRODUCTION TO VERILOG HDL

1.1 Language Motivation

1.1.1 Language Design

The complexity of hardware design has grown exponentially in the last decade. The exponential growth is fueled by new advances in design technology as well as the advances in fabrication technology. The usage of hardware description language to model, simulate, synthesize, analyze, and test the design has been a cornerstone of this rapid development. Verilog is the first hardware description language that was designed to be the language of choice in this domain. It is designed to enable descriptions of complex and large designs in a precise and succinct manner. It can facilitate descriptions of advances in architectures of design such as pipelining, cache management, branch predictions. A smooth top-down design flow is possible with Verilog based designs. It is also designed to facilitate new ECAD technologies such as synthesis and formal verification and simulation. It was designed to unify design process (including behavioral, rtl, gates, stimulus, switches, user-interface, test-benches, and unified interactive and batch modes). It is designed to leverage advances in software development for hardware design.

1.1.2 Verilog World

Verilog HDL has been successfully applied in all the major accomplishments in the field of digital design in the last decade. It is a language that is today IEEE standard 1364, is open and has activities under Open Verilog International umbrella, has annual International Verilog Conference (IVC) and is used in a vast majority of electronics and computer industry projects as well as research in academics in areas such as formal verification and behavioral synthesis. It also has spawned an industry

of CAD tool vendors and consulting/support experts creating a movement to participate in the world of electronics today.

1.1.3 Accessory Specifications

Verilog's accessory specifications such as the Programming Language Interface and the Standard Delay Format (SDF) enable a highly productive design management environment. Verilog HDL is a powerful tool to add to the repertoire of anybody involved with designing circuits in digital and now analog domains.

1.2 Tutorial Via Examples

In the following pages of Chapter 1, we will take examples of circuits and see their Verilog descriptions. This will give us a quick tour of the hardware description language which is explained fully in the following chapters along with the digital design techniques developed with Verilog.

1.2.1 Counter Design

Traditionally a counter is designed with flip-flops and gates. The flip-flops in turn are designed with gates. To test the counter we connect clock and reset signals. Verilog retains the capability of describing structural level descriptions, as shown below, and adds the register transfer level and the behavioral capabilities over traditional methods of design. These abstraction capabilities can be seen in the following models and in comparing the traditional methods versus the Verilog approach.

Example 1-1 describes the gate-level description of a D edge-triggered flip-flop. The schematics are shown in Example 1-3.

```
module d_edge_ff_gates(q, qBar, preset, clear, clock, d);
    inout q, qBar;
    input clock, d, preset, clear;

    nand #1 nl (ol, preset, o4, o2),
        n2 (o2, clear, clock, ol),
        n3 (o3, clock, o2, o4),
        n4 (o4, d, o3, clear),
        n5 (q, preset, o2, qBar),
        n6 (qBar, q, o3, clear);

endmodule
```

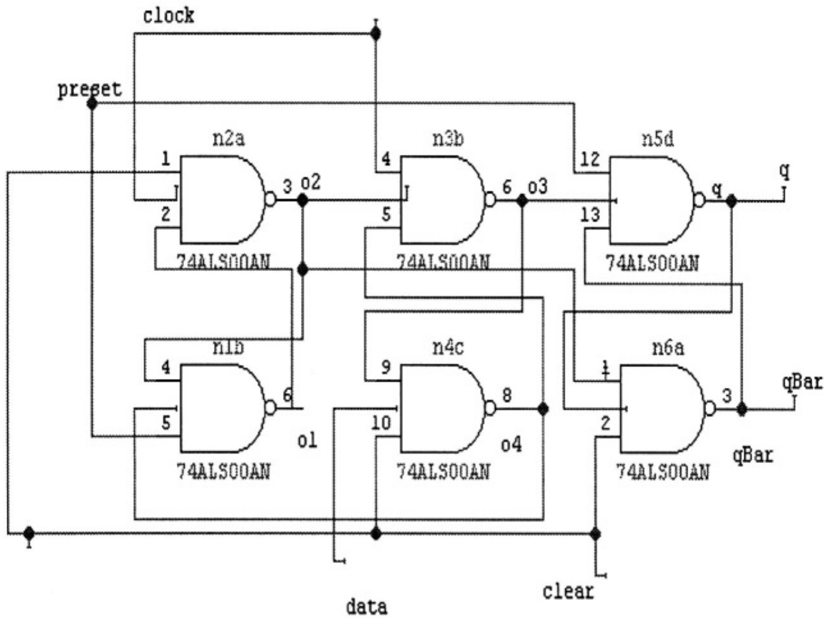
Example 1-1. A gate-level description of edge-sensitive d flip-flop.

```
module counter(q, clock, preset, clear);
    output [3:0] q;
    input clock, preset, clear;
    d_edge_ff_gates dffl(q[0], qBar0, preset, clear, clock, qBar0),
     dff2(q[l], qBarl, preset, clear, qBar0, qBarl),
```

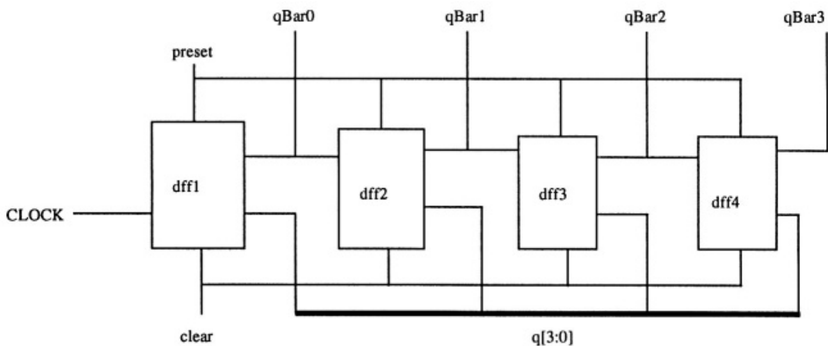
```
        dff3(q[2], qBar2, preset, clear, qBarl, qBar2),
        dff4(q[3], qBar3, preset, clear, qBar2, qBar3);

// initial $monitor("Internal counter signals qb0=%d qbl=%d qb2=%d qb3=%d",
//                                  qBar0, qBar1, qBar2, qBar3);
endmodule
```

Example 1-2. A 4-bit counter built using instances of d flip-flop defined in Figure 1-1.



Example 1-3. Schematics for dff in 1-1.



Example 1-4. Schematic for counter in 1-2.

The description in Example 1-1 begins with the word “module” and ends with the word “endmodule”. The interface to the module is described in the same line as module name “d_edge_ff_gates”. The direction of each port in the interface list is described in the following lines beginning with the words like “inout” and “input”. The “nand” statement has six instances of nand gates with names nl through n6. The interface list on each line enclosed in parentheses. The first identifier describes the output and the subsequent identifiers describe the inputs of each nand gate. Thus, ol through o4 and q and qBar are outputs of the gates nl through n6; preset, clear, clock and data are inputs along with ol-o4, q and qBar which are in the feedback loop. The “#” symbol indicates delay on the gate which is a unit delay (1 unit) in this case.

Example 1-2 builds a 4-bit counter built with d flip-flops defined in Example 1-1. The flip-flop was built using predefined nand gate while the counter is built hierarchically using a module defined earlier. Again, the definition of this block is enclosed between the keywords “module” and “endmodule” and the interface list is described at the top of the module. The four flip-flops are instantiated using the name of the module “d_edge_ff_gates” followed by names (dffl-dff4) and the connection list.

The definition of the counter output q specifies the 4-bit output by using [3:0] expression. This indicates the size of this bit-vector of size 4 and indices from 3 down to 0. Verilog supports single-bit quantities or scalars and multi-bit or vectors. Bits in Vectors are addressed using brackets, as in q[0] indicating bit 0 is vector q.

```
module counter_behav(q, clock, preset, clear);
      output  [3:0]  q;
      reg    [3:0] q;
      input clock, preset, clear;

      always  @(posedge clock)
      begin
           if( (preset = = 1) && (clear = = 1))
           q  =q + 1;
           else
            if ((preset = = 0) && (clear = = 1))
                 q = 4'b1111;
            else
                 q = 0;
      end
endmodule
```

Example 1-5. Behavioral description of the same counter as in 1-2.

In the Example 1-5, the same counter is described at a higher level of abstraction, known as behavioral level. Here there are no flip-flops or gates, but an always block, that is sensitive to “posedge”, a positive or rising clock edge. Inside this block, we see that the count increments by one (“q=q+l”) when preset and clear are inactive

(both 1). The preset and clear actions are modeled in the next two statements whereby q is set to 4’b1111 or 0 under right combinations for preset and clear values.

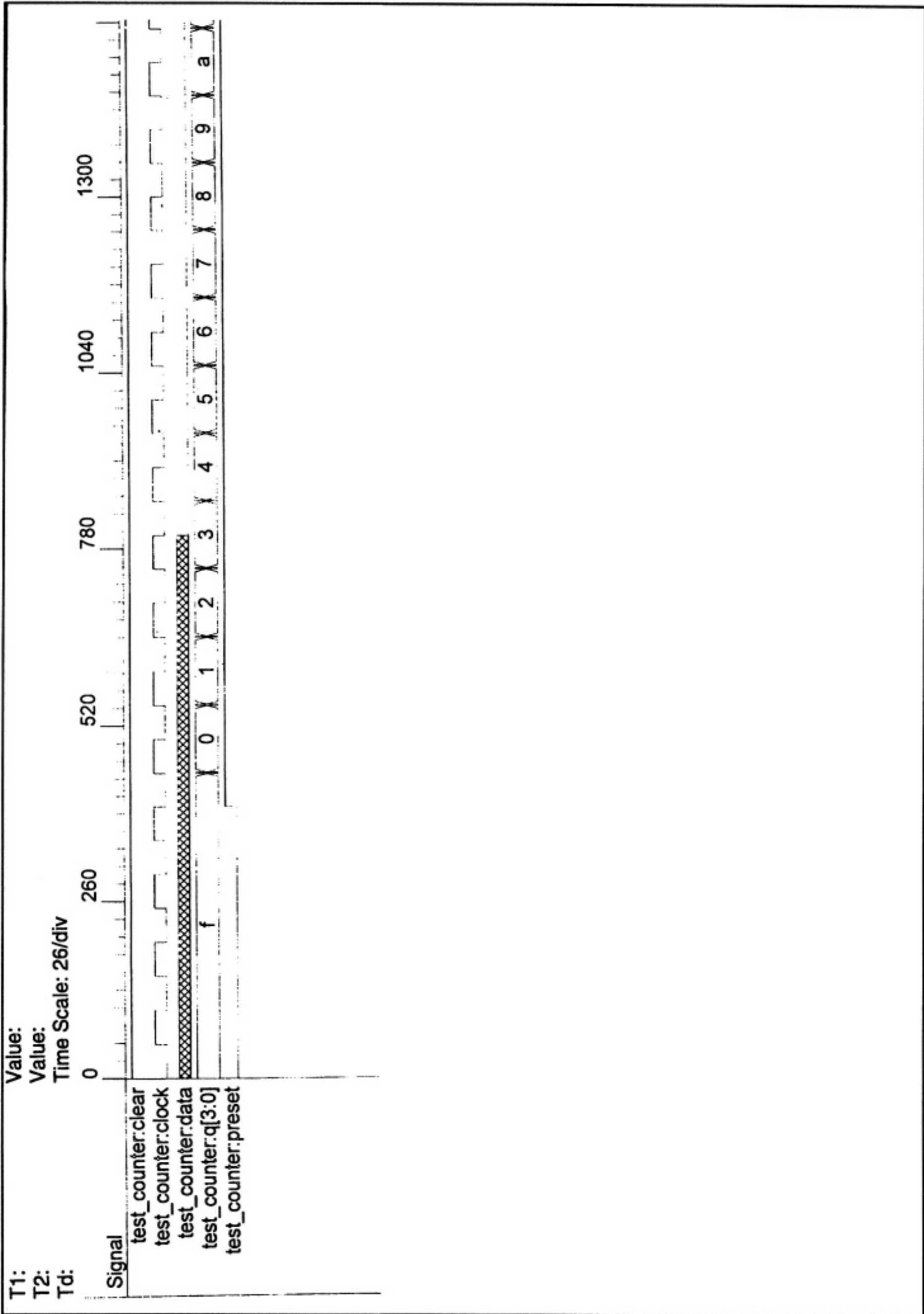
The always block executes as a loop with ‘@’ symbol indicating a wait on the event described in the expression that follows; in this case the ‘posedge clock’ or rising edge of clock. Another name used for such a loop is ‘process’. In a synchronous system, several processes that execute based on clock-edges and resets can describe the synchronous behavior fully.

```
module test_counter;
      reg preset, clear, clock, data;
      wire [3:0] q;
      counter ci(q, clock, preset, clear);
      counter_behav ci(q, clock, preset, clear);
      initial
      begin
            clock = 0;
            forever #50 clock = ~clock;
      end

initial
      begin
            $monitor("time=%d preset=%d clear=%d clock=%d q[0]=%d q[1]=%d
q[2]=%dq[3]=%d",
                  $time, preset, clear, clock, q[0], q[1], q[2], q[3]);
            preset = 0;
            clear = 1;
            #200
/*          preset = 1;
            clear = 0;
*/          #200
            preset = 1;
            clear = 1;
            #200      ;
            #200
            data = 0;
            #1600
            $finish;
      end
endmodule
```

Example 1-6. ***A test module for testing the two descriptions of counter and their equivalence.***

Here, in Example 1-6, a test-bench is built by instantiating the two modules and with two initial blocks. The 'counter' module, with gate-level description, and counter-behav, with behavioral description, are connected to the same inputs, and their outputs are connected together. The first initial block generates clock. The second initial block generates stimulus. It also monitors changes in inputs or outputs



Example 1-7. Waveforms for the counter example.

using $monitor system task. The waveforms are produced as shown in Example 1-7 in a simulation run, while the text output is shown in Example 1-8.

1.2.2 Factorial Generator

```
module factorial(n,  fact);
      nput [31:0] n;
      utput[31:0]fact;
      eg [31:0]  fact;
      eg factReady;
      nteger i;
      initial begin
            factReady = 0;
            fact = 0;
            $monitor("time=%d number =%d factorial = %d", $time, n, fact);
      end

      always@n
      begin
            factReady = 0;

            fact = 1;
            for(i=l;i<=n;i=i+l)
                  fact = fact * i;
            factReady = 1;
      end
endmodule

module test;
      reg[31:0]n;
      wire [31:0] fact;
      integer j;
            reg nReady;
                  factorial f (n, fact);
            initial
            begin
            #1 n =1;
            for(j=l;j<=4;j=j+l)
            begin
                  nReady = 0;

                  #1n=j;
                  nReady = 1;
            end
            end
endmodule
```

Example 1-8. ***Factorial generation of a number.***

In Example 1-8, the factorial module generates the factorial of a number algorithmically. This design module instantiated in a test module and the two modules communicate via the ports n and fact. The module factorial contains an 'always' block that executes based on an event on n. The ‘for’ loop computes the value of the factorial using the loop variable i and the limiting value n. The test module contains an initial block that generates the stimulus for the factorial. Numbers are generated in a for loop that has a delay of 1 unit for each number generated. The factorial computation is zero-delay and the computation is completed before the test block generates the next number. This example illustrates pure behavioral modeling.

1.2.3 System Design with Processor, Memory, and Cache

In the next example, we see a hierarchical building of a system using module definitions and instantiations.

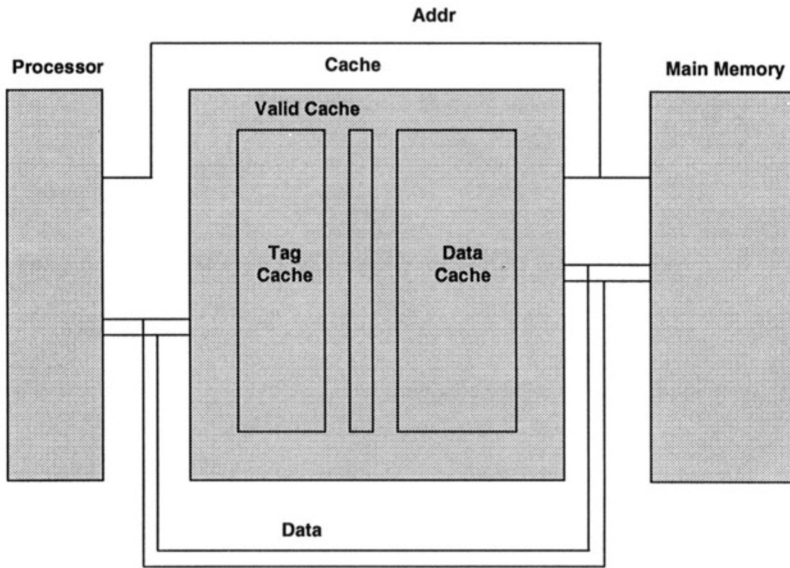


Figure 1-1. Block Diagram of a System with Processor, Main Memory, and Cache

```
module Processor(procRead, procWrite, procAddress, procData, procClock, reset);
        output procRead, procWrite;
        inout [`ADDR_SIZE-l:0]procAddress;
        inout [`DATA_SIZE-l:0] procData;
        input  procClock;
        input reset;
        // processor description
```

```
    //
endmodule

module MainMemory(memRead, mem Write, memAddress, memData, memClock,reset);
    input memRead, mem Write, memClock, reset;
    input [`ADDR_SIZE-1:0] memAddress;
    input [`DATA_SIZE-1:0] memData;
    // memory description
    //
endmodule
`endif

module Cache(procRead, procWrite, procAddress, procData,
                memRead, mem Write, memAddress, memData, reset, clock);
    input procRead, procWrite, reset, clock;
    input [`ADDR_SIZE-l:0] procAddress;
    output memRead, memWrite;
    output [`ADDR_SIZE-1:0] memAddress;
    inout [`DATA_SIZE-l:0]  memData, procData;

    wire [`DATA_SIZE-1:0] dataIn, outData, dataOut;
    wire [`TAG_SIZE-1:0] tagOut;

    tagCache tc(procAddress, tagOut, clock, write, procRead, reset);
    validCache vc(procAddress, valid, clock, write, procRead, reset);
    dataCache dc(procAddress, dataln, dataOut, clock, write, read);
    comparator c(tagOut, procAddress[`ADDR_SIZE-l:`ADDR_SIZE-`TAG_SIZE],
            match);
    cacheControl cc(procRead, procWrite, match, valid, read, write, mem Write,
            memRead, dataOutSel, dataInSel, clock, reset);
    dataMux dmIn(procData, memData, dataInSel, dataIn);
    dataMux dmOut(dataOut, memData, dataOutSel, outData);
endmodule

module System();
    wire [`ADDR-SIZE-l:()] memAddress;
    wire [`DATA_SIZE-l:0] memData;
    wire [`ADDR_SIZE-l:0] procAddress;
    wire [`DATA_SIZE-l:0] procData;
    Processor p(procRead, procWrite, procAddress, procData, procClock,
            eset);
    MainMemory m(memRead, memWrite, memAddress, memData, memClock,
            reset);
    Cache c(procRead, procWrite, procAddress, procData,
            memRead, memWrite, memAddress, memData, reset, clock);
endmodule
```

Example 1-9. A system model with microprocessor, ram, and cache controller.

In Example 1-9, three blocks in hardware, the processor, the memory and the cache are modeled with their interfaces, in the three modules named Processor, Memory and Cache. The module System instantiates all three blocks and connects the signals to the modules, thereby creating the network. The data, address buses on the processor side are procData and procAddr and on the memory side are memData and memAddr. The cache block connects the two with its mapping function for address and data on either side. This is modeled in the module cache upto one level of hierarchy. Full details of the subsequent levels are described in Example 11-5. This is an example of structural style modeling in Verilog using module definitions and instantiations, while functionality is explained in Example 11-5.

1.2.4 Cache System - Behavioral Model

```
// 2 K Cache

    `define CACHE-SIZE 2*1024
    // This is limited by maximum size in this simulator
    `define MEM_SIZE 128*1024

    `define ADDR_SIZE  17
    `define TAG_SIZE  6
    //define  states
    `define IDLE 0
    `define READ 1
    `define WRITE 2
    `define READ_MISS 3
    `define  READ_CACHE 4
    `define WRITE_MISS 5
    `define WRITE_CACHE 6

    `define DATA_SIZE 64
    `define CP  100
    `define CACHE_DRV 1
    `define NONCACHE_DRV 0

    module cache(reset, addr, data, read, write, clock, buscntrl, done);
        input  [`ADDR_SIZE-1:0]  addr;
        inout  [`DATA_SIZE-1:0]  data;
        input read, write, clock, reset;
        input buscntrl;
        output done;    // indicates completion of cache operation

        reg [`TAG_SIZE-1:0] tagCache[`CACHE_SIZE-l:0];
        reg [`ADDR_SIZE-`TAG_SIZE-1:0] dataCache[`CACHE_SIZE-l:0];
        reg [`CACHE_SIZE-l:0] validCache;
        reg match;
        reg [7:0] state;
        integer i;
        integer index, tag;
```

```
        reg [`DATA_SIZE-l:0] MainMemory[0:`MEM_SIZE-1];
        reg [`DATA_SIZE-l:0] dataOut;

        reg done;

                assign data = (buscntrl== `CACHE_DRV) ? dataOut: 'bz;

    function get_caching_scheme;
    input [`ADDR_SIZE-1:0] addr;
    begin
        get_caching_scheme = 1;
    end
    endfunction

        initial
        for (i=0; i <`CACHE_SIZE; i=i+l)
            validCache[i] = 0;

        always @(read or write)
        begin
        done = 0;

        if(read)
            state = `READ;
        else
        if (write)
            state = `WRITE;

        while (state != `IDLE)
        begin
            @(posedge clock)
            if (reset)
            begin
                        // Clear all validCache bits
                for (i=0; i <`CACHE_SIZE; i=i+l)
                    validCache[index] = 0;
            end
            else
            begin
                index = addr[`ADDR_SIZE-`TAG_SIZE-1:0];
                tag = addr[` ADDR_SIZE-1: `ADDR_SIZE-`TAG_SIZE];
                if ((validCache[index]) &&(tagCache[index] == tag))
                    match = 1;
                else
                    match = 0;

                case(state)
```

```
        `READ :
        begin
        // Match Found in cache
            if (match)
                dataOut  =  dataCache[index];
            else
                //a few possibilities here
                // read data from memory and also
        // copy in cache or not copy in cache; determining
        // this is part of another policy; obtain this info
        // from another task; LRU algotihm means bring this in
        // OK
                if (get_caching_scheme(addr) == 0)
                    state = `READ_MISS;
                else
                    state = `READ_CACHE;
    end

        `READ_MISS:
        begin
            dataOut = MainMemory[addr];
            done= 1;
            state = `IDLE;
        end

        `READ_CACHE:
        begin
            dataOut = MainMemory[addr];
            dataCache[index] = data;
            tagCache[index] = tag;
            validCache[index] = 1;
            state = `IDLE;
        end

        `WRITE:
        begin
            if (get_caching_scheme(addr) == 0)
                state = `WRITE_MISS;
            else
                state =  `WRITE_CACHE;
        end

        `WRITE_MISS:
        begin
            MainMemory[addr] = data;
            if (match)
            // Do not maintain this location in cache any more
            validCache[index] = 0;
            state = `IDLE;
        end
```

```
                    `WRITE_CACHE:
                    begin
                        MainMemory[addr] = data;
                        validCache[index] = 1;
                        dataCache[index] = data;
                        tagCache[index] = tag;
                        state =` IDLE;
                    end
                endcase
            end
            end
        end
    endmodule
    `endif
```

Example 1-10. Behavioral description of a cache controller with write-through scheme.

In the Example 1-10, a cache system is modeled whereby the state machine is mapped into a case statement inside an always block. The IDLE state is when neither read nor write operation is in progress. READ, READ_MISS, and READ_CACHE complete the read operation for this cache. The states of WRITE, WRITE_MISS, WRITE_CACHE model the write operation. Several details of timing and structure are abstracted out. For example, the memory read and write are directly modeled as assignments with no delays or control circuitry. This level of modeling is beneficial for two reasons. Checking the algorithms at the higher level and b. For determining efficacy of a method early in the design cycle. In Examples 11-3 to 11-5, we show different cache models that started out with this model and arrive at more sophisticated caching schemes and also implementations with more structural details.

This model begins with a set of "`define" statements that defines text-substitution macros for the Verilog preprocessor. This helps in separating out constants in one place that can then be varied if desired. Notice that these are done outside the module and will be applicable throughout the file for all modules in it. The module cache has a port-interface and certain reg and memory declarations are done in the beginning. This is followed by initial block defining initialization or reset operation. Then the always block as described in the previous paragraph is added to complete the model.

1.3 Overview of Verilog HDL

1.3.1 Correspondence To Digital Hardware

A hardware description language describes hardware using a language. For every piece of hardware there exists a corresponding language description (and vice versa). The correspondence is explained below in terms of building blocks in hardware and the constructs in the language. The building blocks in a hardware design is

dependent on the methodology being used. A design process consists of top-down and bottom up and a mixture of these two styles. Verilog is a powerful tool in the top-down design methodology and is capable of supporting the bottom up style and consequently the mixed approach as well.

The Bottom-up Methodology Building Blocks	***Language Features***
switches and gates and flip-flops and	switches: pmos, nmos, trans, gates like and, or
counters and registers and muxes;	and user-defined primitives, module definitions
gates : and, nand, or, xor, xnor	and instantiations for creating libraries of flip-
Building alus, fpus (data paths),busses	flops, counters, multiplexors
and control logic (state m/cs)	Create instances of these to model blocks like
CPU design with these and other parts	alus, fpus, data-path elements

Figure 1-2. Bottom-up Methodology and Verilog Language Features Support

The Top-down Methodology Building Blocks	***Language Features***
1. Instruction sets with resets and interrupts	1. Tasks and behavioral statements Disable Statement
2. Size (#bits)	2. Parametrization at module and block level
3. Issues per clock cycle - Pipelining, parallel units,	3. Always blocks, fork-joins (concurrent abilities)
4. Cache designs	4. System modeling capabilities
5. I/O design	5. Synchronization capabilities
6. Memory interface (read-write-fetch cycles)	6. wait, event and delay control
7. Advanced features like branch-predictions	7. Event-Controls, Limited Recursion
8. Parameter sizes for various items in these e.g., cache-sizes	8. Stochastic Modeling capabilities

Figure 1-3. Top-Down Methodology and Equivalent Verilog Language Features Support

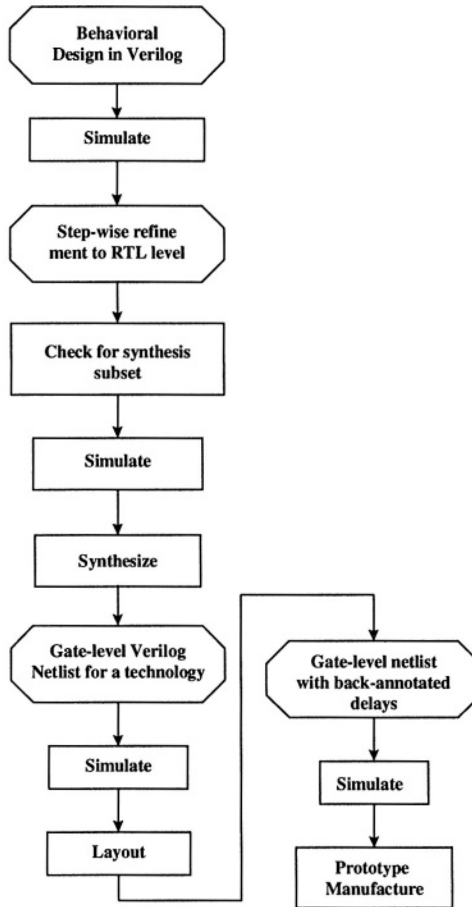


Figure 1-4. Typical Design Flow with Verilog

1.3.2 Typical Design Flow with Verilog

Figure 1-1 illustrates a typical design flow with Verilog. A top-down design starts with a behavioral description and is finally sent to the fab after complete placement, layout and final verification as shown in this diagram.

1. Write a high level behavioral description of the planned design. This step starts with concepts and ends up with a high level description in the Verilog language. This description can have various levels of detail and essentially has architectural elements and algorithmic elements. This may be used with behavioral synthesis for some specialized parts but in general will be simulated

for verifying the parameters, algorithms and architecture. Example here includes the cache controller models for write-through and write-back schemes (Example 11-3 to 11-5). Some level of tests are generated at this point.

2. Next we perform stepwise refinement to the RTL level. This is again simulated and verified for functional correctness. We also check for the RTL synthesis subset during this process. Here we first use the tests developed in step 1 and add tests for the details added at this level. For example, for a cache controller, all communicating wires and registers are modeled here as opposed to higher level models of the blocks in step 1. Thus, correctness of all signals at the (logic) synthesizable blocks are tested in this step.

3. Synthesize the HDL description with the synthesizer. In a typical Synthesizer like Synopsys, this step is divided into two parts—HDL Compilation and the Design Compilation. Synthesizer performs architectural optimizations, then creates an internal representation of the design. Use the Synthesis Design Compiler to produce an optimized gate-level description in the target ASIC library. You can optimize the generated circuits to meet the timing and area constraints wanted. This optimization step must follow the translation to produce an efficient design.

4. The output of a synthesizer is a gate-level Verilog description. This netlist-style description uses ASIC components as the leaf-level cells of the design. The gate-level description has the same port and module definitions as the original high-level Verilog description 1. The gate-level Verilog description from step 3 is now passed through the Verilog simulator. You can use the original Verilog simulation drivers from steps 1 and 2 because module and port definitions are preserved through the translation and optimization processes. Compare the output of the gate-level simulation (step 4) against the output of the original Verilog description simulation (step 3) to verify that the implementation is correct.

5. The synthesis tools can be used at behavioral and at the RTL level. The RTL level is synthesized using techniques that are commonly known as logic synthesis. In this book, the major components of this flow will be discussed. The various representations in Verilog like behavioral, RTL and structural occur at different places in this design cycle and will be discussed fully. Simulation aspects will be discussed for each of those and as a whole as well with the semantic model adding to the depth of this understanding. Synthesis with Verilog will be discussed in various sections and then in Chapter 12. Timing descriptions that are especially important for post-layout verification will be discussed in Chapter 13 on specify blocks,

The entire chip design is a combination of bottom up and top-down design methodologies. In handling the whole design description one deals with top-down module hierarchy; and also multiple views/descriptions of the same module. Ideally we have three views: Architectural, rtl, gates/switch-level. Comparisons of these descriptions provides one of the sources of power in HDL usage. Some observations

of these: Same hierarchical model exists upto certain levels of interest; then the details appear or disappear.

1.3.3 List of Keywords

Verilog defines a set of words to have special meaning. These words are reserved and cannot be used as identifiers or labels in a Verilog model. The type of a statement is identified by the first word in the statement that is a keyword. Examples of these will be 'always', 'and', 'assign', etc. When a statement begins with the word always, there is a special meaning of an always loop attached to that statement. The set of keywords defines the scope or the contents of the language. In other words, the set of features in Verilog can be characterized by the set of keywords as defined below.

always	and	assign	begin
buf	bufif0	bufif1	case
casex	casez	cmos	deassign
default	defparam	disable	edge
else	end	endcase	endfunction
endmodule	endprimitive	endspecify	endtable
endtask	event	for	force
forever	fork	function	highz1
highz0	if	initial	inout
input	integer	join	large
macromodule	medium	module	nand
negedge	nmos	nor	not
notif0	notif1	or	output
parameter	pmos	posedge	primitive
pull1	pull0	pullup	pulldown
rcmos	real	reg	release
repeat	rnmos	rpmos	rtran
rtranif1	rtranif0	scalared	small
specify	specparam	strong1	strong0
supply1	supply0	table	task
time	tran	tranif1	tranif0
tri	tri1	tri0	triand
trior	trireg	vectored	wait
wand	weak1	weak0	while
wire	wor	xnor	xor

Figure 1-5. Verilog Keywords

1.3.4 Comment Syntax

comment
::=short_comment
| long_comment

short_comment
  ::= // comment_text END-OF-LINE

long_comment
  ::= /* comment_text */

comment_text
  ::= {Any_ASCII_character}

1.4 Syntax Conventions

Verilog syntax is precisely defined using BNF(Backus Naur Form). The BNF is described using the following symbols and syntax conventions. As an example, see the definition of comment above in section 1.3.4. The conventions are also known as formal syntax definition conventions.

In this example, we define a non-terminal ‘comment’. The non-terminal name being defined is included in the angle brackets. The definition begins with the non-terminal ‘comment’ followed by ‘::=’ or the ‘is defined as’ symbol. On the right hand side of the ‘::=’ symbol, two alternative definitions ‘short-comment and long-comment are stated. The alternatives are separated by the ‘||=’ symbol. The short_comment is defined to begin with the characters ‘//’ followed by comment_text and ends with the END_OF_LINE character. The long_comment consists of a comment_text enclosed within ‘/*’ and ‘*/’. The comment_text in turn is any sequence of ASCII characters. This is indicated by the repetition symbol ‘*’ preceded by the lexical token ASCII_CHARACTER. The lexical tokens typically consists of a single character, also known as literal, like ‘A’ or ‘0’ or special non-printable characters like the END_OF_LINE’ character. These are defined in the Appendix A along with the complete syntax definition of entire Verilog HDL. In the following chapters, we define syntax using the notation defined here for each construct in the language, along with the semantics and usage with examples.

The lexical conventions in defining the tokens and the conventions in identifying the tokens to be either terminals or non-terminals are as follows:

> Space characters are ignored during the lexical analysis but form token-separators except when present within double quotes.
>
> Identifiers are formed by combination of alphanumeric characters lead by alphabet. Non-alphanumeric characters may be included in an identifier name using leading ‘\’ and are called escaped identifiers. SDF identifiers can contain \ followed by special characters.
>
> Separators of tokens are all-characters that are non-alphanumeric with the exception of escaped identifiers.

All the other rules of writing and reading Verilog HDL code are included in the syntax definitions that follow discussion of every Verilog HDL feature described in the book and are summarized in appendix A and B.

Item	*Meaning*
Spaces	May be used to separate lexical tokens
name ::=	Starts off the definition of a syntax construct item. Sometimes name contains embedded underscores "_". Also, the "::=" may be found on the next line.
|	Introduces an alternative syntax definition, unless it appears bold. (See next item)
name	Bold text is used to denote reserved keywords, operators, and punctuation marks required in the syntax.
[item]	Is an optional item that may appear zero or one time.
{item}	Is an optional item that may appear zero, one or more times. IF the braces are in bold, then they are part of the syntax.
*Name1*_name2	This 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.

Figure 1.1. Formal Syntax Definition Conventions

1.5 Exercises

1. Give three advantages behind using an HDL for hardware design.
   Give three applications or technologies facilitated by Verilog HDL.
2. In the example m555 of timer, rewrite using:
   1. gates
   2. initial and forever
   3. initial and while loops
   4. assign statement
3. Compare the two ways of modeling counter discussed in Chapter 1.
4. Identify the following constructs(one instance of each) from 8085-based example:

   event declarations
   event usage
   posedge/negedge
   disable
   task
   function
   nmos
5. Rewrite the behavioral model of the counter with the following changes:

   a. make clock period 200 units.

   b. the counter counts to 32 numbers (0 through 31) Try making the same changes in the gate-level model and see the advantages of the behavioral modeling.

6. In the simulation of the Example 1-6, the output goes to x in between steady state values. The following lines show one such case, from time 453 to 457:

```
time=     450 preset=l clear=l clock=l  q[0]=x q[1]=x q[2]=xq[3]=x
time=     453 preset=l clear=l clock=l  q[0]=0 q[l]=x q[2]=xq[3]=x
time=     455 preset=l clear=l clock=l  q[0]=0 q[l]=0 q[2]=xq[3]=x
time=     457 preset=l clear=l clock=l  q[0]=0 q[l]=0 q[2]=0q[3]=x
```

Explain the occurrence. Simulate the two design separately by commenting out one instance at a time from the test_counter module.

2 DATA TYPES IN VERILOG

2.1 Overview

Verilog supports only predefined data types. These include bits, bit-vectors, memories, integers, reals, events, and strength types. These define the domain of description in Verilog. Verilog deals mainly in the domain of bits and bytes while describing the circuits. The real type is useful for delays and time and is also useful in higher level modeling such as stochastic analysis and digital signal processing algorithms. The hardware types include net and reg types. This, in general, can be seen as wires and registers. The nets are further declared to be of different types like tri-stated or non-tri-stated and whether the resolution of multiple connections results in anding, oring or uses prior value. The details of these can be seen in the following sections.

2.2 Value Systems

The data types have ramifications on the scope of the description. The value-systems define the kinds of values defined in the language and entail the operations supported on them. These also have corresponding constant (numbers or literals) definitions. The various values in Verilog are:

bits and integers(32 bits), time (64 bits)—bit-vectors and integers can be freely intermixed. Integers are defined to be 32 bits. Time values are 64 bits. The bits actually are of two types as below.

4-state values (0,1,x,z); also known as logic values

128-state types (4 states and 64 strengths (8 '0' and 8 '1' strengths)

floating point types (real numbers)

character strings

delay values – These are single, double, triplet or n-tuple indicating rise, fall any other transition delay

transition values – (01) - change from 0 to 1. These may in user-define primitives or specify blocks

Boolean/conditional values – true /false OR 0/1

units (only for timescale) – femtoseconds (Fs) to seconds (s)

2.3 Data Declarations

2.3.1 Introduction

Different data types in Verilog are declared by data declaration statement. These statements occur in module definitions before usage and some of these can be declared within named sequential blocks. In addition to the value-types that may distinguish different types of data, the hardware characteristics of wires versus registers also are distinguished as net versus reg declarations in Verilog. The term driving is used in hardware descriptions to describe how a value is assigned to a date element. Nets and regs are two main types of data elements in Verilog. Nets are continuously driven from continuous assignments or from structural elements such as module ports, gates, transistors or user defined primitives. Regs are driven strictly from behavioral blocks. Nets are typically implemented as wires in hardware and regs may either be wires, or temporaries or flip-flops (registers).

The different data types in Verilog are declared as one of the following types:

parameter	These are constant valued expressions resolved after compilation and allow modules to be parametrized.
input **output** **inout**	This and the next two types define the direction and size of a port.
net	This is a type of connection or wire in hardware with different resolutions.
reg	This is an abstract type that is like a register and is driven behaviorally.
time	This contains time quantities like delays and simulation time.
integer	This is an integer values type.
real	This is a floating point or real valued type.
event	This indicates a flag that can trigger activity.

These can all be declared at the module level. The other descriptions in Verilog with scope-creation capabilities include tasks, functions and named begin-end blocks.. The non-module scopes are all behavioral scopes. Nets are non-behaviorally driven and thus can not be declared in the other scopes. All other types can be present in tasks and begin-end blocks Each one of these is explained in sections 2.4 through 2.12.

2.3.2 Examples

```
input i1, i2;
        reg [63:0] data;
        time simtime;
```

Example 2-1. Data declarations.

The first line in Example 1-2 is an input declaration line, the second is a 64-bit reg declaration for data. The last line in this example is a time declaration for the variable named simtime.

2.3.3 Syntax

```
data_declarations  ::= parameter_declaration
                   ||= input_declaration
                   ||= output_declaration
                   ||= inout_declaration
                   ||= net_declaration
                   ||= reg_declaration
                   ||= time_declaration
                   ||= integer_declaration
                   ||= real_declaration
                   ||= event_declaration
```

2.4 Reg Declaration

2.4.1 Introduction

Reg declarations are done for all signals that are driven from the behavioral descriptions Regs retain a given value until they are assigned a new value in the sequential description (initial or always blocks). Reg types are more abstract than net types but are closely tied to concepts of registers (with storage) and can be realized in hardware as such. They can also be realized as wires or may be temporaries with no hardware mapping depending on their usage within a behavioral block.

2.4.2 Examples

```
reg rl, r2;
reg [63:0] data_a, data_b, data_c;
```

Example 2-2. Reg declarations.

2.4.3 Syntax

reg_declaration
::= **reg** [range] list_of_register_ identifiers;
list_of_ register_ identifiers ::= register _ identifier , { register _ identifier }

2.5 Net Declaration

2.5.1 Introduction

The net is a set of data types in a Verilog description that represents the physical wires in a circuit. A net connects gate-level instantiations, module instantiations and continuous assignments. The Verilog language allows you to read a value from a net from within behavioral descriptions, but you cannot assign a value to a net within the behavioral descriptions. (An always block is a specific type of begin...end block). A net does not store its value. It must be driven in one of two ways:

- By connecting the net to the output of a gate or module.
- By assigning a value to the net in a continuous assignment.

Different net-types defined in Verilog are described below and the tables in Figure 2-1 summarize their functionality. Resolution is a rule for resolving values with multiple drivers the following is also explained in table on next page.

wire a net with 0,1,x values and resolution based on equality

wand a net with 0,1,x values and resolution of wired and

wor a net with 0,1,x values and resolution of wired or

tri a net with values of 0,1,x,z and resolution of tri-state bus

tri0 a net with values of 0,1,x,z and resolution of tri-state bus and a default value of 0 when no driving value

tri1 a net with values of 0,1,x,z and resolution of tri-state bus and a default value of 1 when no driving value

trior a net with values of 0,1,x,z and resolution of tri-state for z-non-z values using ‘or’ function of non-z values

triand a net with values of 0,1,x,z and resolution of tri-state for z-non-z value using ‘and’ function for non-z values

trireg a net with values of 0,1,z,x and tri-state resolution together with charge storage (previous value used in resolving new value)

supply0, supply1 (gnd and vdd)

tri/wire	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	x

triand/wand	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

trior/wor	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

tri1	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	1

tri0	0	1	x	z
0	0	0	0	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	0

trireg	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	P

Figure 2-1. Tables of Net Types and Resolution Functions

wire

The wire data-type defines a type that is a simple connection between two places where it is used. Multiple drivers resolve using tri-state function during simulation but synthesis creates non-tri-states for wire types. In the Example 2-3, 2-wire declarations are made. The first declares a scalar wire w. The second declares a vector wire w2 with 3 bits. Its most significant bit (msb) has an index of 2 and its least significant bit (Isb) has an index of 0.

```
wire w1;
wire [2:0] w2;
```

Example 2-3. Wire declarations.

wand

The wand (wired-and) data type is a specific type of wire that uses the and function to find the resulting value when multiple drivers are present. In Example 2-4, two variables drive the variable c. The value of out is determined by the logical and of i1 and i2.

```
module wand_test(out, i1, i2);
input il, i2;
output out;
wand out;
assign out = il;
assign out = i2;
endmodule
```

Example 2-4. Wand (wired-AND) declarations and usage.

You can assign a delay value in a wand declaration, and you can use the Verilog keywords scalared and vectored for simulation.

wor

The wor (wired-OR) data type is a specific type of wire. In Example 2-5, two variables drive the variable c. The value of out is determined by the logical OR of in1 and in2.

```
module wor_test(il, i2, out);
input il, i2;
output out;
wor out;
assign out  = in1;
assign out = in2;
endmodule
```

Example 2-5. wor (wired-OR) declarations and usage.

tri

The tri (three-state) data type is a specific type of wire where the resolution of multiple drivers is done using the rules of tri-state bus. All variables that drive the tri must have a value of Z (high-impedance), except one. This single variable determines the value of the tri.

In Example 2-6, three variables drive the variable out. They are set in another module such that only one driver is active at a time.

```
module tri_test (out,  select, a, b, c);
    input [1:0] select, a, b, c;
    output out;
    tri out;

    assign out = a;        //make the tri connection
    assign out = b;
    assign out = c;
endmodule

module abc(a, b, c, select)
    output a, b, c;
    input select;
    always @ ( select ) begin
    a = 1'bz;              //set all variables to Z
    b = 1'bz;
    c = 1'bz;
    case ( select )    //set only one variable to non-Z
    2'b00:a=1'b1;
    2'b01:b=l'b0;
    2'b10:c=l'b1;
```

```
        endcase
        end
    endmodule
```

Example 2-6. Tri (Three-State) declarations and usage.

The drivers may be gate outputs as in the example below.

```
    module tri_test (out,  select, a, b, c);
        input [1:0] select, a, b, c;
        output out;
        tri out;

        nand(out,    a1,a2);    //make the tri connectio
        nand(out, b1, b2)
        nand( out, c1, c2, c3);
    endmodule
```

Example 2-7. Nets of tri types declared and used for multiple drivers with tri-state resolution.

supply0 / supply1

The supply0 and supply1 data types define wires tied to logic 0 (ground) and logic 1 (power or vss/vdd). Using supply0 and supply1 is the same as declaring a wire and assigning a 0 or a 1 to it. In Example 2-7 power is tied to power supply (always logic 1 – overriding strength) and gnd is tied to ground (always logic 0 – overriding strength).

```
        supply0 gnd;
        supply1 power;
```

Example 2-8. supply0 and supply1 constructs.

trireg

The trireg net is like a tri wire but has a capacitive nature and stores its last value when all drivers are tri-stated. Thus, the trireg net will always have a value of 0 or 1 or x but not z. The capacitance on the trireg is specified by a size that is either large, medium or small with the default value of medium when unspecified. For a Verilog model as follows, we obtain the resulting value on wire trg of trireg type when the transistor driving it is in off state.

```
    module m;
        reg c0, c1, i1, i2
        tri d0, d1, d2;
        trireg d;
        and(d0, i1, i2);
        nmos nl  (d1,  d0,  c0),   n2(d, d1, c1);
```

```
initial
begin

        $monitor("time = %d d = %d     c0=%d  c1=%d  d0=%d  d1=%d
        i1=%d i2=%d", $time, d, c0, c1, d0, d1, i1, i2);
            #1
        i1 = 1;
        i2 = 1;
        c0 = 1;
        c1 = 1;
        #5
        c0 = 0;

    end

endmodule
```

```
C1>.
time =          0 d = x     c0=x  c1=x  d0=x  d1=x  i1=x  i2=x
time =          1 d = 1     c1=1  c1=1  d0=1  d1=1  i1=1  i2=1
time =          6d =  1     c0=0  c1=1  d0=1  d1=z  i1=1  i2=1
```

Example 2-9. Trireg net and the switch-level modeling example.

In this example, if the trireg is replaced with a tri, then the value of d at time 6 units will be 'z'.

2.5.2 Syntax

net_declaration ::= NETTYPE [expandrange] [delay] list_of_net_ identifiers;
| **trireg** [charge_strength] [expandrange] [delay]
list_of_ net_ identifiers; | NETTYPE [drive_strength] [expandrange] [delay]
list_of_net_decl_assignments ;
list_of_net_ identifiers ::= *net_* identifier, { *net_* identifier }
NETTYPE ::= : **wire | tri | tri1 | supply0 | wand | triand | tri0 | supply1 | wor | trior | trireg**

expandrange
::= range
| **scalared** range
| **vectored** range

charge_strength
::= (small)
| (medium)
| (large)

In the above syntax definitions expandrange optionally defines vectors and is explained in section 2.7.2, delay expressions are defined in section 2.12,

drive_strength specifications are given while giving transistor-level models and are explained in Chapter 17.

2.5.3 Examples

```
wire w1, w2;

tri [7:0] t1, t2;

trireg large trg1, trg2;

triand [63:0] #(10:5) trnd1;
```

Example 2-10. Net type declarations.

In Example 2-10, the first line with the keyword 'wire' declares w1 and w2 to be single-bit or scalar wires. The second line declares 8-bit-vector-wires of type tri with names of t1 and t2. The trireg (a capacitive net) with the size large are declared on the next line with names of trg1 and trg2. The 64-bit triand type net with minimum and typical delays is declared on the 4th line above.

2.6 Port Types

2.6.1 Introduction

You must explicitly declare the direction (whether input, output, or bidirectional) of each port that appears in the port list of a port definition. Three kinds of ports are defined in Verilog—input, output, inout. The ports must be either nets or regs. The only place where a reg can occur is output port. Constants and expressions are allowed in port declarations.

input

All input ports of a module are declared with an input statement. An input is a type of wire and is governed by the syntax of wire. You can use a range specification to declare an input that is a vector of signals, as for input b in the following example. The input statements can appear in any order in the description but must be declared before they are used. For example:

```
input a;
input [2:0] b;
```

output

All output ports of a module are declared with an output statement. Unless otherwise defined by a reg, wand, wor, or tri declaration, an output is a type of wire and is governed by the syntax of wire. An output statement can appear in any order in the description, but you must declare it before you use it. You can use a range specification to declare an output that is a vector of signals. If you use a reg

declaration for an output, the reg must have the same range as the vector of signals. For example:

```
output a;
output [2:0]b;
reg [2:0] b;
```

inout

You can declare bidirectional ports with the inout statement. An inout is a type of wire and is governed by the syntax of wire. You must declare an inout before you use it. For example:

```
inout a;
inout [2:0]b;
```

2.6.2 Examples

```
module fullAdder(cOut, sum, aIn, bIn, cIn);
        input aIn, bIn, cIn;
        output cOut, sum;
        wire aIn, bIn, cin;
        reg cOut, sum;
         ........
endmodule
```

Example 2-11. Port type declarations.

2.6.3 Syntax

```
list_of_ports
    ::= ( port {,port })

port
    ::= [port_expression]
    | . port_identifier ( [port_expression] )

port_expression
    ::= port_reference
    | { port_reference ,port_reference }

port_reference
    ::= port_identifier
    | port_identifier[ constant_expression ]
    | port_identifier [ msb_constant_expression :lsb_constant_expression ]
```

2.7 Aggregates – 1 and 2 Dimensional Arrays (Vectors and Memories)

2.7.1 Introduction

Only 1- and 2-dimensional arrays are supported in Verilog. One-dimensional arrays are called bit-vectors and may be nets or regs. The 2-dimensional aggregates are called memories and are reg kind. You can define an optional range for all the data types presented in this chapter. The range provides a means for creating a bit-vector. The syntax for a range specification is [msb : lsb]. Expressions for msb (most significant bit) and lsb (least significant bit) must be non-negative constant-valued expressions. Constant-valued expressions are composed only of constants, Verilog parameters, and operators. There is no maximum size for the length of a bit-vector and this is limited only by a particular implementation of simulation, synthesis, or other tools used with the Verilog source.

2.7.2 Examples

```
reg [31:0] data;
reg [23:0] addr;
wire [63:0] system_bus;
```

Example 2-12. Aggregate declarations.

In the first line above, data is a bit-vector reg of 32 bits width, declared with a range specification of 31:0 with the most significant bit taking the index of 31 and the least significant bit that of 0. The third line is a wire declaration of 64 bits width and the index of bits ranging from 63 to 0. Some examples are:

```
wire vectored [31:0] bus1;
tri scalared [63:0] bus2;
```

Example 2-13. Vectored and scalared bit-vector net declarations

In the above example, the first line indicates that the vector bus1 is used as a vector in connecting ports and can be maintained as a vector while on the second line the term scalared indicates usage of the vector bit-selects bus2[n] in connecting on port boundaries and the word 'scalared' indicates to the tools of such usage. These directives of vectored and scalared are only for efficient simulation or synthesis and do not change the functionality of the design.

```
reg [31:0] memory_bank1 [0: (1024*64)-1];
```

Example 2-14. Memory declarations.

In the above example, memory_bank is declared to be a memory of 64K locations of 32 bits each. Elements of memory are reg and are declared and used as such and the memory is addressable in terms of the entire 32 bits (in terms of the word-size).

2.7.3 Syntax

Vectored Net and Reg Declarations

This syntax is covered in net declaration in section 2.5.2.

Memory Declarations

```
memory_variable
  ::= memory_identifier [ constant_expression : constant_expression ]
```

Special Facility for Vectored Nets

```
expandrange
::= range
    | scalared range
    | vectored range
```

2.8 Delays on Nets

2.8.1 Introduction

Nets may have delays associated with them as declared in the declarations. These delays are the delays from the time a driver on a net changes to the time of actual change on the net. Delays are given by the number or the expression following the '#' symbol. These can be constants, parameters, expressions of these or even dynamic expressions using other variables. Delays can be rise, fall, or hold(change to z) delays and each of these delays in turn may have three values—minimum, typical and maximum. The rise, fall and hold delay specifications are separated by commas and the min-typ-max specifications are separated by colons. The rise delay includes delays when values change from 0 to 1, 0 to x and x to 1. Fall delay applies to changes from 1 to 0, 1 to x and x to 0. The hold delay values apply for changes from 0 to z, 1 to z and x to z changes. The same concepts of delays are useful for gates, transistors, user-defined primitive instances and behavioral descriptions.

2.8.2 Examples

```
tri #5 t1, t2;
wire #(10,9,8) w1, w2;
wand #(10:8:6, 9:8:6) w3;
```

Example 2-15. Net declarations with delay specifications.

In the first example above, t1 and t2 have rise, fall and hold delays of five time units. In the second line, wires w1 and w2 have three different values for the three changes—ten for rise, nine for fall, and eight for hold. The rise delay includes delays when values change from 0 to 1,0 to x and x to 1. Fall delay applies to changes from 1 to 0, 1 to x and x to 0. The hold delay values apply for changes from 0 to z, 1 to z and x to z changes.

2.8.3 Syntax

```
delay ::= delay2 | delay3
delay3 ::= #delay_value | #( delay_value [,delay_value [,delay_value]])
delay2 ::= #delay_value | #( delay_value [,delay_value])
delay_value
      ::= unsigned_number
      | parameter_identifier
      | (mintypmax_expression [,mintypmax_expression] [,mintypmax_expression])
```

2.9 Integer and Time

2.9.1 Introduction

The type time is 64-bit wide and contains result of system task $time or computation on other time variables. The type integer is 32 bit and can be assigned and used freely as integer or 32-bit (signed) register in expressions,

2.9.2 Examples

```
      time t1, t2;
  integer i1, i2;
```

Example 2-16. Integer and time declarations.

2.9.3 Syntax

time_declaration
 ::= time list_of_register_ identifiers;
integer_declaration
 ::= integer list_of_register_ identifiers;

2.10 Real Declaration

2.10.1 Introduction

Real numbers are used in Verilog to perform delay descriptions and in abstract algorithms. Their bit representation is done using IEEE floating point standard.

2.10.2 Example

```
real r1, r2;
```

Example 2-17. Real declarations.

2.10.3 Syntax

real_declaration
 ::= real list_of_ real_identifiers;

list_of_ real_identifiers
 ::= real_ identifier, {real_ identifier}

2.11 Event Declaration

Event types are special flags that can trigger activity into a process waiting for an event to occur.

Example and syntax are :

```
event e1;
event_declaration ::= event list_of_event_variables;
```

2.12 Parameter Declarations

You can use a parameter wherever a number is allowed, and you can define a parameter anywhere within a module definition. However, the Verilog language requires that you define the parameter before you use it. Example 2-17 shows two parameter declarations. Parameters TRUE and FALSE are unsized, and have values of 1 and 0, respectively. Parameters SO, S1, S2, and S3 have values 3, 1, 0, and 2, respectively, and are stored as 2-bit quantities. Parameters are a way of defining modules which can be configured at the instance time. The module instantitation statement lets one change the values of parameters in a module to new values for each instance. Separately, the 'defparam' statement also allows one to change the parameters from outside. This facility is extensively used for operations such as delay back-annotations from a post-layout circuit into a pre-layout or rtl design.

The parameter data type is automatically deduced by Veriolog compiler using the expression on the right-hand side of '=' in parameter declarations. These expressions may be of any valid data-type, like bits, vectors, integer, or reals in Verilog.

2.13 Examples

Following are scalar and vector parameter declarations:

```
parameter TRUE=1, FALSE=0;
parameter [1:0] S0=3, S1=1, S2=0, S3=2;
```

Example 2-18. Parameter declaration examples.

2.14 Syntax

parameter_declaration
::= parameter list_of_param_assignments;

```
list_of_param_assignments
::=param_assignment {,param_assignment}

param_assignment
::=identifier = constant_expression
```

2.15 Hierarchical Names

2.15.1 Introduction

Verilog supports names imported from other modules or other scopes in all places a simple name can occur. These are full hierarchical names as follows: Typically these are used in back-annotations or forward-annotations from outside the current design description. SDF files described in chapter 18, may, for example, be used to generate delay files using full-hierarchical names. These are also very useful for debugging as seen in Chapter 10. The usage of these for circuit description should be done with discretion since one is creating connections across module boundaries without putting the connecting wires or regs on the port list.

2.15.2 Examples

```
testbench.top_system.cpu.alu.fulladder.cin
testbench.top_system.cpu.reg11
```

Example 2-19. Hierarchical names.

The name begins with a top-level module-name testbench and then is followed by instance names until the level at which the name is defined is reached. In the first line of Example 2-19, the testbench instantiates the system with instance name of top_system which in turn instantiates cpu with instance name cpu and then alu followed by instance fulladder of an adder and the cin net.

2.15.3 Syntax

```
Hier_name::= name_of_module *.name_of_item
```

2.16 Exercises

1. Write a declaration for a wire and bus of 64 bits wide and a rise delay of 10 and fall delay of 8 time units.
2. Write memory declarations for a 64K x 8-bit memory.
3. Check the correctness of the following declarations:

```
net n1, n2;
reg[63:0] r1,r2,r3;
reg [0:-5] r;
events e1, e2;
```

4. Write a declaration of a tri-state wire with charge storage of medium capacitance.

5. In Example 2-7, change the net-type to triand and trior and trireg from tri and then compute the expected results. Simulate and verify the results [Use $showvars or driver value display to check component and computed values].

3 ABSTRACTION LEVELS IN VERILOG: BEHAVIORAL, RTL, AND STRUCTURAL

3.1 OVERVIEW

3.1.1 Introduction

Hardware can be described in different levels of abstraction that involve different levels of detail and model different characteristics. Three commonly understood levels of abstraction are behavioral, register-transfer-level (RTL), and structural. The three types of descriptions together constitute the descriptions of hardware in a hardware description language. These are explained in the following sections of this chapter.

3.1.2 Examples

```
module mult_behav(out, in1, in2, carry, sign);
      input [31:0] in1,in2;
      output [63:0] out;
      output carry;
      output sign;
      reg carry, sign;
      reg [63:0] out;

      always @(in1 or in2)
      begin

            {carry, out} = in1 * in2;
            if ({carry, out} <0)
                  sign= 1;
            else
```

```
                    sign = 0;
          end
     endmodule
```

Behavioral Level of Modeling of multiplier

```
     module mult_rtl(out, in1, in2, carry, sign);

          input[31:0] in1,in2;
          output [63:0] out;
          output carry;
          output sign;
          assign {carry, out} = in1 * in2;
          assign sign = ({ carry, out} < 0) ? 1 : 0;
     endmodule
```

RTL level of modeling of same multiplier

Example 3-1. Levels of abstractions.

In the above example, a multiplier is modeled at the behavioral and at the RTL level. For descriptions like these, both behavioral and RTL models are used. The behavioral model always uses blocks with procedural statements, while the RTL model uses continuous assignments that begin with keyword 'assign'.

3.1.3 Syntax

source_text
  ::= {description}

description
  ::= module_declaration
  | UDP_declaration

module_declaration
  ::= module_keyword module_identifier [list_of_ports];
    {module_item}
    endmodule
module_Keyword
  ::= module | macromodule
    {module_item}
    endmodule

module_item
  ::=data_declaration
  ||= functional_descriptions
  ||= module_timing_descriptions

functional descriptions
  ::= behavioral_descriptions
  ||= RTL_descriptions
  ||= structural_descriptions

3.2 Behavioral Abstractions In Verilog

3.2.1 Introduction

This level of modeling provides advanced data and control flow in Verilog. This enables descriptions that are algorithmic descriptions of hardware. It enables synchronization between different blocks or processes. This is the highest level of abstraction among the three levels including the structural and the RTL.

Control flow modeling in Verilog at this level is substantially improved. Verilog derives some of the basic algorithm description capabilities from structured programming aspects of "C", but with specific facilitations for hardware descriptions.

Synchronization or timing controls features are also quite advanced especially when mixed with the copious control flow capabilities. The whole design is represented as concurrently executing processes or evaluation blocks as explained in Chapter 4.

Specially created timing specification blocks (Specify Blocks) provide rich inter-module timing behavior description in Verilog. This is structurally bound to the module pins or inputs-outputs.

3.2.2 Examples

```
// Here is an example of behavioral description of the full adder above
// Abstraction is added by a. Use of operator b. Use of vectorizing c. Control flow abstraction
// (always block- Implicit Sensitivity to the RHS operands; Sequential Description adds
power to the
//modeling ability by reducing complexity in control flow (almost random in RTL)

    module fullAdder_b(cOut, sum, aIn, bIn, cIn);
      output        cOut, sum;
      input         aIn, bIn, cIn;

      reg[1:0]tmp;
      reg cOut, sum;

      always @(aIn or bIn or cIn)
      begin
            Imp  = aIn+bIn+cIn;
            sum  = tmp[0];
            cOut = tmp[1];
      end
    endmodule
```

Example 3-2. Behavioral level of abstraction – adder.

In the above example, adder is modeled behaviorally using always loop. The 'always' keyword is followed by an event control described using '@' sign. The

following event expression indicates that any value change in the inputs aIn, bIn or cIn triggers the evaluation of the following ‘begin-end’ block. Inside this ‘begin-end’ block, the addition (sum) and the carryOut (cOut) are computed. The addition operation, using 2 bits with the tmp[1:0] as the target, computes both the output bits in the first statement of the block. The next two statements separate out the 2 bits into the sum and cOut bits that appear on the module boundary.

3.2.3 Syntax

```
behavioral_descriptions
      ||= initial_statement
      ||= always_statement
      ||= task_declaration
      ||= function_declaration
```

3.3 Register Transfer Level Abstractions in Verilog

3.3.1 Introduction

Registers are aggregates of data. Data transfer between registers is known as RTL transfer and the event-driven model as basis for these transfers. The event-driven model here is crucial to these descriptions. In Verilog, the mechanism to model these is called continuous assignments. The LHS gets a new value when anything on RHS changes. This will be the main driver in (logic) synthesis, especially for datapath. (Control Logic comes from FSM descriptions that will be discussed later.)

These are concurrent. Thus, there is no order implied by the model in these. However, due to the event driven nature of these and the fact that these are interdependent, will create an order like in the hardware that is being modeled. As the RTL descriptions are in between structural and behavioral, these work in between the other two. In these, we drive the left-hand side just like in ports that are input ports, the rules for ports that can be driven apply here. Thus, we will only use types that are physically realizable or can physically be driven on a continuous basis. Registers (reg declarations) are one-shot deals as far as assignments are concerned. They are assigned when we get to them in a serial block in "C"-like fashion. But the continuous assigns can only happen on nets.

In Verilog, one is allowed to use concatenations of nets as a vector net with no name (like in a synthesizer with property on the net, to keep it post-synthesis). This is useful at times, but it is better to create another net and assign the concatenations all other nets to this net. This helps in debugging and better readability of the code.

Like "C", one can also combine the declaration and assignment for nets with continuous assignments. This is the first abstraction from a gate-level description.

Rather than use an and gate, one would write:

```
assign c = a & b;
```

In RTL descriptions, power of expressions is available to you in place of a network of several gates, which are hard to program with for achieving actualization of an algorithm or a machine in hardware. Boolean Algebra is also now applicable for optimizations and reorganizations. The additional modeling capabilities here include: delay specifications and strength specifications. See the examples below and syntax as well for details.

If you want to drive a value onto a wire, wand, wor, or tri, use a continuous assignment to specify an expression for the wire's value. You can specify a continuous assignment in two ways:

- Use an explicit continuous assignment statement after the wire, wand, wor, or tri declaration.
- Specify the continuous assignment in the same line as the declaration for a wire.

Example 3-3 shows two equivalent methods for specifying a continuous assignment for wire a.

```
wire a;          //declare
assign a = b & c;   //assign
wire a = b & c;     //declare and assign
```

Example 3-3. Two equivalent continuous assignments.

3.3.2 Example

```
module fullAdder_r(cOut, sum, aIn, bIn, cIn);
      output        cOut, sum;
      input         aIn, bIn,  cIn;

      wire          x1,x2,x3,x4,x5,x6,x7,x8,x9;
      // This describes the same full adder using RTL

      assign x2 = ~(aIn & bIn);
      assign cOut=  ~(x2& x8);
      assign x9 = ~(x5 ^  x6);
      assign x5 = ~(x1 | x3);
      assign x1 = ~(aIn | bIn);

      assign x8 = (x1 | x7);
      assign  sum = ~x9;

      assign x3 = ~x2;

      assign x6 = ~x4;
      assign x4 = ~cIn;
      assign x7= ~x6;
endmodule
```

Example 3-4. RTL abstractions – adder.

In Example 3-4 above, the adder is modeled using continuous assignments at the RTL level.

```
module fullAdder_r(cOut, sum, aIn, bIn, cIn);
      output    cOut,   sum;
      input     aIn,  bIn,  cIn;

      wire          x2;

      // This describes the same RTL above, but using power of expressions

      // This is not simplified just to show how it was obtained by substituting
      // each intermediate term above until only primary inputs remain on RHS

      assign cOut = ~(~(aIn & bIn) & (((-(aIn | bIn)) |
                                                  ~(~(~cIn)))));
      assign  sum = ~ (~((~((~(aIn | bIn)) | (~~(aIn & bIn))))^ (~(~cIn))));

endmodule
```

Example 3-5. RTL abstractions – adder with boolean optimizations.

```
module (cOut, sum, aIn, bIn, cIn);
      output        cOut, sum;
      input         aIn, bIn, cIn;
      wire          x2;
   // This describes the same RTL above, but simplifying using Boolean algebra rules
      assign cOut =  ~(~(aIn & bIn) & (((~(aIn|bIn))|~(cIn)));

      assign  sum = (~((~(aIn | bIn)) | (aIn & bIn)))^ (cIn);
endmodule

module  ffNand(q,qBar,preset,clear);
      output        q, qBar;
      input      preset, clear;
      wire           q, qBar;
      tri          q1,q2;
      wire           preset, clear;


   // Following 2 declarations are examples of declarations with
   //  continuous  assignments
      // The following net needs a pull for it to survive
      tri (pull0, pull1) #(10: 5 :20) d = ~(qBar & preset);

      // The following is weaker than weak
      tri (highz0, highz1) #(10: 5 :20) d1 = ~(qBar & preset);

      //  Simple  Continuous  assignments
      assign q = ~(qBar & preset);
```

```
        assign qBar = ~(q & clear);

    // Continuous assignments with rise and fall delays
            assign #(10:1)q1 = ~(qBar & preset);

    //Continuous assignments with delays and drive strengths
        assign (strong0, strong1) #(10:1)q1 = ~(qBar & reset);
        // Here is the overriding strength even to the strong
        // This also models rise, fall and hold delay
        assign (supply0, supply1) #(10: 1: 20) v = ~(qBar & reset);
    endmodule
```

Example 3-6. Continuous assignments – RTL modeling.

3.3.3 Syntax

```
RTL_descriptions
    ||= continuous_assign

continuous_assign
    ::= assign [drive_strength] [delay] list_of_assignments ;
    | NETTYPE [drive_strength] [expandrange] [delay] list_of_assignments ;
```

3.3.4 RTL Descriptions – Other Definitions

The word RTL descriptions is also used for the synthesizable subset of Verilog described in Chapter 13. This typically includes state descriptions using behavioral blocks, and combinational description using behavorial or continuous assignments.

3.4 Expressions

3.4.1 Overview

In general, expressions are allowed in continuous assignments on the right-hand side. Expressions provide ability to abstract operations up from the gate-level. Expressions consist of operators and operands. The operands are formed by the various data declared items like nets, regs, integers, reals, time, event, etc. They may also be formed by sub-expressions which may be parenthesized. The operators are formed by symbols that represent operations such as addition, subtraction, multiplication, etc., represented by symbols like +, -, *.

3.5 Operators in Expressions

3.5.1 Introduction

Operators identify the operation to be performed on their operands to produce a new value. Most operators are either unary operators that apply to only one operand, or

binary operators that apply to two operands. Two exceptions are conditional operators, which take three operands, and concatenation operators, which take any number of operands. Verilog provides a rich set of operators as described in the next few pages.

The main operators under various categories are:

Unary Operators: + - ! ~ & ~& | ^| ^ ~^

Relational Operators : **< > <=> = =**

Arithmetic Operators : + - */ %

Logical Operators : ! && II != == === !==

Boolean Operators: & | ~ ^ ^~

Shift Operators : >> <<

Concatenation: { }

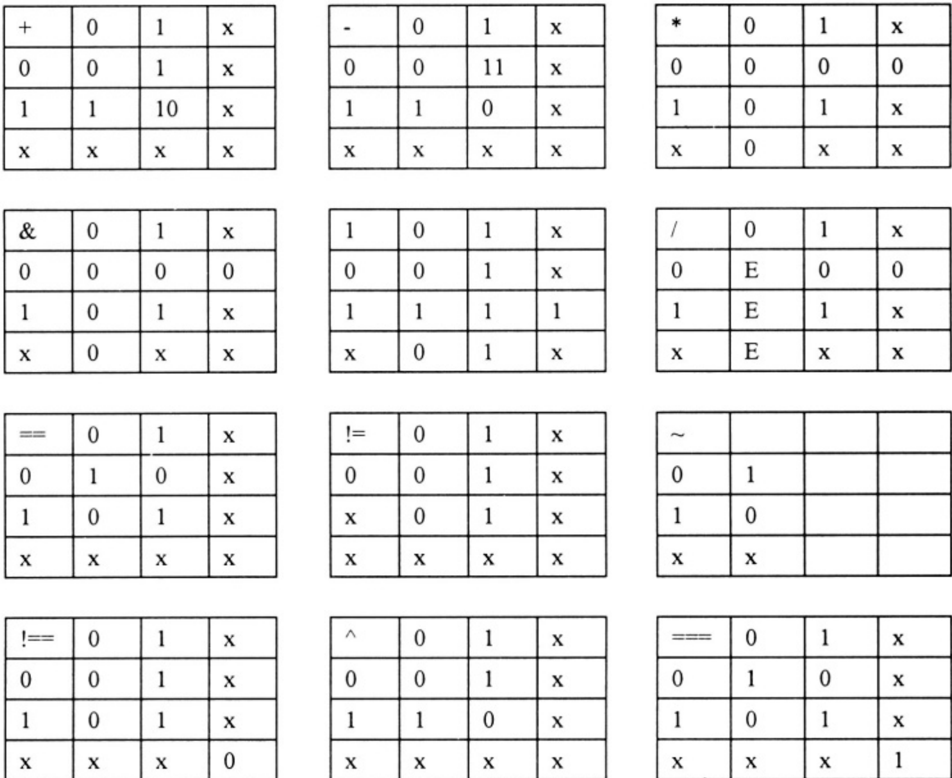
If Operator : ? :

The name for the operators individually are given below.

<table>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td>{ }</td><td>concatenation</td></tr>
<tr><td>+</td><td>add</td></tr>
<tr><td>-</td><td>subtract</td></tr>
<tr><td>*</td><td>multiply</td></tr>
<tr><td>/</td><td>divide</td></tr>
<tr><td>%</td><td>modulus</td></tr>
<tr><td>></td><td>greater than</td></tr>
<tr><td>>=</td><td>greater than or equal to</td></tr>
<tr><td><</td><td>less than</td></tr>
<tr><td><=</td><td>less than or equal to</td></tr>
<tr><td>=</td><td>equal to</td></tr>
<tr><td>!=</td><td>not equal to</td></tr>
<tr><td>!</td><td>logical NOT</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>bit-wise NOT</td></tr>
<tr><td>&</td><td>bit-wise AND</td></tr>
<tr><td>|</td><td>bit-wise OR</td></tr>
<tr><td>^</td><td>bit-wise XOR</td></tr>
<tr><td>^~ ~^</td><td>bit-wise XNOR</td></tr>
<tr><td>&</td><td>reduction AND</td></tr>
</table>

| reduction OR
~& reduction NAND
~| reduction NOR
^ reduction XOR
~^ reduction XNOR
« left shift
» right shift
? : conditional or if operator

The tables for the binary operators are given in Figure 3-1.



+	0	1	x
0	0	1	x
1	1	10	x
x	x	x	x

-	0	1	x
0	0	11	x
1	1	0	x
x	x	x	x

*	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

1	0	1	x
0	0	1	x
1	1	1	1
x	0	1	x

/	0	1	x
0	E	0	0
1	E	1	x
x	E	x	x

==	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

!=	0	1	x
0	0	1	x
x	0	1	x
x	x	x	x

~			
0	1		
1	0		
x	x		

!==	0	1	x
0	0	1	x
1	0	1	x
x	x	x	0

^	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

===	0	1	x
0	1	0	x
1	0	1	x
x	x	x	1

Figure 3-1. Tables of Operators in Verilog Used for Evaluating Expressions

3.5.2 Examples and Explanations

In the following descriptions, the terms variable and variable operand refer to operands or expressions that are not constant-valued expressions. This group includes wires and registers, bit-selects, and part-selects of wires and registers, function calls, and expressions that contains any of these elements.

Arithmetic Operators

Arithmetic operators perform simple arithmetic on operands.

The Verilog arithmetic operators are:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Modulus (%)

You can use the addition (+), subtraction (-), and multiplication (*) operators with any operand form (constants or variables). The addition (+) and subtraction (-) operators can be used as either unary or binary operators.

Example 3-7 shows three forms of the addition operator.

```
parameter size=8;
wire [3:0] a,b,c,d,e;
assign c = size + 2;  //constant + constant
assign d = a + 1;     //variable + constant
assign e = a + b;     //variable + variable
```

Example 3-7. Addition operation.

Relational Operators

Relational operators compare two quantities and yield a 0 or 1 value. A true comparison evaluates to 1; a false comparison evaluates to 0. All comparisons assume unsigned quantities.

The Verilog relational operators are:

- Less than (<)
- Less than or equal to (<=)
- Greater than (>)
- Greater than or equal to (>=)

Example 3-8 shows the use of a relational operator.

```
function [7:0] min( i1, i2);
input [7:0] i1, i2;
if(a<=b)
        min = i1;
else
        min = i2;
endfunction
```

Example 3-8. Relational operators.

Equality Operators

Equality operators generate a 0 if the expressions being compared are not equal and a 1 if the expressions are equal. Equality and inequality comparisons are performed bit-wise. The Verilog equality operators are:

Equality (==)

Inequality (!=)

Example 3-9 shows the equality operator used to test for a **JMP** instruction. The output signal opcode is set to 1 if the two high-order bits of instruction are equal to the value of parameter **JMP**; otherwise jump is set to 0.

```
module check_opcode_jmp( instruction, opcode);
    parameter jmp = 0;
    input [7:0] instruction;
    output opcode;
    assign opcode = (instruction[7:6] == jmp);
endmodule
```

Example 3-9. Equality operator.

Logical Operators

Logical operators generate a 1 or a 0, according to whether an expression evaluates to true (1) or false (0). The Verilog logical operators are:

Logical not (!)

Logical and (&&)

Logical or (||)

The logical not operator produces a value of 1 if its operand is zero, and a value of 0 if its operand is nonzero. The logical and operator produces a value of 1 if both operands are nonzero. The logical or operator produces a value of 1 if either operand is nonzero.

Example 3-10 shows some logical operators.

```
    module check_inst(inst, ok);
    `define ADD=0,
    `define SUB=1,
    `define MUL=2,
    `define DIV =3;
     input [7:0] inst;
    assign ok = ((inst == `ADD) | (inst == `SUB) I
          (inst == `MUL) | (`inst == `DIV));
endmodule
```

Example 3-10. Logical operators.

Bit-Wise Operators

Bit-wise operators act on the operand bit-by-bit. The Verilog bit-wise operators are:

Unary negation (~)

Binary and (&)

Binary or (I)

Binary xor (^)

Binary xnor (^~ or ~^)

Example 3-11 shows some bit-wise operators.

```
module full_adder( i1, i2, cin, sum, cout);
    input i1,i2,cin;
    output sum, cout;

    assign sum = i1 ^ i2 ^ cin;
    assign cout = (i1&i2) | (cin & (i1|i2));
endmodule
```

Example 3-11. Bit-wise operators.

Reduction Operators

Reduction operators take one operand and return a single bit. For example, the reduction and operator takes the and value of all the bits of the operand and returns a 1-bit result. The Verilog reduction operators are

Reduction and (&).

Reduction or (|)

Reduction nand(~&)

Reduction nor (-|)

Reduction xor (^)

Reduction xnor (^~ or ~^)

Example 3-12 shows the use of some reduction operators.

```
module par (in, parity, all_ones);
    input [7:0] in;
    output parity, all_ones;
    assign parity = ^in;
    assign all_ones = & in;
endmodule
```

Example 3-12. Reduction operators.

Shift Operators

The Verilog shift operators are:

Shift left (<<)

Shift right (>>)

A shift operator takes two operands and shifts the value of the first operand right or left by the number of bits given by the second operand. After the shift, vacated bits are filled with zeros. Example 3-13 shows how a right-shift operator is used to perform a division by 4.

```
module div4( dividend, res);
      input [7:0] dividend;
      output [5:0] res;
      assign res = dividend >> 2; //divide by 4 by shifting right 2 bits
endmodule
```

Example 3-13. Shift operator.

If Operators

If or Conditional operators (? :) evaluate an expression and return a value that is based on the truth of the expression. Example 3-14 shows how to use conditional operators. If the expression (op == ADD) evaluates to true, the value a+b is assigned to result; otherwise, the value a-b is assigned to result.

```
module add_or_subtract( i1, i2, op, result);
`define ADD =1'b1;
      input [7:0] a, b;
      input op;
      output [7:0] result;
      assign result=(op==`ADD) ?i1+i2: i1-i2;
endmodule
```

Example 3-14. Conditional operator.

Conditional operators can be nested to produce an if . . . else if construct. Example 3-15 shows the conditional operators ? : used to evaluate the value of op successively and perform the correct operation.

```
module alu( in1, in2, operator, result);
`define ADD=0
`define SUB=1
`define AND=2
`define OR=3
`defien XOR=4;
input [7:0] in1, in2;
input [2:0] operator;
output [7:0] result;
```

```
assign result = ((op == 'ADD) ? in1+in2 : (
            (op == 'SUB) ? in1-in2 : (
            (op == 'AND) ? in1&in2: (
            (op=='OR) ? a|b : (
             (op == 'XOR) ? in1^in2 : (a))))));
endmodule
```

Example 3-15. Nested conditional operator.

Concatenations

Concatenation combines one or more expressions to form a larger vector. In the Verilog language, you indicate concatenation by listing all expressions to be concatenated, separated by commas, in curly braces ({}). Any expression except an unsized constant is allowed in a concatenation. For example, the concatenation {1'bl,1'b0,1'b0} yields the value 3'bl00.

You can also use a constant-valued repetition multiplier to repeat the concatenation of an expression. The concatenation {1'b1,1'b0,1'b0} can also be written as {1'b1,{2{1'b0}}} to yield 3'b100. The expression {2{expr}}within the concatenation repeats expr two times. Example 4-12 shows a concatenation that forms the value of a condition-code register.

```
output [7:0] ccr;
wire half_carry, interrupt, negative, zero, overflow, carry;
 ...
assign ccr = { 2'b00, half_carry, interrupt,
negative, zero, overflow, carry };
```

Example 3-16. Concatenation operator.

Example 3-17 shows an equivalent description for the concatenation.

```
output [7:0] ccr;
assign ccr[7] = 1'b0;
assign ccr[6] = 1'b0;
assign ccr[5] = half_carry;
assign ccr[4] = interrupt;
assign ccr[3] = negative;
assign ccr[2] = zero;
assign ccr[l] = overflow;
assign ccr[0] = carry;
```

Example 3-17. Concatenation equivalent.

3.5.3 Operators in Expressions – Syntax

```
expression
    ::= primary
    | unary_operator primary
```

| expression binary_operator expression
| expression question_mark expression : expression
| string

binary_operator ::=
+ -{ } / % == != === !== && || = = & | ^ ^~

<UNARY_OPERATOR> is one of the following tokens:

+ - ! ~ & ~& | ^| ^ ~^
<BINARY_OPERATOR> is one of the following tokens:

+ - * / % == != === !== && || < <= > >= & | ^ ^~ >> <<

<QUESTION_MARK> is ? (a literal question mark).

3.6 Operands in Expressions

3.6.1 Introduction

The following kinds of operands can be used in an expression:

Numbers

Wires and registers

Bit-selects

Part-selects

Function calls

Integers

Reals

Time

Thus, any data type declared using statements discussed in Chapter 2 and its derivative can be used in an expression. The following pages discuss the usage of various types of operands in an expression.

3.6.2 Examples and Explanations

Numbers

A number is either a constant value or a value specified as a parameter. You can define constants as sized or unsized, in binary, octal, decimal, or hexadecimal bases. The default size of an unsized constant is 32 bits. The based constants can be specified using the form ‘size’ followed by ‘base’ followed by the number. Size is written as a decimal number like 32 or 64. The base can be written as quote character followed by ‘b’ or ‘h’ or ‘d’ or ‘o’ indicating binary, hexadecimal, decimal, or octal number.

```
4'b1010  –  sized binary constant with length 4 and value 1010
'b1010   –  unsized binary constant with value 1010
10       –  unsized and unbased decimal constant with value 10
2'd10    –  decimal constant 10 using the length and base notation
1'ha     –  sized and based hex constant with value a
2'o12    –  sized and based octal constant 12
```

Example 3-18. Different representations of constant 10 in Verilog.

Wires and Registers

Variables that represent both wires and registers are allowed in an expression. If the variable is a multi-bit vector and you use only the name of the variable, the entire vector is used in the expression. Bit-selects and part-selects allow you to select single or multiple bits, respectively, from a vector. These are described in the next two sections.

In the Verilog fragment shown in Example 3-19, a, b, and c are 8-bit vectors of wires. Since only the variable names appear in the expression, the entire vector of each wire is used in evaluating the expression.

```
wire [7:0] a,b,c;
assign c = a & b;
```

Example 3-19. Wire operands.

Bit-Selects

A bit-select is the selection of a single bit from a wire, register, or parameter vector. The value of the expression in brackets ([ ]) selects the bit you want from the vector. The selected bit must be within the declared range of the vector. Example 4-15 shows a simple example of a bit-select with an expression.

```
wire [7:0] a,b,c;
assign c[0] = a[0] & b[0];
```

Example 3-20. Bit-select operands.

Part-Selects

A part-select is the selection of a group of bits from a wire, register, or parameter vector. The part-select expression must be constant-valued in the Verilog language, unlike the bit-select operator. If a variable is declared with ascending indices or descending indices, the part-select (when applied to that variable) must be in the same order. The expression in Example 3-21 can also be written (with descending indices) as shown in Example 3-21.

```
assign c[7:0] = a[7:0] & b[7:0]
```

Example 3-21. Part-select operands.

Function Calls

Verilog allows you to call one function from inside an expression and use the return value from the called function as an operand. Functions in Verilog return a value consisting of one or more bits. The syntax of a function call is the function name followed by a comma-separated list of function inputs enclosed in parentheses. Example 3-22 shows the function call fcall used in an expression.

```
assign error = ! fcall1(in1, in2);
```

Example 3-22. Function call used as an operand.

Functions are described in detail in Chapter 5 on Behavioral Descriptions

Concatenation of Operands

Concatenation is the process of combining several single- or multiple-bit operands into one large bit vector. The use of the concatenation operators, a pair of braces ({}), is described in the "Concatenations" section earlier in this chapter. Example 3-23 shows two 32-bit vectors (halfword1 and halfword2) that are joined to form a 64-bit vector that is assigned to a 64-bit wire vector (byte).

```
wire [63:0] word;
wire [31:0] half_word1, half_word2;
assign word = {half_wordl, half_word2};
```

Example 3-23. Concatenation of operands.

Integers and Time

Integers and time are 32-bit and 64-bit quantities respectively and can be used as such. All the operations on vector-regs can then be used on these types of operands.

Reals

Reals can be used in expressions to evaluate different arithmetic operations. The reals do not have a bit-representation that represents them exactly and therefore they cannot be used in bit-operations without converting using the system function $bitstoreal or $realtobits.

3.6.3 Syntax of Operands in Expressions

```
primary
    ::= number
    | identifier
    | identifier [ expression ]
```

| identifier [msb_constant_expression : lsb_constant_expression]
| concatenation
| multiple_concatenation
| function_call
| (mintypmax_expression)

number
::= decimal_number
| octal_number
| binary_number
| hex_number
| real_number

real_number ::=
[sign] unsigned_number.unsigned_number
[sign] unsigned_number.[.unsigned_number]e[sign]unsigned_number
[sign] unsigned_number.unsigned_number

decimal_number ::=
[sign] unsigned_number |
[size] decimal_base unsigned_number

binary_number ::= [size] binary_base binary_digit { _ | binary_digit)
octal_number ::= [size] octal_base octal_digit { _ | binary_digit}
hex_number ::= [size] hex_base hex_digit { _ | hex_digit}
sign ::=+|-
size ::= unsigned_number
unsigned_number ::= decimal_digit { _ | decimal_digit}
decimal_base ::= ‘d | ‘D
binary_base ::= ‘b | ‘B
octal_base ::= ‘o | ‘O
decimal_number
::= A number containing a set of any of the following characters, optionally preceded by + or -
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9_

hex_digit **::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | A | B | C | D | E | F | x | X | z | Z**

BASE is one of the following tokens:
'b 'B 'o 'O 'd 'D 'h 'H

concatenation
::= { expression{ ,expression} }

multiple_concatenation
::= { expression { expression{ ,expression} } }

function_call
::= function_identifier (expression{ ,expression})

```
| system_function_identifier (expression{,expression})
| system_function

system_function_identifier
    ::=_$identifier
```

3.7 Special Considerations in Expressions

3.7.1 Constant-Valued Expressions

A constant-valued expression is an expression whose operands are either constants or parameters. The expression (op == ADD) ? a+b : a-b is not a constant-valued expression, since the value depends on the variable op. If the value of op is 1, b is added to a; otherwise, b is subtracted from a.

```
// all expressions are constant-valued,
// except in the assign statement.
module add_or_subtract( a, b, op, s);
// performs  s = a+b  if op is ADD
//          s = a-b  if op is not ADD
        parameter  size=8;
        parameter ADD=1'b1;
        input  op;
        input  [size-1:0]  a, b;
        output  [size-1:0]  s;
        assign s = (op == ADD) ? a+b : a-b; //not a constant-valued expression
endmodule
```

Example 3-24. Constant valued expressions.

3.7.2 Operators on Reals

All of the above operators apply to reals except the following:

{ } % === !== ~ & |^^~ ~^& ~& |~ | << >>

The reasons are lack of semantics for these operations for reals. In general, the bit representation for reals is not considered unique, and thus, bit-operations on these are not defined.

3.7.3 Operator Precedence

The order is:

Unary

Arithmetic

Shift

Relational

Boolean

Logical, if

3.7.4 Examples of Various Operator Usage

10% 3=1

The remainder operation of 10 by 3 gives 1.

-10%3 = -1 (sign is sign of first operand)

The remainder operation of -10 by 3 gives -1.

a < size

Compare if a is less than size

a == b

bit by bit compare of and b for equality

`b00x = `b00x

Comparison with '==' results in x

'b00x === 'b00x

Comparison with exact equality '===' result is 1

`b00x != `b00x

Compare if not equal

regA = alpha && beta

assign regA with the boolean and of alpha and beta

regA = alpha || beta

assign regA with the boolean and of alpha and beta

result = start << 2

assign result with start left shifted twice

wire [15:0] busa = drive_busa ? data: 16`bz;

assign busa with data if the drive_busa is one otherwise assign z to it from this driver.

This models a tri-state bus with control signal

{ a.b[3:0],w,3'b101}

Concatenate a, 4 bits of b with index 3 to 0, w and a constant with bit value '101'

{ a, b[3], b[2], b[1], b[0], w, 1'b1, 1'b0, 1'b1 }

This concatenation is same as above but written using bit-selects on b abd separating bits of a constant.

{ 4{w}} = {w,w,w,w}.

This is a concatenation using repeat factor.

Example 3-25. Examples of operator usage.

3.7.5 Comparisons With X's and Z's

	0	1	x	z
0	0	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

This applies to comparisons done in:

a. comparison operators
b. case statements
c. if operators

3.7.6 Expression Bit Lengths

In the earlier versions of Verilog, size of an expression was taken on the left and right side of the assignment operator could be different, but with the IEEE 1364 standardization, largest operand size is taken including the left-hand side of the assignment operator. Thus, in the following example, the three assignments to c all perform 17-bit arithmetic. In earlier versions of Verilog, the first assignment to c would result in 16-bit operation and potential loss of carry.

```
reg [15:0] a, b;
reg [16:0] c;
reg [63:0] d;

c = a + b
c = 0 + (a+b);
c= 17'b0 + a + b;
```

Example 3-26. ***Different ways to perform sized operations.***

3.8 Syntax for Expressions

```
net_1value
    ::=net_dentifier
    | net_identifier [ expression ]
    | net_identifier [ constant_expression: constant_expression ]
    | net_concatenation

reg_1value ::=
    reg_identifier
    | reg_identifier[expression]
    | reg_identifier[msb_constant_expression: 1sb_constant_expression]
    | reg_concatenation

constant_expression
    ::= constant_primary
    | unary_operator constant_primary
    | constant_expression  binary_operator constant_expression
    | constant_expression ? constant_expression : constant_expression
    | string

unary_operator ::=
    +|-|!  |  |  ~& ||| ~|| ^| ~^ |^~
```

mintypmax_expression
::= expression
| expression : expression : expression

expression
::= primary
| unary_operator primary
| expression binary_operator expression
| expression question_mark expression : expression
| string

binary_operator ::=
+ - { } / % = = != = = = != = & & || = = & | ^ ^~

QUESTION_MARK is ? (a literal question mark).

STRING is text enclosed in "" and contained on one line.

primary
::= number
| identifier
| identifier [expression]
| identifier [msb_constant_expression : lsb_constant_expression]
| concatenation
| multiple_concatenation
| function_call
| (mintypmax_expression)

number
::= decimal_number
| octal_number
| binary_number
| hex_number
| real_number

real_number ::=
[sign]unsigned_number.unsigned_number
[sign] unsigned_number.[.unsigned_number]**e**[sign]unsigned_number
[sign] unsigned_number.unsigned_number

decimal_number ::=
[sign] unsigned_number |
[size] decimal_base unsigned_number

binary_number ::= [size] binary_base binary_digit { _ | binary_digit)
octal_number ::= [size] octal_base octal_digit { _ | binary_digit}
hex_number ::= [size] hex_base hex_digit { _ | hex_digit}
sign ::=+|-
size ::= unsigned_number

unsigned_number ::= decimal_digit { _ | decimal_digit)
decimal_base ::= ‘d | ‘D
binary_base ::= ‘b | ‘B
octal_base ::= ‘o | ‘O
decimal_number
::= A number containing a set of any of the following characters, optionally preceded by + or-
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 _

hex_digit **::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | A | B | C | D | E | F | x | X | z | Z**

BASE is one of the following tokens:
'b 'B 'o 'O 'd 'D 'h 'H

concatenation
::= { expression{ ,expression} }

multiple_concatenation
::= { expression { expression{ ,expression} } }

function_call
::= function_identifier (expression{ ,expression})
| system_function_identifier (expression{ ,expression})
| system_function

system_function_identifier
::=_$identifier

3.9 Example of Register Transfer Level of Abstraction

```
//Parametrized Models for datapath using continuous assignments

`define clock_period 10

module adder(in1, in2, sum, carry, cc);

    parameter size = 32; //default is 32-bit adder
    input [size-1:0] in1, in2;
    output [1:0] cc; // condition code


    output [size-1:0] sum;
    output carry;

    assign #`clock_period{carry, sum} = in1 + in2;

    // Set condition codes
    assign cc[0] = (sum == 0)? 1:0;//Condition Code 0 - Zero Value
```

```
        assign cc[l] = (sum[size-l]);   //Condition Code 1 - Negative Value

endmodule



module multiplier(inl, in2, product, carry, cc);

        parameter size = 32; //default is 8-bit adder
        input [size-l:0] inl, in2;

        output [size-1:0] product;
        output carry;
        output [1:0] cc; // condition code


        assign #(2*`clock_period) {carry, product} = inl * in2;
        // Set condition codes
        assign cc[0] = (product == 0)? 1:0; //Condition Code 0 - Zero Value
        assign cc[l] = (product[size-l]);    //Condition Code 1 - Negative Value

endmodule

module alu;
endmodule

module multiplexor(out, control, inl, in2, in3, in4);
        parameter size = 32;
        input [1:0] control;
        input [size-l:0] inl, in2, in3, in4;
        output [size-1:0] out;

        assign out = (control == 0) ? inl :
                        ((control == 1) ? in2 :
                        ((control == 2) ? in3 :
                        ((control == 3) ? in4 :
                        'bx)));

endmodule

module bus_control;
        parameter size = 32;
        tri [size-l:0]bus;
        wire [size-l:0] data;
        wire dcontrol;
        assign bus = dcontrol ? data: 128'bz;
endmodule

module comparator(inl, in2, compare);
        parameter size=32;
```

```
        input [size-l:0] inl, in2;
        output compare;

        assign compare = (inl == in2);
endmodule

module xdetect(in, xdet);
        parameter size=32;
        input [size-1:0] in;
        output xdet;

        assign xdet = ((in == in) ? 0:1);
endmodule

module barrel_shifter(func, mode, out, in);

        parameter size = 32;
        output [size-1:0] out;
        input [size-l:0] in;

        input func, mode;

`define SHIFT 1
`define ROTATE 0
`define LEFT 1
`define RIGHT 0

`define   shift_expr (mode ? (in << 1): (in >> 1))
`define   rotate_expr (mode ? ({in[size-2:0], in[size-l]}): ({in[0], in[size-l:l]}))

        assign out = `shift_expr;

        assign out = `rotate_expr;

        assign out = func ? `shift_expr: `rotate_expr;

endmodule

module test;


        parameter alu_size = 64;

        //Build and test a 64-bit alu using the above datapath elements
        wire [alu_size-l:0] sum, prod, shift_out, out;
        wire cl,c2;

        reg [alu_size-l:0] rl, r2;
        reg func1, mode;
        reg [1:0] control;
```

```
    reg  [1:0]  func2;

    wire  [1:0]  ccodel, ccode2;

/* Build 64-bit DataPath and test it using the above Verilog modules */

    adder a  (rl, r2, sum, cl, ccodel);

    multiplier m  (rl, r2, prod, c2, ccode2);

    comparator c  (rl, r2, compare_out);

    barrel_shifter b (funcl, mode, shift_out, rl);

    multiplexor mx(out, func2, sum, prod, {compare_out,63'b0}, shift_out);



    defparam a.size = alu_size;
    defparam m.size = alu_size;
    defparam c.size = alu_size;
    defparam b.size = alu_size;
    defparam mx.size = alu_size;

    initial
        begin: testl
            rl = 5;
            r2 = 3;
            funcl  =  1;
            mode = 1;

            for (func2 =0;  func2 <=3;  func2=func2+l)
            begin
        #100
            $display("alu inputs func2 = %0d inl=%0d and in2=%0d func=%0d
mode=%0d give output out =%0d with carryl=%0d carry2=%0d and condition codel=%0d
code2=%0d",
                func2, rl, r2, funcl, mode, out, cl, c2, ccodel, ccode2);
                if (func2 == 3) disable testl;

            end


        end

endmodule
```

Example 3-27. ***Datapath design using continuous assignments RTL abstractions.***

3.10 Structural Descriptions In Verilog

3.10.1 Structural Constructs – Overview

3.10.1.1 Introduction – The structural abstraction level entails the hierarchy and connectivity descriptions of module. Modules in turn can have other modules, behavioral blocks, RTL descriptions, built-in gates and transistors (single bits), and user-defined primitives. The other modules are included using the constructs of module instantiations. The gates, switches and udps can be viewed as predefined modules. Then rules for instancing these fall into the rules of instancing modules.

3.10.1.2 Example

```
module fullAdder_s(cOut, sum, aIn, bIn, cIn);
    output      cOut, sum;
    input       aIn, bIn, cIn;
    wire    x2;
    nand    (x2, aIn, bIn),
            (cOut, x2, x8);
    xnor    (x9, x5, x6);
    nor     (x5, x1,x3),
            (x1,aIn,bIn);
    or      (x8,x1,x7);
    not     (sum, x9),
            (x3, x2),
            (x6,x4),
            (x4, cIn),
            (x7,x6);
endmodule
module ALU(Data, Address, control);
        inout Data, Address, control;
        fullAdder(cOut, sum, aIn, bIn, cIn);
        Multiplier(overflow, product, aIn, bIn, cIn);
endmodule

module CPU(Data,Address,control);
inout Data, Address, control;
    ALU a(Data, Address, control);
    FPU f(Data, Address, control);
    REGISTERS r(Data, Address, control);
    QUEUES q(Data, Address, control);
    CONTROL_UNIT c(Data, Address, control);
endmodule
```

Example 3-28. Structural design – A CPU with details of adder at the gate-level.

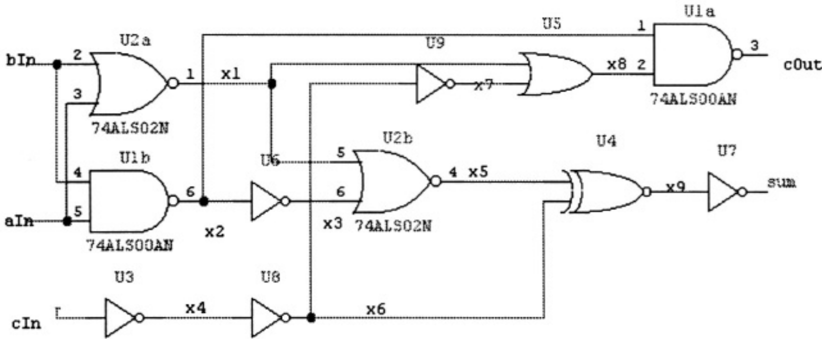


Figure 3-2. Schematics for the Adder in Example 3-28

3.10.1.3 Syntax

structural_descriptions::=
gate_declaration
||= UDP_instantiation
||= module_instantiation
||= parameter_override

3.10.2 Structural Constructs - Module Definitions

3.10.2.1 Introduction – Modules in Verilog provides framework for the hierarchical design. These form a basic design unit that can be instantiated. Module allows creating different views of the same design unit [architectural, rtl, structural]. Each module has its own namespace or scope. It also has IOs, parameters and body.

The transfer of data across a module boundary happens via well-defined interface known as the ports of a module. These have been defined in the section 2.6 on port types. The syntax section 3.5.2.3 again provides the details of syntax of ports for ready reference.

A body of a module has everything else in the language, i.e., all other constructs in Verilog discussed throughout this book. A module has data declaration of type parameters. Parameters are a way of defining generic units that can be configured in some dimensions at instance time or at redefinition. Verilog parameters allow you to customize each instantiation of a module. By setting different values for the parameter when you instantiate the module, you can cause different logic to be constructed. Back-annotating delays is the most common application of parameters.

A parameter represents constant values symbolically. The definition for a parameter consists of the parameter name and the value assigned to it. The value can be any constant-valued expression of integer or Boolean type, but not of type real. If you do not set the size of the parameter with a range definition or a sized constant, the parameter is unsized and defaults to a 32-bit quantity.

3.10.2.2 Examples – The following is an example of module definition with a parameter.

```
module adder(in1,in2, sum, carry, cc);
      parameter size  = 32; //default is 32-bit adder
      input [size-1:0] inl, in2;
      output [1:0] cc; // condition code
      output [size-1:0] sum;
      output  carry;

      assign #`clock_period{carry, sum}  = inl + in2;
      // Set condition codes
      assign cc[0] = (sum == 0)? 1:0;//Condition Code 0 - Zero Value
      assign cc[l] = (sum[size-l]);   //Condition Code 1 - Negative Value
endmodule
```

Example 3-29. Example of parametrized module definitions.

Parameters

Verilog parameters allow you to customize each instantiation of a module. By setting different values for the parameter when you instantiate the module, you can cause different logic to be constructed. A parameter represents constant values symbolically. The definition for a parameter consists of thè parameter name and the value assigned to it. The value can be any constant-valued expression of integer or Boolean type, but not of type real. If you do not set the size of the parameter with a range definition or a sized constant, the parameter is unsized and defaults to a 32-bit quantity. The typical uses of parameters are in delays and sizes, but they have broad applications.

Macromodules

Macromodules allow definition of modules that get compiled as parts of the modules in all instantiations without the module boundaries.

```
macromodule adder (inl,in2,outl);
            input [3:0] in1,in2;
            output [4:0] outl;
            assign outl = inl + in2;
endmodule
```

Example 3-30. Example of a macromodule construct.

Named Ports in Modules

```
module ex4( .in_a(a), .in_b(b), .out(z));
            input a, b;
            output z;
            ........................
```

```
endmodule

module ex5( .il(a[l]), .i0(a[0]),z );
          input [1:0] a;
          output z;
          ......................
endmodule

module ex6( .i({a,b}), z);
          input a,b;
          output z;
endmodule
```

Example 3-31. ***Named ports in modules.***

You can rename a port by explicitly assigning a name to a port expression by using the dot (.) operator. The module definition fragments in Example 3-31 show how to rename ports. The ports for module ex4 are explicitly named in_a, in_b, and out and are connected to nets a, b, and z. Module ex5 shows ports named il, i0, and z connected to nets a[l], a[0], and z, respectively. The first port for module ex6 (the concatenation of nets a and b) is named i.

3.10.2.3 Syntax

module_declaration
 ::= module_keyword module_identifier [list_of_ports];
  {module_item}
  endmodule
module_keyword
 ::= **module | macromodule**
  {module_item}
  endmodule

list_of_ports
 ::=(port {,port })

port
 ::= [port_expression]
 | .port_identifier ([port_expression])

port_expression
 ::=port_reference
 | { port_reference ,port_reference }

port_reference
 ::= port_identifier
 | port_identifier[constant_expression]
 | port_identifier [msb_constant_expression :lsb_constant_expression]

3.10.3 Structural Constructs – Module Instantiation

3.10.3.1 Introduction – This is a mechanism to create copies of the modules. It is also a mechanism to describe connectivity and a mechanism to create hierarchical design. This is a mechanism to select various views of the design units. This supports hierarchical names.

Module instantiations are copies of the logic in a module that define component interconnections. A module instantiation is done using the following form:

```
module_name instance_name1 (terminal1, terminal2), instance_name2 (terminal1,terminal2);
```

A module instantiation consists of the name of the module (module_name), followed by one or more instantiations. An instantiation consists of an instantiation name (instance_name) and a connection list. A connection list is a list of expressions called terminals, separated by commas. These terminals are connected to the ports of the instantiated module. Terminals connected to input ports can be any arbitrary expression. Terminals connected to output and inout ports can be identifiers, single- or multiple-bit slices of an array, or a concatenation of these. The bit widths for a terminal and its module port must be the same.

Named and Positional Notation

Module instantiations can use either named or positional notation to specify the terminal connections. In name-based module instantiation, you explicitly designate which port is connected to each terminal in the list. Undesignated ports in the module are unconnected. In position-based module instantiation, you list the terminals and specify connections to the module according to the terminal's position in the list. The first terminal in the connection list is connected to the first module port, the second terminal to the second module port, and so on. Omitted terminals indicate that the corresponding port on the module is unconnected. If you use an undeclared variable as a terminal, the terminal is implicitly declared as a scalar (1-bit) wire. After the variable is implicitly declared as a wire, it can appear wherever a wire is allowed

3.10.3.2 Examples of Structural Descriptions Using Modules

Example 1

Example 3-32 shows the declaration for the module m with two instances (ml and m2).

```
module next_level(bus0, bus1, out); //description of module SEQ
      input bus0, bus1;
      output out;
......
endmodule
```

```
module top( il, i2, i3, i4, ol, o2);
      input  il, i2, i3, i4;
            output  ol, o2;
      next_level  10(il,  i2, i3),   //instantiations of module SEQ
      l1(.out(ol), in.(i3),. Bus0(i2));
endmodule
```

Example 3-32. Module instantiations.

Example 2

In Example 3-33, SEQ_2 is instantiated by using named notation, as follows:

Signal OUT1 is connected to port OUT of the module SEQ.
Signal D3 is connected to port BUS1.
Signal D2 is connected to port BUS0.
SEQ_1 is instantiated by using positional notation, as follows:
Signal DO is connected to port BUS0 of module SEQ.
Signal Dl is connected to port BUS1.
Signal OUT0 is connected to port OUT

Example 3-33. Module definitions and instantiation – hierarchical design example.

Features:

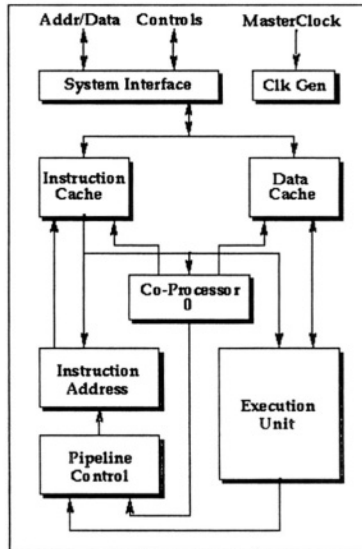
Low Power Dissipation:
- 1.5W (normal)
- 0.4W (reduced power)
- 0W (instant-off)

High Performance:
- 55 SPECint92
- 30 SPECfp92

High integration on-chip:
- 16K I-cache, 8K D-cache
- Unified datapath
- 32 double-entry J-TLB
- 2 entry micro I-TLB

Power Management Features:
- Reduced Power Mode
- Instant On/Off

R4000 compatibility:
- R4000PC software compatible
- R4000PC hardware compatible (R4200PC only) packaging



Applications:

X-terminals
Laser printers
Notebook computers
Low-cost desktop computers
Low-cost workstations
Factory automation
Networking: routers, bridges
Graphics acceleration
Personal Digital Assistants

Packaging Options:

R4200LP (208-pin PQFP)
R4200PC (179-pin C-PGA)

Figure 3-3. Top Level Block Diagram of r4200

```
module r4200 (SysAD, SysADC, SysCmd, SysCmdP, ValidIn_, ValidOut_,
        SysRqst_, ExtRqst_, Release_, RdRdy_, WrRdy_, MasterClock,
        MasterOut, TClock, RClock, SyncOut, SyncIn, Reset_,
        ColdReset_, ByPassPLL_, BigEndian_, DataRate_, Int_,
        NMI_, Status, JTDI, JTCK, JTDO, JTMS);

    // The following input, output, or inout information is from
    //Tables 1-5 thru 1-9.

    inout   [63:0]  SysAD;
    inout   [7:0] SysADC;
    inout   [8:0] SysCmd;
    inout           SysCmdP;
    input           ValidIn_;
    output          ValidOut_;
    input           ExtRqst_;
    output          Release_;
    output          MasterOut;
    output          SysRqst_;
    input           RdRdy_;
    input           WrRdy_;
    input           MasterClock;
    output          TClock;
    output          RClock;
    output          SyncOut;
    input           Syncln;
    input           Reset_;
    input           ColdReset_;
    input           ByPassPLL_;
    input           BigEndian_;
    input           DataRate_;
    input   [4:0]   Int_;
    input           NMI_;
    output  [3:0] Status;
    input           JTDI;
    input           JTCK;
    output          JTDO;
    input           JTMS;

    // Declarations needed for signals assumed to exist.
    // These signals are required for proper functioning
    // of various modules.
    reg  [63:0]     gpr_in;
    reg   [4:0]gpr_addr;
    reg             gpr_write;
    reg             gpr_read;
    wire [63:0]     gpr_out;
    wire            clk;
    reg      [63:0]     fpgr_in;
    reg      [4:0]fpgr_addr;
```

```
reg                 fpgr_write;
reg                 fpgr_read;
wire [63:0]    fpgr_out;
reg      [143:0]    d_cache_in;
reg       [8:0]d_cache_addr;
reg                 d_cache_write;
reg                 d_cache_read;
wire [143:0]   d_cache_out;
reg      [23:0]     d_tag_in;
reg      [8:0]d_tag_addr;
reg                 d_tag_write;
reg                 d_tag_read;
wire [23:0]    d_tag_out;
reg      [287:0]    i_cache_in;
reg       [8:0]i_cache_addr;
reg                 i_cache_write;
reg                 i_cache_read;
wire [287:0]   i_cache_out;
reg      [22:0]     i_tag_in;
reg       [8:0]i_tag_addr;
reg                 i_tag_write;
reg                 i_tag_read;
wire [22:0]    i_tag_out;
reg      [63:0]     IVA;
reg      [63:0]     DVA;
wire [31:0]    DBus;
wire [31:0]    Cache_data;
reg       [3:0] Interrupts;
wire [20:0]    I_PFN;
wire [20:0]    D_PFN;
wire [4:0] Exceptions;
wire [3:0] Status;

// General purpose register module.
gpr gpr(gpr_in, gpr_addr, gpr_write, gpr_read, gpr_out, clk);

// FP general purpose register module.
fpgr fpgr(fpgr_in, fpgr_addr, fpgr_write, fpgr_read, fpgr_out, clk);

// PC, Hi, Lo, FCR0, FCR31 and LLbit register.

reg      [31:0]     prog_counter;
reg      [31:0]     Hi;
reg      [31:0]     Lo;
reg      [31:0]     FCR0;
reg      [31:0]     FCR31;
reg                 LLbit;

// Flush buffer. Refer to fig. 1-18.
reg      [148:0]  flush_buffer  [0:1];
```

```
        // Data cache module.
        d_cache d_cache(d_cache_in, d_cache_addr, d_cache_write, d_cache_read,
              d_cache_out, d_tag_in, d_tag_addr, d_tag_write,
              d_tag_read, d_tag_out, clk);

        // Instruction cache module.
        i_cache i_cache(i_cache_in, i_cache_addr, i_cache_write, i_cache_read,
              i_cache_out, i_tag_in, i_tag_addr, i_tag_write,
              i_tag_read, i_tag_out, clk);

        // CP0 module.
        wire      [20:0]     IFN;
        wire      [20:0]     DFN;

        cop0 cp0(IVA, DVA, DBus, Cache_data, Interrupts, IFN, DFN,
              Exceptions, Status);

        execution_unit exec_unit(Dbus, nReset, pClock, sys_inst, selecte);
        pipeline_control pipe_ctrl (Dbus, nReset, pClock, sys_inst, selectp, control);
    endmodule
```

Example 3-34. A structural model of R4200 processor with declarations and instances at top-level

Example 3

In this example, we describe the input-outputs and top-level units of ultrasparcIIi. Figure 3-4 on page 82 depicts the block diagram. A single-chip implementation, UltraSparc Iii integrates the following components:

- independently clocked PCI interface fully decoupled from main CPU

- PCI bus module (PBM)
- PCI I/O Memory management unit (IOM)
- External cache control unit (ECU)
- Memory controller unit (MCU)
- 16 KB instruction cache (ICU)
- 16 KB data cache (DCU)
- prefetch, branch prediction and dispatch unit (PDU)
- 64 entry instrutcion translation lookaside buffer (ITB) and 64-entry data TLB (DTB)
- integer execution unit (IEU)
- Floating point and graphics unit (FGU)
- Load and Store Buffer Unit (LSU)

```
module sparc_cpu (
devsel_l, frame_l,irdy_l,      par,    perr_l, stop_l,
trdy_l,  tdata,  tpar,         ad,     cbe_l,   edata,
mem_data, edpar,  p215clk,     rmtv_sel, ad31z_en, clksel,
```

```
clka,    clkb,   ext_event,      pciclk,   pci_ref_clk,
pllbypass,        ram_test, spare,  itb_test_mode,   spare3v,   sram_clk,
sram_clk_pos,     sys_reset_tck,    tdi,          tms,     trst_l,
upa_clk_neg,      upa_clk_pos, p_reset_l, x_reset_l,  p_reply,  req_l,
int_num,          sb_empty, ecache_2, adr_vld,  doe_l,   dsyn_wr_l,
epd,           15clk,   mem_we_l, pmo,   rst_l,   sb_drain,
serr_l, stop_clock,      tdo,          toe_l,   tsyn_wr_l,
 xcvr_oea_l,      xcvr_oeb_lxcvr_sel, mem_addr,       ecat,
ecad,   mem_cas_l,       temp_sen, xcvr_rd_cntl,
xcvr_wr_cntl, ssysadr, s_reply,  xcvr_clk
gnt_l,           mem_rasb_l, mem_rast_l,    bytewe_l
);

inout           irdy_l;
inout           par;
inout           perr_l;
inout           stop_l;
inout           trdy_l;
inout     [15:0]    tdata;
inout     [1:0]     tpar;
inout     [31:0]    ad;
inout     [3:0]     cbe_l;
inout     [63:0]    edata;
inout     [71:0]    mem_data;
inout     [7:0]     edpar;

output          p215clk;
input           rmtv_sel;
input           ad31z_en;
input           clksel;
input           clka;
input           clkb;
input           ext_event;
input           pciclk;
input           pci_ref_clk;
input           pllbypass;
input           ram_test;

input           spare;
input           itb_test_mode;
input           spare3v;
input           sram_clk_neg;
input           sram_clk_pos;
input           sys_reset_l;
input           tck;
input           tdi;
input           tms;
input           trst_l;
input           upa_clk_neg;
input           upa_clk_pos;
input           p_reset_l;
```

```
input           x_reset_l;
input   [1:0]   p_reply;
input   [3:0]   req_l;
input   [5:0]   int_num;
input [1:0]   sb_empty;
input           ecache_22_mode;

input           s_clk_mode;


output          adr_vld;
output          doe_l;
output          dsyn_wr_l;
output          epd;
output          15clk;
output          mem_we_l;
output          pmo;
output          rst_l;
output          sb_drain;
output          serr_1;
output          stop_clock;
output          stop_clock;

output          tdo;
output          toe_I;
output          tsyn_wr_l;
output          xcvr_oea_l;
output          xcvr_oeb_l;
output          xcvr_sel_l;

output   [12:0]   mem_addr;
output   [14:0]   ecat;
output   [17:0]   ecad;
output   [1:0]    mem_cas_l;
output   [1:0]    temp_sen;
output   [1:0]    xcvr_rd_cntl;
output   [1:0]    xcvr_wr_cntl;

output   [28:0]   sysadr;
inout    [28:0]   sysadr;

output   [2:0]    s_reply;
output   [2:0]    xcvr_clk;
output   [3:0]    gnt_l;
output   [3:0]    mem_rasb_l;
output   [3:0]    mem_rast_l;
output   [7:0]    bytewej;

wire [19:0]   UNCONN_ac;
wire [19:0]   UNCONN_ac;
wire [15:0]   UNCONN_b;
```

```
wire ecu_13se;
wire.........

/* Data Management Unit */
sparcdmu dmu (
   .dmu_utlb_inv_r0          (dmu_utlb_inv_r0),
   .io_ec_edsyn0_r1          (io_ec_edsyn0_r1),
   .io_ec_edsyn4_rl          (io_ec_edsyn4_rl),
    ......
   .dmu_13rsttrien           (dmu_13rsttrien),
   .dmu_spare_out            (dmu_spare_out),
   .dmu_13se                 (dmu_13se),
    ...........
   .lsu_any_misaligned_n 1       (lsu_any_misaligned_nl),
   .lsu_flush_nl             (lsc_flush_nl),
   .lsu_ldd_nl               (lsu_ldd_nl),
   .ieu_trap_level           (ieu_trap_level_vl),
   .lsu_dmu_enable           (lsu_dmu_enable),
    ...................
   .lsc_dmu_prefetch_c       (lsc_dmu_prefetch_c)
   );

/* External Cache Unit */
sparcecu ecu (
   .zzz_asi_data2            (zzz_asi_data2[63:0]),
   .clkctl_e_clk_en          (clkctl_e_clk_en_v2_r0),
   .ecu_scan_in              (ex_scan_out),
   .ecu_spare_in             (ldbctl_spare_out),
   . errorctl_scan_out       (errorctl_scan_out),
   .ieu_pstate_priv_v1       (ieu_pstate_priv_v1_r1),
   .io_ec_edsyn              ({io_ec_edsyn7_rl,io_ec_edsyn6_rl,
   io_ec_edsyn5_rl   io_ec_edsyn4_rl,
   io_ec_edsyn3_rl,io_ec_edsyn2_rl,   io_ec_edsynl_rl,io_ec_edsyn0_rl}),
   .io_ec_tdata              (io_ec_tdata[15:0]),
   .iom_ecu_pa               ({iom_ecu_pa[33:4], iom_ecu   pa_3}),
   ....................
   .iom_ecu_pa_type          (iom_ecu_pa_type),
   .12clk                    (s_12clk_iobotvl),
   .12clk_top                (s_12clk_iobotvl),
   .ldb_ec_atomic            (ldb_ec_atomic),
   . ldb_ec_cacheable        (ldb_ec_cacheable),
   .ldb_ec_pf                (ldb_ec_pf),
      ..........
   .south_12rsttrien         (south_12rsttrien),o_ec_edsyn 12_r0,
   (imu_sprrpt_gnd_l),       .lsc_jmpl_return_nl       (lsc_jmpl_return_nl),
   .lsc_dmu_prefetch_c       (Isc_dmu_prefetch_c)
   );

sparcex ex (          .movx_enable          (movx_enable),
   .movx_enable              (movx_enable),
   x_4_rptr_out              (pbm_mcu_asi_csr_rl [2]),
```

```
   ...............
   .ex_5_rptr_out              (pbm_mcu_csr_addr_rl [0]),
   .ex_6_rptr_out              (pbm_mcu_csr_addr_rl [1]),
   .......  );

/* Floating an Graphics Unit */
sparcfgu fgu (
   .fgu_0_rptr_out          (mcuctl_mcu_io_upa_addr_18_rl),
   .fgu2_lfeed_b_63         (zzz_asi_datal [62]),
     ........
   .fgu2_lfeed_b_64         (zzz_asi_datal [63])
   );

sparciblock iblock (
   .recover_movx_stall_c          (recover_movx_stall_c),
   .w_12clk_ibk2                  (w_12clk_ibk2),
  ..........
   .w_12clk_ibk5                  (w_12clk_ibk5),
   .w_12clk_ibk4                  (w_12clk_ibk4)
   );

/* Instruction Management Unit */
sparcimu imu (
   .spr_io_obs_bus_rl   (spr_io_obs_bus_rl[14:0]),
   .imu_sprrpt_gnd_l   (imu_sprrpt_gnd_l),
   .imu_sprrpt_gnd_l    (imu_sprrpt_gnd_l),
   ………………………
   . imu_sprrpt_in_l_l   ({8 {imu_sprrpt_gnd_l}}),
   .spr_io_obs_bus_r0   (spr_io_obs_bus_r0[14:0]),
   . io_ram_test_wr_ram (io_ram_test_wr_ram)
   );

/* Buffers */
sparci_sb_iob iob (
   .iob_rtm_in          (iol_rtm_out),
   .iob_rtm_out         (iob_rtm_out),
   .iob_rst_pin_en_in      (iol_rst_pin_en_v2),
   .iob_rst_pin_en_put    (iob_rst_pin_en_out),
   .io_misc_bidir         (io_misc_bidir_r0[14:0]),
   .clkctl_e_clk_en         (clkctl_e_clk_en_v2_rl),
   ......................
 . io_ecache_22_mode      (io_ecache_22_mode)
   );

sparci_sb_iol iol (
   .tdo               (tdo),
   .tck               (tck),
   .tms                (tms),
   ......................
   .pbm_p212clk_iolb     (pbm_p212clk_iolb),
   .p2clk_en              (clkctl_p_clk_en_v2_r0)
```

```
   );

sparci_sb_ior ior (

   .sysadr            (sysadr[28:18]),
   .p_reply            (p_reply[l:0]),
   .............
   .ior_ring_osc         (ior_ring_osc),
   .ior_scan_out         (ior_scan_out)
   );

sparci_sb_iot iot (
   .mem_addr            (mem_addr[12:0]),
   .mem_data            (mem_data[9:0]),
   .xcvr_rd_cntl         (xcvr_rd_cntl[1:0]),
   .xcvr_wr_cntl         (xcvr_wr_cntl[l :0]),
   ................

   );

sparcldbctl Idbctl (
   .lsu_raw_enq_nl_l           (lsu_raw_enq_nl_l),
   .lsu_stall_ll               (lsu_stall_ll),
db_13clk                 (ldb_13clk)
......
   .ldb_13clk                 (ldb_13clk)
   );

sparcpadp padp (
   .PADP_LFEED_R0)              (PADP_LFEED_RO),

 ............
   .dmu_13se                 (dmu_13se),
   .dmu_13clk                (dmu_13clk)
   );


sparcrptr rptr (
   .rptr_scan_in        (rptr_scan_in),
  ...........
   .spare314_in         ({{13{spare3_gnd}}, rmtv_sel_r0}),
   .spare414_in         ({{12{spare4_gnd}},
   );

sparcpc pc (
   .obs_tap_bus_0      (obs_tap_bus_0_l 1),

 .............
   .ibk_13se_v4        (ibk_13se_v4),
   .ibk_13clk_v4      (ibk_13clk_v4)
```

```
  );

sparcrst rst (
  .rst_reset_v1  _r0  (rst_reset_v1_r0),

.............
  .io_ec_sdb_ueh_r0   (UNCONN_eq),
  .io_ec_sdb_uel_r0  (UNCONN_er)
  );

sparcsbdp sbdp (
  .io_edata               ({io_edata_r2_v4[63:48],io_edata_r2[47:40],

.............
  .lsu_imu_enable          (lsu_imu_enable_r0),
  .sbdp_ram_test_out      (sbdp_ram_test_out[7:0])
  );

sparcsbdpctl sbdpctl (
  .obs_tap_bus2_l3         (obs_tap_bus_2_l3),
.............
  .sbdpctl_spare_in        (fgu_spare_out),
  .sbdpctl_spare_out        (sbdpctl_spare_out)
  );

sparcstbctl  stbctl  (
  .ec_data_index_sel          (ec_data_index_sel[3:0]),

.............
  .ibk_13clk_v4       (ibk_13clk_v4),
  .rmtv_sel         (rmtv_sel_rl)     );

sparcrst rst (
  .rst_reset_v1_r0    (rst_reset_v1_r0),

..............
  .io_ec_sdb_ueh_r0  (UNCONN_eq),
  .io_ec_sdb_uel_r0   (UNCONN_er)
  );

sparcsbdp sbdp (
  .io_edata                  ({io_edata_r2_v4[63:48],io_edata_r2[47:40],
...........
  .sbdp_ram_test_out         (sbdp_ram_test_out[7:0])
  .sbdp_ram_test_out      (sbdp_ram_test_out_r0[7:0])      );

sparcsbdpctl  sbdpctl  (
  .obs_tap_bus2_l3          (obs_tap_bus_2_l3),
............
  .sbdpctl_spare_out        (sbdpctl_spare_out)
  .sbdpctl_spare_out        (sbdpctl_spare_out)     );
```

```
sparcstbctl stbctl (
   .ec_data_index_sel           (ec_data_index_sel[3:0]),
 ............
   .ldb_13clk                 (ldb_13clk)
   .ldb_13clk                 (ldb_13clk)
   );

sparctap tap (
   .tap_io_tdo           (tap_io_tdo),
    ............
   .tap_sprrpt_in_t_t    ({mcu_io_upa_addr[17:0],
 tap_inst_isr_r0,mcu_io_addr_valid,mcu_io_assrt_s_reply})
   );


sparctr tr (
   .ieu_done                  (ieu_done),
   .ieu_retry                 (ieu_retry),
 ............
   .rmtv_sel                  (rmtv_sel_rl)
   );

sparcmisc_cen misc_cen (
   .c_12clk_exl    (c_12clk_exl),
   .c_12clk_ex2    (c_12clk_ex2),
 ............
   .rpt_in       ({23{misc_cen_gnd}})
   );

/* Phase Locked Loop */
sparcp_pll pll (
   .11clk              (11 clk),
 ............
   .left_13clk_v2      (left_13clk_v2)
   .left_13clk_v2      (left_13clk_v2)
   );

/* Clock Control */
sparcclkctl clkctl (
   .clkctl_spare_out      (clkctl_spare_out),
   .clkctl_spare_out      (clkctl_spare_out),
 ............
   .left_13clk_v2       (left_13clk_v2)
   .left_13clk_v2         (left_13clk_v2)
   );
/* Spare Dcache Control Unit */
sparcr_dcu dcu(
   .din2             (Isu_tag2_din[29:0]),
   .we2              (padp_tag2_we),
   .tap_ram_wee_vl_r2(tap_ram_wee_vl_r2),
```

```
.............
   .doutl           ({dcu_dct_tagl[40:13],dcu_dct_validl[l:0]}),
   .dout2             ({dcu_dct_tag2[40:13],dcu_dct_valid2[l :0]})
   );


/* Spare Icache Control Unit */
sparcr_icu icu(
   .din2              (Isu_tag2_din[29:0]),
   .we2              (padp_tag2_we),
   .tap_ram_wee_vl_r2(tap_ram_wee_vl_r2),
.............
   .doutl           ({dcu_dct_tagl [40:13],dcu_dct_validl [1:0]}),
   .dout2             ({dcu_dct_tag2[40:13],dcu  dct  valid2[l:0]})
   );


sparcm_icrf corerf (
   .iex_13rsttrien_v2   (iex_13rsttrien_v2),
   .iex_13se_v2         (iex_13se_v2),
   ........
   .iblock_cwp_valid_g  (iblock_cwp_valid_g),
   .spr_cwp_bypass_g     (spr_cwp_bypass_g)
   );

sparcg_13buf_v100 13buf (
   .13clk (ldb_13clk),
   .13clk(ldb_13clk),
   .12clk (s_12clk_ldb)
   );

/**** Memory and UPA Control unit Data PAth */
sparcmcudp mcudp (
   .def_csr_mdp_data_mux_en         (def_csr_mdp_data_mux_en),
.............
   .mdp_miu_ecc_en                 (mdp_miu_ecc_en),
   .mdp_mrwctl_simm_present         (mdp_mrwctl_simm_present[4:0])     );

/**** Memory and UPA Control unit Control ***/
sparcmcuctl mcuctl (
   .clkctl_s_clk_en                (clkctl_s_clk_en_v2_r0),
   .ecu_mcu_addr                     (ecu_mcu_addr[33:3]),
.............
   .vdb_wr_sel                     (vdb_wr_sel[7:0])     );

/* PCI Bus Module */
sparcpbm pbm (
   .13clk                   (ldb_13clk),
   .sbr_rst                  (rst_reset_v2),
.............
   .vdb_rd_sel                       (vdb_rd_sel[7:0]),
```

```
    ,vdb_wr_sel                    (vdb_wr_sel[7:0])    );

/* PCI Synchronization Module */
sparcpcisync pcisync (
    .pcisync_iom_pa_gnt   (pcisync_iom_pa_gnt),
    .pcisync_ecu_pa_avail    (pcisync_ecu_pa_avail),
 ............
    .pcisync_pci_scan_in     (pcisync_pci_scan_in),
    .pcisync_cpu_scan_in  (pcisync_cpu_scan_in)
    );


/* Prefetch and Dispatch of Instructions */
sparcpdp pdp (
    .pdp_mcu_pio_data          (pdp_mcu_pio_data[63:0]),
    .pdp_mcu_dma_data       (pdp_mcu_dma_data[63:0]),
 ............
    );

sparcpierst pierst (
    .13clk                 (1db_13clk),
    .13clk                  (1db_13clk),
    .left_13clk_v2          (left_13clk_v2),
 ..............
    );

sparcpiect1 piect1 (
    ,pbm_pie_csr_rd_en  (pbm_pie_csr_rd_en),
 .............
    .piect1_1feed_t    ({ex_20_rptr_in,....  })
    );


/* Instruction Translation Lookaside buffer */
sparcr_itb itb (
    . itb_data_out     (itb_iom_data_out),
    .tlb_hit          (itb_iom_hit),
    .cam_do           (itb_iom_cam_do),
    .comp_o            (itb_iom_comp_o),
    .itb_p213clk      (itb_p213clk),
    .........................................
    .so              (itb_scan_out),
    .west_12se           (west_12se),
    .tap_ram_wee          (tap_ram_wee_v2),
    .ex_p212clk_itb      (ex_p212clk_itb),
    . itb_scan_capture_ct1  (itb_scan_capture_ctl),
    .itb_test_mode        (io_itb_test_mode),
    .itb_write_mode_ctl     (itb_write_mode_ctl)
    );

/* Data Translation Lookaside buffer */
```

```
sparcr_dtb dtb (
    .itb_data_out     (itb_iom_data_out),
    .tlb_hit          (itb_iom_hit),
    .cam_do           (itb_iom_cam_do),
    .comp_o           (itb_iom_comp_o),
    .itb_p213clk      (itb_p213clk),
    ..........................................
    .so               (itb_scan_out),
    .west_12se              (west_12se),
    .tap_ram_wee            (tap_ram_wee_v2),
    .ex_p212clk_itb         (ex_p212clk_itb),
    .itb_scan_capture_ctl  (itb_scan_capture_ctl),
    . itb_test_mode         (io_itb_test_mode),
    .itb_write_mode_ctl     (itb_write_mode_ctl)
    );

/* I/O and Memory managament unit */
sparciom iom (
    .iom_ecu_pa              (iom_ecu_pa[33:0]),
    .iom_ecu_pa              (iom_ecu_pa[33:0]),
    .iom_ecu_pa_3             (iom_ecu_pa_3),
    .iom_ecu_pa_type          (iom_ecu_pa_type),
    .iom_ecu_cacheable         (iom_ecu_cacheable),
 .............
    .iom_spare_in            (iom_spare_in)
    .iom_spare_in            (pbm_spare_out)
    );

sparcmid_buf mid_buf (
    .11clk  (11clk),
    .pre_clk (pre_clk)
    );

/* Instruction execution unit */
sparcieu ieu (
        .clk (clk),
        ......
endmodule
```

Example 3-35. A structural model of UltraSPARC-IIi.

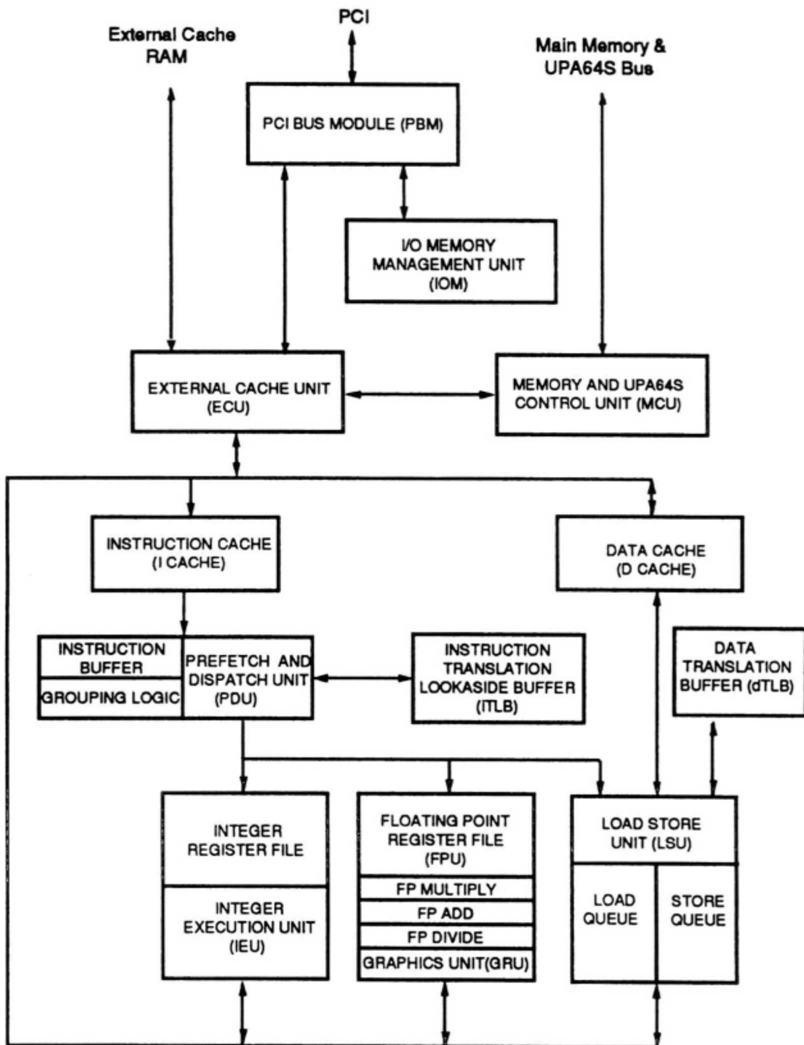


Figure 3-4. UltraSPARC-IIi Block Diagram

3.10.3.3 Module Instantiation Syntax

```
module_instantiation
        ::= module_identifier  [parameter_value_assignment]
             module_instance{ ,module_instance}

parameter_value_assignment
        ::= # ( expression{  ,expression} )

module_instance
        ::= name_ of_ instance ( [list_of_module_connections] )

list_of_module_connections
        ::= ordered_port_connection{ , ordered _port_connection}
        | named_port_connection{ ,named_port_connection}

ordered  _port_connection
        ::= [expression]

named_port_connection
        ::= . port_identifier ( expression )
```

3.11 Exercises

1. Instantiate the two modules in Example 3-1 in a test module. Apply stimulus and see the two modules give out exactly the same results. Use the Example 1-6 as a guideline to writing this test module.

2. Convert the following RTL description into a behavioral statement using case statement.

```
        module multiplexor(control, in1, in2, in3, in4, out);
    input  [1:0]  control;
    input in1, in2, in3, in4;
    output out;

    assign out = (control == 0)  ? in1:
                      ((control == 1) ? in2 :
                      ((control == 2) ? in3 :
                      ((control == 3) ? in4 :
                      'bx)));
    endmodule
```

3. Given that a, b and c are declared as below:

```
reg [7:0] a, b; reg [8:0]c; reg [15:0] d;
```

Evaluate the following expressions:

```
i)   a = 255; b = 255; c = a + b; /* Evaluate c */
ii)  c = 9'b0 + a + b; /* Evaluate c */
iii)d={a,b};/* Evaluate d*/
iv) c = &b;
```

4. The barrel shifter in the Example 3-7 shifts or rotates by 1 bit. Modify the barrel shifter to have a module port that contains the number of bits to be shifted. [You will modify the module statement as well as the assign statement. Port declarations will also be added for the new port]. Create a testbench for this, test and obtain results.

4 SEMANTIC MODEL FOR VERILOG HDL

4.1 Introduction

Verilog HDL provides a mechanism for specifying digital hardware precisely. This specification can then be used for verification by a simulator, by a formal verification tool, or for synthesis or timing analysis or any other process pertaining to the design. The syntax of the language is defined precisely by a formal notation like BNF. However, the semantics definition of a language does not have a formal notation developed and usable in the realm of computer science. Thus, to help understand the commonly understood semantics of a program in a language, some abstract models are used. For Verilog HDL, this model for simulation has been defined to some extent in the original definition and then by IEEE 1364. Here a model is provided based on the original ideas that we had while designing and implementing the language.

This model is based on the notion of value-changes and evaluations and their sequencing. The simulation can be thought of a series of value-changes on signals (nets and regs) intermixed with evaluations of blocks of Verilog Code (that describe circuit elements).

4.2 Example

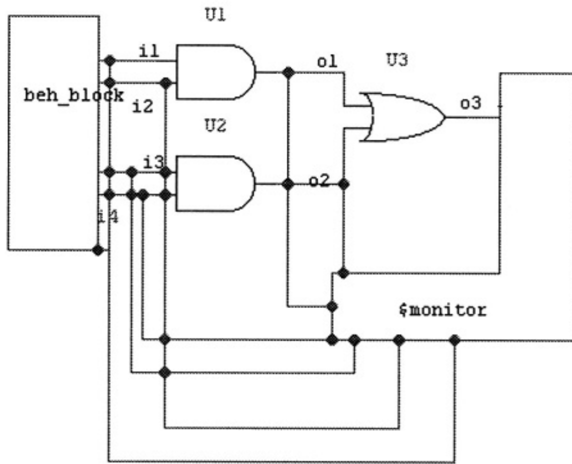


Figure 4-1. Schematics for Example 4-1

For example, take an andor circuit where 2 and gates are connected to an or gate. The Verilog model is given below:

```
module andor;
        reg i1, i2, i3, i4;
        and #2 a1 (o1, i1, i2), a2 (o2, i3, i4);   or r1 (o3, o1, o2);
        initial
        begin : beh_block
            $monitor("Sim Time=%d i1=%d i2=%d i3=%d i4=%d o1=%d o2=%d
                o3=%d", $time,i1, i2, i3, i4, o1, o2, o3);
            #25 i1 = 1;      #25 i2 = 0;
            #25 i3=1;        #25 i4=1;
            #100 $finish;
        end
endmodule
```

Example 4-1, A sample design with structure and behavior.

In this example, behavioral representation is used for stimulus and result capturing, while gates are used to describe the design. The rules of semantics in a behavioral block are the same whether it is used for test bench or design description, and the explanation in the following sections will apply to examples containing behavioral description of design.

4.3 Simulation with Full Analysis

First, let us run this by single stepping through our simulator. We will also set $monitor on all signals in the circuit. In some ways, the single stepping is stepping through the sequence of evaluations. $monitor enables us to capture all value changes. Thus, with these two traces, we can see some series of evaluations and value-changes that makes the simulation run for a given Verilog model. Capturing this in a log file and visiting this later also gives us the insight into what is happening in the simulation cycles that lead to the whole simulation. Here we list an ideal log file while running on your favorite simulator will typically produce a subset of this information. The log file on a sample simulator is also shown for comparison.

4.4 Log of a Typical Simulator

```
VeriWell for Win32 HDL <Version 2.0.5> Wed Jul 03  14:23:31  1996
..................
        Compiling source file : EX_SIMMD.V
        No errors in compilation
        Top-level modules:
        andor

        C1>.
        L6 "EX_SIMMD.V" (andor): INITIAL
        L7 "EX_SIMMD.V" (andor): BEGIN
        L9 "EX_SIMMD.V" (andor): $monitor ("Sim Time-d i1=d i2=d i3=d i4=d
                o1=d o2=d o3=d", $time (), i1, i2, i3, i4, o1, o2, o3)
        L7 "EX_SIMMD.V" (andor): #25
        L3 "EX_SIMMD.V" (andor): WIRE o1 >>> NET = 1'hx, 0
        L3 "EX_SIMMD.V" (andor): WIRE o2 >>> NET = 1'hx, 0
//---------------------------------------------------------------------------------
slide 5
        L4 "EX_SIMMD.V" (andor): WIRE o3 >>> NET = 1'hx, 0
        SimTime=                0 i1=x i2=x i3=x i4=x o1=x o2=x o3=x
        SIMULATION TIME IS 25
        L7 "EX_SIMMD.V" (andor): #25 »> CONTINUE
        L10 "EX_SIMMD.V" (andor): i1 = 1;
        Sim Time=               25 i1=1 i2=x i3=x i4=x o1=x o2=x o3=x
        L10 "EX_SIMMD.V" (andor): #25
        SIMULATION TIME IS 50
        L10 "EX_SIMMD.V" (andor): #25 >>> CONTINUE
        L11 "EX_SIMMD.V" (andor): i2 = 0;
        Sim Time=               50 i1=1 i2=0 i3=x i4=x o1=x o2=x o3=x
        SIMULATION TIME IS 52
        L3 "EX_SIMMD.V" (andor): GATE >>> 1'b0
        L11 "EX_SIMMD.V" (andor): #25
        L3 "EX_SIMMD.V" (andor): WIRE o1 >>> NET = 1'h0, 0
        Sim Time=               52 i1=1 i2=0 i3=x i4=x o1=0 o2=x o3=x
        SIMULATION TIME IS 75
        L11 "EX_SIMMD.V" (andor): #25 >>> CONTINUE
        L12 "EX_SIMMD.V" (andor): i3 = 1;
```

```
Sim Time=           75 i1=1 i2=0 i3=1 i4=x o1=0 o2=x o3=x
L12 "EX_SIMMD.V" (andor): #25
SIMULATION TIME IS 100
L12 "EX_SIMMD.V" (andor): #25 >>> CONTINUE
L13 "EX_SIMMD.V" (andor): i4= 1;
Sim Time=           100 i1=1 i2=0 i3=1 i4=1 o1=0 o2=x o3=x
SIMULATION TIME IS 102
L3 "EX_SIMMD.V" (andor): GATE >>> 1'b1
L13 "EX_SIMMD.V" (andor): #100
L3 "EX_SIMMD.V" (andor): WIRE o2 >>> NET = 1'h1, 1
L4 "EX_SIMMD.V" (andor): GATE >>> 1'b1
slide 6 part 1
L4 "EX_SIMMD.V" (andor): WIRE o3 >>> NET = 1'h1, 1
Sim Time=           102 i1=1 i2=0 i3=1 i4=1 o1=0 o2=1 o3=1
SIMULATION TIME IS 200
L13 "EX_SIMMD.V" (andor): #100 >>> CONTINUE
L14 "EX_SIMMD.V" (andor): $finish
Exiting VeriWell for Win32 at time 200
...... Thank you for using VeriWell for Win32
```

Example 4-2. ***Log of a typical simulator with tracing.***

4.5 Log of an Ideal Simulator

The ideal log file that will take us through the tour of a event driven simulation run of Verilog HDL for the example is provided below:

```
-------------------------------------------------------------------------------
IDEAL SIMULATION LOG FOR andor circuit with tracing and monitor
EVENT DRIVEN SIMULATION ENGINE OF VERILOG HDL
      EXPLAINED
Top-level modules:
      andor

L6 "EX_SIMMD.V" (andor): EVALUATE beh_block - INITIAL
L7 "EX_SIMMD.V" (andor): EVALUATE beh_block - BEGIN : beh_block
L9 "EX_SIMMD.V" (andor): EVALUATE beh_block - $monitor
          ("Sim Time=d i1=d
      i2=d i3=d i4=d o1=d o2=d o3=d", $time (), i1, i2, i3, i4, o1, o2, o3)
L10 "EX_SIMMD.V" (andor): #25 >>> SUSPEND beh_block
      [for 25 time units]
Sim Time=           0 i1=x i2=x i3=x i4=x o1=x o2=x o3=x
Finished Activity at time 0; Look for next time of activity; Advance
      time to 25
-------------------------------------------------------------------------------
SIMULATION TIME IS 25
L10 "EX_SMMD.V" (andor): #25 >>> CONTINUE [EVALUATION
      of beh_block]
L10 "EX_SIMMD.V" (andor): i1 = 1;
Sim Time=           25 i1=1 i2=x i3=x i4=x o1=x o2=x o3=x
```

```
L11 "EX_SIMMD.V" (andor): #25 >>> SUSPEND beh_block
      [for 25 time units]
Finished Activity at time 25; Look for next time of activity;
Advance time to 50

SIMULATION TIME IS 50
L11 "EX_SIMMD.V" (andor): #25 >>> CONTINUE
L11 "EX_SIMMD.V" (andor): i2 = 0;
                EVALUATE $monitor and SCHEDULE for end of time
L3 "EX_SIMMD.V" (andor): EVALUATE GATE a1 >>> 1'b0
                 SCHEDULE Value Change on o1 for time=52
L9   EVALUATE $monitor (and display values)
Sim Time=             50 i1=1 i2=0 i3=x i4=x o1=x o2=x o3=x
Finished Activity at time 50; Look for next time of activity; Advance
      time to 52

SIMULATION TIME IS 52
L12 "EX_SIMMD.V" (andor): #25>>>EVALUATE and SUSPEND
      [SCHEDULE] beh_block [for 25 time units]
L3 "EX_SIMMD.V" (andor): UPDATE Value Change WIRE o1 >>>
      NET = 1'h0, 0
Sim Time=             52 i1=1 i2=0 i3=x i4=x o1=0 o2=x o3=x
Finished Activity at time 52; Look for next time of activity; Advance
      time to 75

SIMULATION TIME IS 75
L12 "EX_SIMMD.V" (andor): #25 >>> CONTINUE [EVALUATE
      beh_block]
L12 "EX_SIMMD.V" (andor): i3 = 1;
Sim Time=             75 i1=1 i2=0 i3=1 i4=x o1=0 o2=x o3=x
L13 "EX_SIMMD.V" (andor): #25 >>>EVALUATE and SUSPEND
      [SCHEDULE] beh_block[for 25 time units]
Finished Activity at time 75; Look for next time of activity; Advance
      time to 100

SIMULATION TIME IS 100
L13 "EX_SIMMD.V" (andor): #25 >>> CONTINUE[EVALUATE
      beh_block]
L13 "EX_SIMMD.V" (andor): i4 = 1;
L3 "EX_SIMMD.V" (andor): EVALUATE GATE a2 >>> 1'b1

Sim Time=             100 i1=1 i2=0 i3=1 i4=1 o1=0 o2=x o3=x

Finished Activity at time 100; Look for next time of activity; Advance
      time to 102

SIMULATION TIME IS 102

L14 "EX_SMMD.V" (andor): #100 >>>EVALUATE and SUSPEND
      [SCHEDULE] beh_block [for 100 time units]
L3 "EX_SIMMD.V" (andor): UPDATE Value Change WIRE o2 >>>
```

```
      NET=1'h1, 1
L4 "EX_SIMMD.V" (andor): EVALUATE GATE r3 >>> 1'b1
                  SCHEDULE Value Change on o3 for time=102
L4 "EX_SIMMD.V" (andor): UPDATE Value Change WIRE o1 WIRE o3
      >>NET=1'h1, 1
Sim Time=            102 i1=1 i2=0 i3=1 i4=1 o1=0 o2=1 o3=1
Finished Activity at time 102; Look for next time of activity; Advance
      time to 200

SIMULATION TIME IS 200
L14 "EX_SIMMD.V" (andor): #100 >>> CONTINUE[EVALUATE
      beh_block]
L14 "EX_SIMMD.V" (andor): $finish
Exiting Verilog Simulator at time 200

Normal exit
Thank you for using the Digital Design with Verilog Hardware Description Language
```

Example 4-3. Ideal simulation log for a sample circuit with tracing.

4.6 Analysis and Concepts in Event-Driven Simulation in Verilog

In the ideal simulation log, we see that the whole simulation is a sequence of Evaluate, Schedule, and Update Activities. Simulation starts at time 0 when, the model performs evaluations of all initial and always blocks. The evaluation of a block continues this until it suspends (or schedules itself) for a later time. During the course of this evaluations, value changes on reg variables are effected and their fanouts are evaluated or are scheduled for evaluations. For example, change in i1 in the beh_block due to assignment statement evaluation results in valuation of a gate a1. This may or may not result in further value changes to be scheduled.

4.7 Internal Data Structure Representation

Internally, during simulation, a Verilog HDL model is represented by a set of signals, a set of evaluation blocks and their inter-connections. This can be called the network data structure. In the above diagram, the rectangles represent the evaluation blocks and the ellipses represent the signals. The names of these are given inside and these correspond to the names in the design. Coupled with this is the event queue data structure that now enables us to see throughout this event driven simulation cycle in Verilog HDL.

4.8 Update and Evaluate Events

As seen before, the activity in Verilog Simulation consists of Evaluate Blocks and Update Values. This is reflected in two types of events in the event queue (also known as scheduler)—Update Event and Evaluate Events. In a model known as 2 pass model, the types of events are held in separate lists and are processed in two passes—the first pass for update and second for evaluate.

Network data structure for the module andor

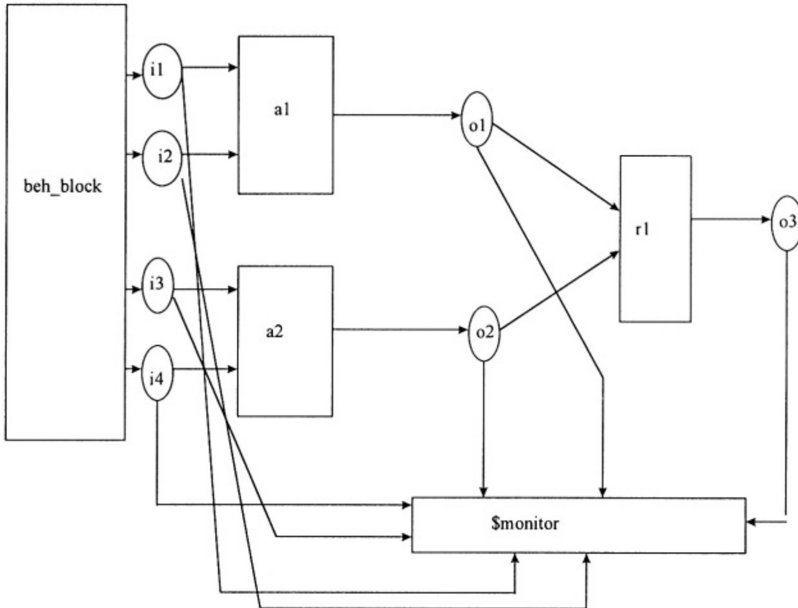


Figure 4-2. Network Data Structure for the andor Verilog Model in Example 4-1

4.9 Order of Execution of Events in Verilog

For parallel blocks, no order is defined. Each initial and always block, continuous assignments, structural gates, udp instances, and transistors form parallel blocks in Verilog. A simulator can choose an order in any manner for all events that are scheduled to happen at the same time. [Special events like $monitor are processed as the last set of events or a given time and are exception to the above.]

Sometimes a Verilog description can produce different results based on which block or process is executed first. This happens when multiple processes update a reg at the same time with no interdependence. This results in non-determinism OR different results on different simulators or in different simulation runs on the same simulator. This is a feature of Verilog HDL unlike VHDL where the results are always deterministic. A simple example of non-determinism is as follows:

```
initial
    #5
    rega= 1;
```

```
initial
      #5
      rega = 0;

initial
      #6 $display("rega at time = %d is %d", $time, rega);
```

/*
The above may result in 1 or 0 depending on the simulation run and the simulator tool */.

4.10 Algorithm for Event-Driven Model for Verilog HDL

4.10.1 Definitions

Signal – A declared data type in Verilog that can hold a value.

This can be a driver on a net (of any type like tri, triand etc.) or a net or a reg, an integer, a real or time type.

Process or Evaluation Block – A basic unit of evaluation in Verilog. This can be a behavioral block or its sub-part, a structural unit like gate or rtl unit like 'assign' statement.

Verilog objects – Signals and Evaluation Blocks form basic Verilog objects which form units of activity in the Verilog model.

Event – The atomic unit of activity in Verilog HDL semantic model. An event has the following three attributes associated with it: time, type, and an object. The activity that defines the event can be of the following two types (making the event type).

Update Event – The activity indicating a value change on a signal.

Evaluation Event – The activity indicating evaluation of a block (process).

Schedule – List of all events in the Verilog model at any time.

Network Representation of Verilog – A network consisting of Evaluation Blocks, Signals and their interconnections is a mapping from a given Verilog description.

More on the effects of evaluation events.

Evaluation a basic block results in one or more of the following:

1. update values of non-net types
2. schedule an update event on a net or a non-net type
3. schedule an evaluation event
4. change the interconnections in the network

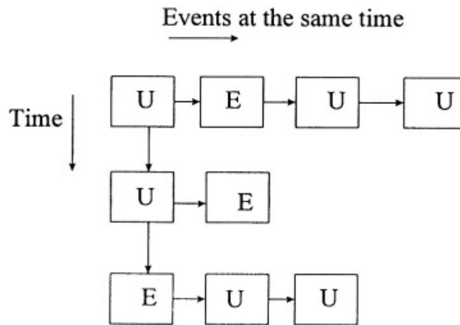


Figure 4-3. Schedule of Events for a Verilog Model

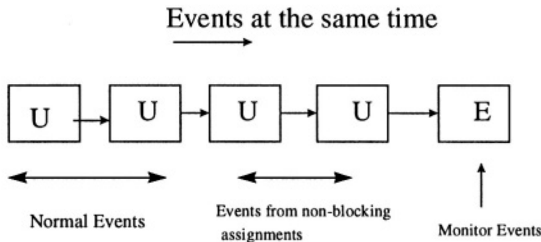
4.10.2 Algorithm

```
Start:
{
      Create a network representation of Verilog description. Set current time to 0. Schedule
evaluation events on all behavioral blocks for current time and update events for all UDP
outputs with initial state.
}
while ((events in schedule( and (no $finish))
{
      if (no events in schedule at current time)
            advance time to next event time in the schedule
      while (events at current time)
      {
            fetch next event in the schedule
            process event (either evaluation or update)
            remove this event from schedule
      }
}
```

Figure 4-4. Algorithm for Verilog Model Execution

```
process_events:
    {
        if (event is of type update)
        {
            update value of an object
            while (fanout blocks of event object signal)
            {
                Schedule Evaluation Event with block as object.
                If block is behavioral also remove it from fanout list.
            }
        }
        else if (event is of type evaluate)
        {
            if (event object evaluation block is structural or RTL)
                evaluate and schedule an update event on output net or net-driver
            else
                start processing statements in behavioral block
        }
    }
```

Figure 4-5. Algorithm for Processing an Event



Order of Events in the Schedule for a Verilog Model
E - Evaluate Event; U - Update Event

U
time=100
signal=o1

Inside an event

Figure 4-6. Order of Events at a Time and Event Structure Diagrams

```
schedule_event:
    {
        if (event type is update)
        {
            if (event object changed due to behavioral non-blocking statement)
                insert event in the schedule at end of the list at the appropriate
                time
```

```
        else
            If an event already exists on this signal then
                if the value on the existing event same as new event,
                    if yes, check the time on the existing event.
                            if earlier than new event then done;
                    else
                        reschedule for the new time, then done
                else
                    deschedule the existing event and insert the new event
            else
                    insert event at the appropriate time in the schedule.
    }
    else
    if (evaluate block is of type monitor)
    {
        insert event at the end of the current simulation time.
    }
}
```

Figure 4-7. Algorithm for Scheduling an Event

Note: Events from non-blocking assignments and $monitor statements must be maintained at the end of the lists at any time in the schedule during all scheduling.

4.11 Highlights of the Algorithm – Concurrent Processes and Event Cancellations

4.11.1 Concurrent Processes

Hardware consists of different design units that always run concurrently. For example, in a computer, CPU, memory, peripheral boards like the IO controllers (floppy disk controller), hard disk controller are always running in parallel. The states of some units may imply waits but they will be providing certain outputs as a function of inputs and current state, independent of other units. A hardware description language must model this behavior correctly. In the above algorithm, we can clearly see that all the always blocks and the initial blocks in the behavioral description of the Verilog model are executed concurrently. Similarly, the gates, module instances, UDPs and the RTL assignments are continuously providing the outputs as a function of inputs. In an event-driven model, this translates into creating an evaluation event on the evaluation blocks whenever one of the inputs changes. This again can be seen in the above algorithm.

In terms of the network-data structure, all the rectangles representing the evaluation blocks are executing concurrently. Events may be present on one or all of them at the same time within the schedule. The following two examples illustrate the inertial delay model followed in general in Verilog and described in the algorithm above.

```
module m;
    wire out;
    reg in1, in2;

    assign #5 out = in1 | in2;
    initial
    begin
        $monitor("Time = %d out = %d in1=%d in2=%d", $time, out,
            in1, in2);
            $dumpfile("ex4_2.dmp");
        $dumpvars;
        //    $gr_waves(out, in1. in2);
        #1
            in1= 1;
            #2
            in2= 1;
            #10
            $finish;
    end
endmodule
```

Example 4-4. ***Multiple events on a reg – but no cancellation (algorithm 4-7 applied)***

Running this produces the following results:

```
C1>.
Time =          0 out = x in1=x in2=x
Time =          1 out = x in1=1 in2=x
Time =          3 out = x in1=1 in2=1
Time =          6 out = 1 in1=1 in2=1
```

When multiple changes happen on inputs before output actually changes, event cancellation mechanism comes into play.

Thus, the event created at time for out to change to 1 does not get descheduled at time 3 as the new value of out is consistent with the scheduled value. However, in the following example, for the and gate, new value is different and the event is canceled and replaced with a new event with a different output value and different time of change.

4.11.2 Event Cancellations

```
module m;
    wire out;
    reg in1, in2;

    assign #5 out=in1 & in2;
    initial
```

```
begin
        $monitor("Time = %d out = %d in1=%d in2=%d", $time, out, in1,
        in2);
        $dumpfile("ex4_2.dmp");
        $dumpvars;
        //          $gr_waves(out, in1, in2);
        #1
        in1 = 1;
        #2
        in2 = 1;
        #2
        in1 = 0;
        #10
        $finish;
    end
endmodule
```

Example 4-5. Multiple events on a reg resulting cancellation (algorithm 4-7 applied)

In this example, at time 3 units, an event gets created on out to change at 8 time units. This event gets canceled at time 5 units when in1 changes to 0. A new event for out to change at 10 is created to change to 0. Here the new value of out computed at time = 5 units is different from the scheduled value and thus, we see event cancellation. This is generally known as inertial delay model. The transport delay model is supported in Verilog by use of non-blocking assignments as explained in the next chapter.

```
C1>.
Time =          0 out = x in1=x in2=x
Time =          1 out = x in1=1 in2=x
Time =          3 out = x in1=1 in2=1
Time =          5 out = x in1=0 in2=1
Time =         10 out = 0 in1=0 in2=1
Exiting VeriWell for Win32 at time 15
```

In the above algorithm, in the processing of events (Figure 4-5), events may or may not get descheduled and rescheduled when an event already exists on the signal. This depends on the new value and the scheduled value. In Verilog, typically only one event can be present on a net or a reg at a time. Exceptions to these are events arising out of non-blocking assignments.

4.12 Exercises

1. Obtain the simulation log for the following Verilog module using 'step and trace' OR with 'trace on' option. Modify this log using the semantic model to obtain the ideal simulation log in terms of 'EVALUATE EVENTS, UPDATE EVENTS, SCHEDULE and STATEMENT EXECUTION' primitive steps as in Examples 4-2 and 4-3 in the book.

```
module mixed_sim;
// This is a part of bigger circuit that drives the bus
    // This is modeled with mixed structure, rtl and behavior
    reg dcontrol, x; y;
    reg [31:0] data;          wire [31:0] bus;
    assign bus = dcontrol ? data : 'bz;
    always @x
        y = ~x;

    initial
        begin
            $monitor("time=%d dcontrol=%d data=%d x=%d y=%d bus=%d",
                    $time, dcontrol, data, x, y, bus);
            dcontrol = 0;
            #10  data = 15;
            #10  x = 0;
            #10  dcontrol = y;
            #10;
        end
endmodule
```

5 BEHAVIORAL MODELING

5.1 Overview of Behavioral Modeling

5.1.1 Introduction

This is the next level of abstraction after RTL. This allows modeling algorithmic style descriptions. Naturally, we need the ability to capture the algorithms like in a programming language. This is provided with "C"-like statements—assignments, if-else, case, for (loop), begin-end blocks, functions and procedures (tasks). In an HDL, as seen in the structural and RTL descriptions, the flow of control is multiple. Verilog is essentially a Concurrent Programming Language. This implies that we need a mechanism to synchronize the algorithmic descriptions together with the structural and RTL descriptions. This is provided by 'initial' and 'always' blocks that enclose all algorithmic descriptions. Within each of these blocks, timing controls or synchronization primitives are provided in terms of delays, event controls, fork-join statements, and wait statements.

The algorithmic descriptions are also called sequential blocks and the statements are termed sequential statements. This does not imply that these are sequential designs—the term originates from 'in sequence' or 'in order'. No storage is implied in these descriptions. The top level blocks are the initial and the always blocks in the behavioral descriptions. The tasks and function declarations also occur at the same top level in a module. Inside each of these, there are procedural assignments, if-else, case, loops, begin-end blocks, fork-joins, functions and task calls, and few other assignments ‘assign’, ‘deassign’, ‘force’ and ‘release’.

5.1.2 Examples

```
module clock(clock);
    output clock;
    initial
        clock = 0;
    always
        #100 clock = ~clock;
endmodule
```

Example 5-1. Behavioral clock generation.

In the Example 5-1, clock is generated behaviorally. One initial and one always block is used.

5.1.3 Syntax

As seen in section 3.2.3, behavioral descriptions are of four types—initial, always, tasks, or functions. The tasks and functions are explained in section 5.10. The initial and always are expanded as follows:

```
initial_construct
    ::= initial statement

always_ construct
    ::= always statement

statement_or_null
    ::= statement
    | ;

statement
    ::=blocking assignment;
    | non-blocking assignment;
    | procedural_continuous_assignment
    | procedural_timing_control_assignment
    | conditional_statement
    | case_statement
    | loop_statement
    | wait_statement
    | disable_statement
    | event_trigger
    | seq_block
    | par_block
    | task_enable
    | system_task_enable
```

Each of these above will be explained in the sections in this chapter.

5.2 Procedural Assignments

5.2.1 Overview

These allow transfer of values from an expression into a non-net 1value. All algorithmic descriptions can be thought of as temporary computations whose results can then be put onto physical entities like nets via RTL or gate descriptions that are closer to physical reality. Thus, the original intent of behavioral level HDL is to allow expressing your ideas into a precise form whose results can be easily transferred to the RTL and structural level for ease of development. This will allow stepwise refinement of your model from a higher level description into gates. All non-net constructs (reg, integers, reals, time and any aggregate of these) are abstract and there are different ways to actualize these into hardware. Synthesis has some role to play here, but in reality, a very small subset of the behavioral level is synthesized by logic synthesis and some larger subset by behavioral synthesis.

The procedural assignments are of blocking and non-blocking kind and they are overridden by quasi-continuous assignments 'assign' and 'deassign' which in turn can be overridden by 'force' and 'release' statement.

5.2.2 Blocking (Immediate) Procedural Assignments

5.2.2.1 Examples

```
// This is an example of blocking assignments
      initial
      begin
            rega = 2;
            regb = 3;
            #5;
            rega = 0;
            regb= 1;
            regc = regb+rega;
            #5;
      $display(regc);
end
```

Example 5-2. Blocking assignments – inter-assignment delays

In this example, the assignment to regc happens after assignments to rega and regb are complete at time 5 units. The previous assignments to rega and regb at time 0 are overridden with the assignments at time 5 before the right-hand side expression for assignment to c is computed. This implies that regc gets a value of 1 and is displayed as such at time of 10 units. This is in contrast with the Example 5-4 where the non-blocking assignments take place in a deferred manner. The delays of 5 time units ocurring twice in this example are between the assignments and block the flow of control of this behavioral block (or process) for 5 time units each. This is in

contrast to intra-assignment delays explained later. There is no scheduling activity involved in the blocking assignments.

```
// This is another example of blocking assignment
// Compare with similar looking example for blocking
// assignments in previous section.
// This shows delays are not blocking the flow of control
    initial
    begin
        rega=#5 2;
        $display($time, rega, regb);
        regb=#5 3;
        $display($time, rega,regb);
        #6
        $display($time, rega,regb);
        #5
        $display($time, rega,regb);
    end

    /* This will result in
    5 2 x
    10 2 3
    16 2 3
    21 2 3
    */
```

Example 5-3. Blocking Assignment – intra-assignment delays.

In the above example, the delays are present inside the assignment statement which implies that they are intra-assignment delays as opposed to the inter-assignment delays of Example 5-2.

5.2.2.2 Syntax

```
blocking assignment
    ::= reg_1value = [delay_or_event_control] expression ;
```

5.2.3 Non-Blocking Procedural Assignments

5.2.3.1 Introduction – When modeling states in a sequential block, the IOs of a block may be used both on the right- and left-hand side of assignments. The right-hand side would then be from previous state while the left-hand side will get the new value in this state. This kind of modeling is not possible with blocking assignments and the concept of non-blocking assignment is introduced.

To allow modeling whereby a state of the system is captured at a delta cycle time and then all values are determinable by the delta cycle of the simulator, notion of non-blocking procedural assignments was introduced in later versions of Verilog. Thus, use blocking assignments for temporary computations within a delta cycle for

those regs that are not directly transformed to hardware registers. This (non-blocking) type of assignment is a better model for most real registers when modeling synchronous systems. However, by following certain consistent conventions throughout Verilog model, one can use the blocking assignments which are more efficient for simulation and which are easier to understand.

5.2.3.2 Examples

```
// This is an example of non-blocking assignments
// Compare with similar looking example for blocking
// assignments in previous section
initial
begin
    rega <= 2;
    regb <= 3;
    #5;
    rega <= 0;
    regb<= 1;
    regc <= regb+rega;
    #5;
    $display(regc);
end
```

Example 5-4. ***Blocking assignments – interassignment delays.***

In this example, the assignment to regc happens with the right-hand side computed before new assignments to rega and regb are complete at time 5 units. The previous assignments to rega and regb at time 0 are used for computing assignment to c at time 5. This implies that regc gets a value of 1 and is displayed as such at time 10 units. This is in contrast with the previous Example 5-2 where the blocking assignments take place immediately and regc gets a value of 5.

```
// This is an example of non-blocking assignments
// Compare with similar looking example for blocking
// assignments in previous section. This shows
// delays are not blocking the flow of control
initial
begin
    rega<=#5 2;
    $display($time, rega, regb);
    regb<=#5 3;
    $display($time, rega,regb);
    #6
    $display($time, rega,regb);
    #5
    $display($time, rega,regb);
end

/* This will result in displaying values
    0 x x
```

```
        0 x x
        6 2 3
        11 2 3
    */
```

Example 5-5. Blocking assignments – intra-assignment delays.

5.2.3.3 Syntax

non-blocking assignment
  ::= 1value = expression
  | 1value = delay_or_event_control expression ;

delay_or_event_control
  ::= delay _control
  | event_control
  | **repeat** (expression) event_control

5.2.4 Examples Comparing Blocking and Non-Blocking Assignments

5.2.4.1 Example 1

```
module evaluates3(out); //The simulator evaluates
    output out;       //     right-hand side of the
    reg a, b, c;      //  non-blocking assignments and
                      // schedules the assignments of
    initial
    begin             // the new values at posedge c.
        a = 0;
        b = 1;            // Step 2 : At posedge c,the simulator
        c = 0;            // updates the left-hand side of
    end                   // each non-blocking assignment
                      // statement.
    always c = # 5 ~c;

    always @ ( posedge c )
    begin: exch_block
        a <= b; // evaluates, schedules
        b <= a; // and executes in two steps
    end

    //simulation control section
    initial
    begin
        $monitor("$time=%d a=%d b=%d c=%d", $time, a, b, c);
        #30 $finish;
    end
endmodule
```

This displays the following results:

```
$time=        0 a=0 b=1 c=0
$time=        5 a=1 b=0 c=1
$time=       10 a=1 b=0 c=0
$time=       15 a=0 b=1 c=1
$time=       20 a=0 b=1 c=0
$time=       25 a=1 b=0 c=1
```

Example 5-6. Blocking and non-blocking comparison – exchange of values for blocking.

5.2.4.2 Example 2 – On replacing the non-blocking assignments in Example 1 above by blocking assignments, we obtain the following code for exch_block.

```
always @ ( posedge c )
begin: exch_block
      a = b; // evaluates, schedules
      b = a; // and executes in two steps
end
```

On running the module evaluate3 with this change, following result is obtained:

```
$time=        0 a=0 b=1 c=0
$time=        5 a=1 b=1 c=1
$time=       10 a=1 b=1 c=0
$time=       15 a=1 b=1 c=l
$time=       20 a=1 b=1 c=0
$time=       25 a=1 b=1 c=1
```

Example 5-7. Blocking and non-blocking comparison – no exchange of values for blocking.

5.2.4.3 Example 3 – This example illustrates a special feature of non-blocking assignments—ability to do multiple scheduling.

```
module multiple;
      reg r1;
      reg [ 2 : 0 ] i;

      initial begin
            $monitor("time=%d, i=%d r1 = %d", $time, i, r1);
            // starts at time 0, doesn't hold the block
            r1=0;
            // make assignments to r1 without canceling previous assignments
            for ( i = 0; i <= 5; i = i+1 )
            r1<=#(i*10)i[0];
```

```
    end
endmodule
```

This produces following output:

```
time=     0, i=6 r1 = 0
time=    10, i=6 r1 = 1
time=    20, i=6 r1 = 0
time=    30, i=6 r1 = 1
time=    40, i=6 r1 = 0
time=    50, i=6 r1 = 1
```

Example 5-8. Non-Blocking assignments – support of multiple schedules.

5.2.4.4 Example 4 - Here we take the Example 3 above and replace the non-blocking with a blocking assignment.

```
module multiple;
    reg r1;
    reg [2:0]i;

    initial begin
        $monitor("time=%d, i=%d r1 = %d", $time, i, r1);
        // starts at time 0, doesn't hold the block
        r1= 0;
        // make assignments to r1 without canceling previous assignments
        for ( i = 0; i <= 5; i = i+1 )
            r1 = # (i*10) i[0];
    end
endmodule
```

Running this produces the following output:

```
time=     0, i=1 r1 = 0
time=    10, i=2 r1 = 1
time=    30, i=3 r1 = 0
time=    60, i=4 r1 = 1
time=   100, i=5r1= 0
time=   150, i=6r1 = 1
```

Example 5-9. Blocking assignments – no multiple schedules.

5.3 Conditional Statement

5.3.1 Overview

This statement allows execution of code based on evaluation of a condition as in the following form:

```
if (condition)
    statement;
else
    statement;
```

5.3.2 Examples

```
if   (index >0)
    if (rega > regb)
        result = rega;
    else
        result = regb;
```

Example 5-10. If statement example.

```
if(index > 0)
begin
    if (rega > regb)
        result = rega;
end
else
        result = regb;
```

Example 5-11. Nested if statement.

5.3.3 Syntax

conditional_statement ::=
 if (expression) statement_or_null
 | **if** (expression) statement_or_null **else** statement_or_null

5.3.4 Special Considerations

Nesting rules apply to if statements as if statements can occur within other if statements and along with else statements. In general, if statement is just like any other statement and can be present in the statement part in the above syntax. Multiple mutually exclusive conditions can be modeled using 'if' immediately after else at the same level. Only a difference in indentation adds clarity to such descriptions.

```
if (c1)
      a1;
else
if (c2)
      a2;
else
if (c3)
```

Example 5-12. If-else-if mutually exclusive multiple conditions.

5.4 Case Statement

5.4.1 Overview

When a condition or a value of a value of a variable is checked against several alternatives and different actions taken, then case statement is the construct to use. Comparisons go from top to bottom (first to last). Unlike "C", overlap of case-values is allowed and the order of evaluation is important.

5.4.2 Examples

```
module m(a,b,c,d, select, mux);
input a, b,c,d;
input [1:0] select;
output mux;
reg mux;

always@({a,b,c,d,select})
 case (select)
        2'b00:    mux = a;
        2'b01:    mux = b;
        2'b10:    mux = c;
        2'b11:    mux = d;
        default:  mux = 'bx;
 endcase
endmodule
```

Example 5-13. Case statement – a multiplexor model.

5.4.3 Syntax

case_statement::=
    case (expression) case_item {case_item} **endcase**
    | **casez** (expression) case_item {case_item} **endcase**
    | **casex** (expression) case_item {case_item } **endcase**

5.4.4 Don't Cares and Case Statements – Casex and Casez

Special statements which use keywords casex and casez are provided in Verilog for special treatment to comparisons of x and z values. After evaluation of case operand, in comparison to the case-values, x/z on either side is a don't care (true) condition. Usage of these constructs is made in logic synthesis for optimizations and these have been added to Verilog HDL as enhancements to facilitate the development of popular synthesis tools. The more don't care conditions one can specify to a synthesizing compiler, the better job synthesizer is going to do as known by the optimization techniques.

5.4.5 Examples Comparing Case, Casex, and Casez

5.4.5.1 Case Example

```
module m;
     reg sig;
     always @(sig)
     case (sig)
               1'bx :  $display ("signal is unknown");
               1'bz:  $display ("signal is  floating");
               default: $display ("signal is %b", sig);
     endcase
     initial
     begin
          $monitor("time=%d sig=%b",$time, sig);
          #10 sig = 0,
          #10  sig = 1;
          #10  sig = 'bz;
          #10  sig = 'bx;
     end
endmodule
```

The above will result in :

```
time=              0 sig=x
signal is  0
time=             10 sig=0
signal is  1
time=             20 sig=1
signal is floating
time=             30 sig=z
signal is unknown
time=             40 sig=x
```

Example 5-14. Case statement with unknowns and tri-states.

5.4.5.2 Case Example

```
module m;
    reg sig;
    always @(sig)
    casez (sig)
            1'bx:  $display ("signal is unknown") ;
            1'bz:  $display ("signal is  floating") ;
            default: $display ("signal is %b", sig) ;
    endcase
    initial
    begin
        $monitor("time=%d  sig=%b",$time, sig);
        #10 sig = 0;
        #10 sig = 1;
        #10 sig = 'bz;
        #10 sig = 'bx;
    end
endmodule
```

This will result in :

```
time=                  0 sig=x
signal is floating
time=                 10 sig=0
signal is floating
time=                 20 sig=1
signal is unknown
time=                 30 sig=z
signal is unknown
time=                 40 sig=x
```

Example 5-15. Casex statement with unknowns and tri-states.

5.4.5.3 Case Example

```
module m;
    reg sig;
    always @(sig)
    casex (sig)
            1'bx :  $display ("signal is unknown") ;
            1 'bz :  $display ("signal is  floating") ;
            default:  $display ("signal is   %b", sig) ;
    endcase
    initial
    begin
        $monitor("time=%d sig=%b",$time, sig);
        #10 sig = 0;
        #10 sig = 1;
        #10 sig = 'bz;
```

```
            #10 sig = 'bx;
        end
    endmodule
```

This will result in:

```
    time=                    0 sig=x
    signal is unknown
    time=                   10 sig=0
    signal is unknown
    time=                   20 sig=1
    signal is unknown
    time=                   30 sig=z
    signal is unknown
    time=                   40 sig=x
```

Example 5-16. Case statement with unknowns and tri-states.

5.5 Loops

5.5.1 Overview

Looping constructs allow modeling of repetitive behavior. There are four kinds of looping constructs in Verilog: **for, while, forever and repeat.** The for, repeat and while loops are similar to those in programming language but forever has notion of time. The for statement provides for initial value of loop variable, an increment on each loop and an exit condition. The repeat statement specifies the repeat count and the while statement loops on a condition specified after keyword while. The forever statement is like always, but can also be used within block. An initial immediately followed by forever is identical to always. Fork-join blocks may have a forever but not always inside them.

5.5.2 Examples

FOR STATEMENT EXAMPLE

```
        Val = 5;
        for (fibNum = 0; Val != 0; Val = Val -1)
            fibNum = fibNum + Val;
```

Example 5-17. Loops – for statement usage.

WHILE STATEMENT EXAMPLE

```
        Val = 5;
        while (Val != 0)
        begin
            fibNum = fibNum + Val;
            Val= Val-1;
        end
```

Example 5-18. Loops – while statement usage.

```
REPEAT STATEMENT EXAMPLE
    repeat(5)
    begin
        fibNum =fibNum+Val;
        Val = Val - 1;
    end
```

Example 5-19. Loops – repeat statement usage.

```
FOREVER STATEMENT EXAMPLE
        initial
        // initialization here

                forever
                begin
                    if (reset)
                        reset_actions;
                    else
                        fetch_and_execute_instructions;
            end
```

Example 5-20. Loops – forever statement usage.

5.5.3 Syntax

loop_statement: :=
forever statement
| **repeat** (expression) statement
| **while** (expression) statement
| **for** (assignment; expression ; assignment) statement

5.6 Begin-End Blocks

5.6.1 Introduction

This is Pascal-like way of grouping procedural statements together. In the syntax, begin-end forms a single statement and thus, can occur wherever a single statement can occur. They may have names in which case these are called named blocks. Certain declarations can be locally made within named blocks and named blocks can be disabled.

5.6.2 Example

```
while (Val != 0)
begin: fblock
      fibNum=fibNum+Val;
      Val = Val - 1;
end
```

Example 5-21. Sequential blocks – begin-end usage.

5.6.3 Syntax

seq_block
  ::= begin{ statement} **end**
  | **begin** : block_identifier{ block_item_declaration}{ statement} **end**

5.7 Wait Statements

5.7.1 Introduction

These provide a way to synchronize two or more threads of control (each initial or always block provide a thread of control; these may bifurcate into multiple threads if fork-joins are used). The Example 5-22 below is an example of simple handshaking between two blocks.

5.7.2 Example

```
        initial          initial
        begin            begin
        b=0;             a=0;
        wait (a=1);      #10;
        b = 1;           a=1;
        wait (b=1);        ...
    end
    end
```

Example 5-22. Wait statement – synchronizing two processes.

5.7.3 Syntax

wait_statement::=
  wait (expression) statement_or_null

5.8 Event and Delay Controls

5.8.1 Overview

5.8.1.1 Introduction – Event control is probably the most elegant feature in Verilog for synchronization and timing abilities. Several event related facilities are discussed in the following sections of:

1. event declarations
2. multi-event event – event OR
3. event usage – @event
4. event generalization
5. generalized event transitions

5.8.1.2 Examples

```
event ec1, ec2; //from 8085 based system - 2 phases of cycle
->ec1 ; //trigger ec1 event
always @ ec1 //event usage
    check_for_interrupts;
```

Example 5-23. Event declarations and usage.

5.8.1.3 Syntax

```
event_statement::= event_declaration
        ||= multi_event_event
        ||= event_trigger_statement
        ||= scalar_event_expression
        ||= event_control event_expression
```

5.8.2 Event Declarations

5.8.2.1 Introduction – Special flags or events that can be triggered and can be waited upon are declared using event declarations. Examples and usage is discussed in the subsequent sections in this chapter.

5.8.2.2 Example

```
event e1, e2;
```

Example 5-24. Event declarations.

5.8.2.3 Syntax

```
event_declaration
    ::= event { name_of_event, } ;
```

5.8.3 Multi-Event Event – Event OR

5.8.3.1 Introduction – The only operator supported on the event data-type is the OR operator. This allows multiple events to trigger an action when any one of these is triggered.

5.8.3.2 Example

```
(e1 or e2)
```

Example 5-25. Multi-event event.

5.8.3.3 SYNTAX

```
multi_event_event ::= event_expression or event_expression
```

5.8.4 Event Usage

5.8.4.1 Introduction – Events can be triggerred by using the -> symbol. All triggered events immediately start the activity at the place where sensitivity has been created for that event using the @ symbol or using the wait statement. Triggering events is analogous to setting a condition to be satisfied and then the code waiting for this condition to be satisfied resumes execution on the event trigger. In the following example, the tasks action_set1 and action_set2 are waiting on the event e1 and e2, respectively. On triggering events e1 and e2, the 2 tasks will be executed. Typically the trigger and the action will be located in different threads of control. This can be seen in Example 5-26 below.

5.8.4.2 Example

```
->e1; ->e2;

always
@el
action_set1;
always
@e2
action_set2;
```

Example 5-26. Event triggering and event based synchronization.

Control flow modeling with events must have correct sensitivity and appropriate enclosing control structures like always, initial or forever.

5.8.4.3 Syntax

```
event_trigger ::=
        -> event_identifier ;
```

5.8.5 Event Generalization

5.8.5.1 Introduction –Any expression is allowed in place of an event variable. The semantics of this construct is as follows: any value change in this expression will result in an implicit trigger.

5.8.5.2 Example

```
    @(var1)
        action_set1;

// the following is not a good way to model
// semantics is not obvious for this
    @(var1==1)
        action_set1;
```

Here are equivalent, cleaner ways to model:

```
    For gating value of var1, use wait
    wait(var1 == 1);
    action_set1;
```

If a change is must alongwith gating value, use @ combined with wait.

```
    always
    begin
    @var1
    wait(var1 ==1);
    action_set1;
end
```

This is same as:

```
    always @var1
        if (var1==1)
        action_set1;
```

5.8.5.3 Syntax

```
scalar_event_expression ::=
    Scalar event expression is an expression that resolves to a one bit value.
```

5.8.6 Generalized Event Transitions

5.8.6.1 Introduction – For any expression that evaluates to a bit-value, event can be created for a value change on that expression using the '@' symbol followed by the

expression enclosed in parentheses. This forms a generalized event construct in Verilog and is a powerful tool for modeling synchronizations and sensitivities. In addition, rising and falling edge transitions via posedge (positive edge) and negedge (negative edge) operators is supported as generalized event transitions. In specify blocks, while modeling path-delays, any transition on a signal forms an event. Same is true of the user-defined primitives that are edge-sensitive.

5.8.6.2 Example

```
always  @reset
    if (reset ==1)
        reset_operation;
    else
        normal_operation;
```

5.8.6.3 Syntax

```
event_control
    ::= @ identifier
    | @ ( event_expression )

event_expression
    ::= expression
    | posedge scalar_event_expression
    | negedge scalar_event_expression
    | event_expression or event_expression
```

5.9 Fork-Join Blocks

5.9.1 Introduction

This is a way of providing concurrency within a procedural block. All statements within this block will execute concurrently.

5.9.2 Examples

```
begin
    fork
        forever
            begin: IP
                wait(reset= 1);
                fetch_instruction;
                process_instruction;
            end

        forever
            begin
                wait (reset ==0);
                disable IP;
                Reset_registers;
```

```
            wait (reset == 1);
        end
    join
```

Example 5-27. ***Fork-join statements – modeling asynchronous reset and instruction loop concurrency in microprocessors.***

5.9.3 Syntax

par_block
 ::=fork{ statement} **join**
 | **fork**: block_identifier{ block_item_declaration}{ statement}
 block_identifier

5.9.4 Special Considerations – Modeling Pipelines

These are useful for modeling instructions in a microprocessor whereby parallelism or pipelines begin after some sequential operations. For example, fetching of operands from memory is typically serial, but processing may be done in a parallel fashion. Resets and interrupts are also handled in parallel to the instruction execution.

```
    // All variables here are globally declared
    task add4;
    begin
        @posedge(clk)
            fetch(op1);
        @posedge(clk)
            fetch(op2);
        @posedge(clk)
            fetch(op3);
        @posedge(clk)
            fetch(op4);
    fork
                @posedge(clk)
            sum1 = op1 + op2;
        @posedge(clk)
            sum2 = op3 + op4;
    join
     @posedge(clk)
    sum = sum1 + sum2;
end
```

Example 5-28. ***Modeling instructions with no parallelism.***

```
// All variables here are globally declared
    task add4;
    begin
        @posedge(clk)
```

```
        fetch(op1);
    @posedge(clk)
        fetch(op2);
    @posedge(clk)
        fetch(op3);
    @posedge(clk)
        fetch(op4);
    @posedge(clk)
    fork
        sum1 = op1 +op2;
        sum2 = op3 + op4;
    join
    @posedge(clk)
            sum = sum1 + sum2;
end
```

Example 5-29. Modeling instructions with parallelism.

```
// Total number of clock cycles here is 6
// If sum1 and sum2 are done in sequence, this would be 7 cycles
```

The join waits for all statements within the fork-join to complete before moving on to next instruction after the join

```
// add_mult4
@posedge clk
fork
    sum1 = op1 + op2;
    @posedge clk
        mult1 = op3 * op4;
join
@posedge (clk)
    result = sum1 + mult1;
```

Example 5-30. Modeling instructions with pipeline – fork-join usage.

Here the control thread coming from sum1's computation will wait for an extra clock cycle as multiplication is taking 2 clock cycles. Sequential and parallel blocks can be embedded within each other to produce complex control structure, that models complex machines elegantly. (Possibly other control flow architectures.)

5.10 Functions and Tasks

5.10.1 Functions

5.10.1.1 Overview – Functions have been encountered in expressions in RTL descriptions. These can also be used in procedural assignments and other places in algorithmic descriptions. These are explained here, since their definitions are strictly

algorithmic style descriptions. Functions can contain zero delay statements only , returns a single value, must have at least one parameter and can not call a task.

As opposed to this, tasks can have any sequential statements including waits and event control, can be invoked from within any sequential block, do not return a value, and need not have a parameter.

5.10.1.2 Examples

```
function      mux;
input             a, b, c, d;
input         [1:0]   select;

    case (select)
        2'b00:      mux = a;
        2'b01:      mux = b;
        2'b10:      mux = c;
        2'b11:      mux = d;
        default:    mux = 'bx;
    endcase
endfunction
```

Example 5-31. Function definition – a mux.

```
module multiplexor(a, b, c, d, select, e);
    input             a, b, c, d;
    input         [1:0]   select;
    output            e;
        assign        e = mux (a, b, c, d, select);
endmodule
```

Example 5-32. Function call – creating a multiplexor module with function mux.

5.10.1.3 Syntax

```
function_declaration
    ::= function [range_or_type] function_identifier ;
        function_item_declaration
        statement
endfunction

function_item_declaration
    ::= block_item_declaration
    |   input_declaration

range_or_type
    ::= range
    | integer
    | function_identifier
    ::= IDENTIFIER
```

```
block_item_declaration
        ::= parameter_declaration
reg_declaration
        | time_declaration
        | integer_declaration
        | real_declaration
        | realtime_declaration
        | event_ declaration
```

5.10.2 Tasks

5.10.2.1 Overview – Tasks are procedural blocks that can be called from within any sequential statement or block. As opposed to functions, tasks can have any sequential statements including waits and event control, can be invoked from within any sequential block, do not return a value, and need not have a parameter. The parameter passing proceeds by copying in at invocation time and copying back at return time.

5.10.2.2 Examples

Task Definition Example

```
module waveShReg;
    wire        shiftout;    //net to receive circuit output value
    reg         shiftin;        //register to drive value into circuit
    reg     phase1,phase2;    //clock driving values
    parameter       d = 100;        //define the waveform time step
    shreg       cct (shiftout, shiftin, phase1, phase2);
    initial
        begin :main
            shiftin = 0;                /initialize waveform input stimulus
            phase1 = 0;
            phase2 = 0;
            setmon;             // setup the monitoring information
            repeat(2)                   //shift data in
                clockcct;
    end

task setmon;                    //display header and setup monitoring
    begin
        $display("  time  clks  in out wal-3 wbl-2");
        $monitor ($time,,,,phasel, phase2,,,,,,shiftin,,,, shiftout,,,,,
            cct.wa1, cct.wa2, cct.wa3,,,,,cct.wb1, cct.wb2);
    end
endtask
endmodule
```

Example 5-33. Task declaration and usage – a shift register.

5.10.2.3 Syntax

```
task_declaration
        ::= task task_identifier;
                {task_item_declaration}
                statement_or_null
        endtask

task_item_declaration
        ::= block_item_declaration
        |   input_declartion
        |   output_declaration
        |   inout_declaration
```

Syntax for Task Enabling

```
task_enable
        ::= task_identifier
        | task_identifier (expression{ ,expression} ) ;

system_task_enable
        ::=  system_task_name;
        | system_task_name( expression{ ,expression} ) ;
system_task_name
        ::= $system_identifier (Note: the $ may not be followed by a space.)
```

5.10.2.4 Special Considerations in Tasks – Verilog mimics hardware and does not support pure recursion. The task recursion in Verilog is limited and no stack is built; only control thread duplication takes place but data structures get reused like in a pipelined design.

5.11 Task Disabling

5.11.1 Introduction

Task disabling allows stopping a thread of control. Disabling is also supported for named blocks. Reset and interrupt modeling in a microprocessor cannot be done without an asynchronous change in thread of control like in disable.

5.11.2 Examples

```
module microprocessor(reset,data,address);
input reset;
        inout data[63:0], address[31:0];

        initial
```

```
        fork
            forever
            begin: IP
                    wait(reset == 1);
                    fetch_instruction;
                    process_instruction;
            end
            forever
            begin
                    wait (reset == 0);
                    disable IP;
                    Reset_registers;
                    wait (reset == 1);
            end
        join

        task fetch_instructions;
        .......
        endtask;

        task process_instructions;
        ........
        endtask;

    endmodule
```

Example 5-34. Task disabling – reset modeling for a microprocessor.

5.11.3 Syntax

disable_statement ::= **disable** name_of_block_or_task;

5.12 Assign-Deassign Statements

5.12.1 Introduction

The assign and deassign statements provide a way of modeling connections and disconnections. This is like the tri-state bus or other switching networks or a latching phenomenon, which are some of the simplest examples of connections and disconnections. Assign creates a quasi-continuous assignment on the reg type variable on the left hand side of the assignment.

5.12.2 Example

```
module dFlop (preset, clear, q, clock, d);
    input  preset, clear, clock, d;
    output  q;
    reg  q;
```

```
    always
        @(clear or preset)
            begin
                if(!clear)
                    #10    assign q = 0;
                else if (!preset)
                    #10    assign q=1;
                else
                    #10    deassign q;
            end

    always
        @(negedge clock)
            #10    q = d;
endmodule
```

Example 5-35. ***Aassign – deassign – a flip-flop model with quasi-continuous assignments.***

5.12.3 Syntax

procedural_continuous_assignment::=
assign reg_assignment |
deassign reg_1value

5.13 Force-Release Statements

5.13.1 Introduction

The statements of force-release are provided as a debugging mechanism in Verilog. The 'force' statement allows you to put a value on a net and override all other drivers to it. 'release' will release this. Use of these in a model is not recommended. These are for use only in stimulus or test-bench or in interactive sessions. Use assign, deassign in a model. The reason for this is semantics (or lack thereof for this construct). 'assign' is also a way of modeling level-sensitive behavior that is prevalent upon specific input conditions. 'force' is a debugging tool in general and is useful especially for arbitrarily "assigned" lvalues.

5.13.2 Examples

```
module test_dff;
//Instantiate d flip-flop module defined in previous section
    dFlop dff(preset, clear, q, clock, d);

    initial
    begin
        // set initial state; This is the only way to set
        // a variable that has an "assign" on it
```

```
            force q = 0;
            preset = 1;
            #20;
            preset = 0;
            release q;
            #20;
            preset = 1;
        end
    endmodule
```

Example 5-36. Force-release statements – debugging the flip-flop model.

Force and release statements are designed for use in an interactive debugging session. When some variable goes to value x, to see if it is the source of problem and if the fanouts are also causing the problem, force this to non-x value and set the trace on. This will indicate all fanout evaluation results as they are evaluated with the inputs forced to known values. If any of the results are bad (x), the problem is now located. If not, release this node and continue onto other nodes with the same debugging process using force and release.

5.13.3 Syntax

force_statement::= **force** reg_assignment |
release reg_lvalue |
release net_lvalue

5.14 A Behavioral Modeling Example – An Essential Microprocessor

In this section, we describe a model of a microprocessor that entails the essence of behavioral modeling. Here we model a few instructions and the reset operation. Usage of case statements in the decoding operation, disable statements for resets, memory to store the source program and data, as well as methods of input and output are noteworthy in this example. The input and output files are also listed after the model.

```
// Write a model of a processor with the following characteristics:
// This is 8 bit system. Separate data and address bus of 8 bits exist.
// Has 5 instructions: ADD, SUB, MUL, DIV, BRANCH The instructions of
// ADD, SUB, MUL, DIV have 3 operands. BRANCH is relative.
// This has memory of 256 locations that contains the instruction stream.
// This has a reset sequence that is based on the wire RESET becoming '1'.
// The clock cycle is 50 ns.
// The ADD and SUB take 1 clock -cycle, BRANCH takes 2 clock cycles, MUL
// and DIV take 3 clock cycles.
// Assume any other characteristics as necessary as long as they do not

// conflict with those above.

/*****************************************************************
```

```
* Processor Assumptions/Description
*********************************************************************/
// This processor has a separate data and instruction memory. The instructions
// are contained in the file "InstrMemory.data". The data memory image is
// contained in the file "DataMemory.data".
// The instruction word length is 32-bits.
//
// The instruction format for the ADD, SUB, MUL and DIV instructions is:
//
//     31            23            15            7            0
//    -------------------------------------------------------------
//    |   OPCODE    |   DEST ADDR  |  SOURCE ADDR 1 | SOURCE ADDR 2 |
//    -------------------------------------------------------------
//
// The instruction format for the BRA instruction is:
//
//     31         23            15            7            0
//    -------------------------------------------------------------
//    |   OPCODE  |  BRA ADDR (REL)  |  NOT USED   |   NOT USED |
//    -------------------------------------------------------------


`timescale 1ns/1ns

module processor;

reg clock;
      reg reset;

      reg [31:0] Imem[0:255];       // 256 word x 32-bit instruction memory
      reg [7:0]  Dmem[0:255];       // 256 word x 8-bit data memory

      reg [7:0] pc;               // 8-bit program counter
      reg [31:0] instruction;       // 32-bit Instruction Register

      reg [7:0] opcode;             // 8-bit opcode
      reg [7:0] destAddr;           // 8-bit Destination Address
      reg [7:0] src1Addr;           // 8-bit Source Address #1
      reg [7:0] src2Addr;           // 8-bit Source Address #2

      reg [7:0] src1Data;           // 8-bit Source Data #1
      reg [7:0] src2Data;           // 8-bit Source Data #2
      reg [7:0] Result;            // 8-bit Result Data

      reg [7:0] braAddr;           // 8-bit Relative Branch address

      // define mnemonics to represent opcodes
        `define ADD 8'b00000000
        `define SUB 8'b00000001
        `define MUL 8'b00000010
```

```
       `define DIV 8'b00000011
       `define BRA 8'b00000100

       `define clk_period  50

/*****************************************************************
 *  Clock oscillator
 *****************************************************************/
      initial
#5  assign clock=1;
      always
#( clk_period/2)  assign clock=~clock;


/*****************************************************************
*  Processor Model
******************************************************************/
always  @(posedge reset or posedge clock) begin
      if  (reset)
      begin     // Reset Sequence
           disable fetch_and_execute;
           pc=0;
      end
else begin :  fetch_and_execute // Fetch and execute instructions

  instruction=Imem[pc];

  opcode=instruction[31:24];            // Get Opcode
  destAddr=instruction[23:16];           // Get Destination Data Address
  src1Addr=instruction[15:8];           // Get Source Data Address #1
  src2Addr=instruction[7:0];            // Get Source Data Address #2

  braAddr=instruction[23:16];           // Get relative address for BRA instruction

  case(opcode)
     `ADD:
     begin
           $display("ADD INSTRUCTION:");
           src1Data=Dmem[src1 Addr];       // Get Data Operands
           src2Data=Dmem[src2 Addr];
           Result=src1 Data+src2Data;        // Calculate Result
           Dmem[destAddr]=Result;         // Write Result data
           #1 pc=pc+1;                  // ADD instruction takes one clock cycle
     end

`SUB:
     begin
           $display("SUB INSTRUCTION:");
           src1Data=Dmem[src1Addr];        // Get Data Operands
           src2Data=Dmem[src2Addr];
```

```
            Result=src1Data-src2Data;      // Calculate Result
            Dmem[destAddr]=Result;         // Write Result data
            #1 pc=pc+1;                    // SUB instruction takes one clock cycle
      end

      `MUL:
      begin
            $display("MUL INSTRUCTION:");
            src1 Data=Dmem[src1 Addr];      // Get Data Operands
            src2Data=Dmem [src2 Addr];
            Result=src1  Data*src2Data;    // Calculate  Result
            Dmem[destAddr]=Result;         // Write Result Data
            #(`clk_period*2);              // MUL instruction takes three clock cycles
            #1 pc=pc+1;
      end

      `DIV:
      begin
            $display("DIV INSTRUCTION:");
            src1Data=Dmem[src1Addr];
            src2Data=Dmem[src2Addr];
            Result=src1Data/src2Data;
            Dmem[destAddr]=Result;
            #(`clk_period*2);              // DIV instruction takes three clock cycles
            #1 pc=pc+1;                    // increment program counter
      end

      `BRA:
      begin
            $display("BRA INSTRUCTION:");
            #(`clk_period);                // BRA instruction takes two clock cycles
            #1 pc=pc+braAddr;              // calculate new program counter
      end

     endcase

  end
end

/******************************************************************
*  Display Results
******************************************************************/
 initial
begin

      // Display a header for the test Output

       $display("\t           PROG        DEST  SRC1   SRC2  SRC1  SRC2  RESULT");
       $display("\t TIME  RESET  CNTR  OPCODE  ADDR  ADDR  ADDR  DATA  DATA
DATA");
```

```
        $display("\t----  ----  ----  ------  ----  ----  ----  ----  ------\n");

       // Display Results at every clock edge
        forever @(posedge clock)
        $display ("\t %0d \t %b    %h    %h    %h   %h   %h   %h   %h    %h",
        $time,reset,pc,opcode,destAddr,src1Addr,src2Addr,src1Data,src2Data, Result);

end



/********************************************************************
*  Initialize and Run
********************************************************************/
  initial
begin
            $readmemh("InstrMemory.data",Imem);        // Initialize instruction memory
            $readmemh("DataMemory.data",Dmem);         // Initialize data memory

            assign reset=1;                          // Initialize reset
           #220 assign reset=0;

           #1000
           $writememh("DataMemory2.data",Dmem,0,32);     // Write Out data memory
                                     // at end of simulation
           $ finish;

end
endmodule
/********************************************************************
*  Instruction Memory
********************************************************************/
00000102 // Opcode=ADD, DestAddr=00, SrcAddr1=01, SrcAddr2=02
01030405 // Opcode=SUB, DestAddr=03, SrcAddr1=04, SrcAddr2=05
02060708 // Opcode=MUL, DestAddr=06, SrcAddr1=07, SrcAddr2=08
03090aOb // Opcode=DIV, DestAddr=09, SrcAddr1=0a, SrcAddr2=0b
04040000 // Opcode=BRA, BraAddr =04
000c0d0e // Opcode=ADD, DestAddr=0c, SrcAddr1=0d, SrcAddr2=0e (SKIPPED
INSTRUCTION)
000f1011 // Opcode=ADD,DestAddr=0f, SrcAddr1=10, SrcAddr2=11 (SKIPPED
INSTRUCTION)
00121314 // Opcode=ADD, DestAddr=12, SrcAddrl=13, SrcAddr2=14 (SKIPPED
INSTRUCTION)
03151617  // Opcode=DIV, DestAddr=15, SrcAddr1=16, SrcAddr2=17
0218191a // Opcode=MUL, DestAddr=18, SrcAddr1=19, SrcAddr2=1a
011b1c1d // Opcode=SUB, DestAddr=1b, SrcAddr1=1c, SrcAddr2=1d
001e1f20 // Opcode=ADD, DestAddr=1e, SrcAddr1=1f, SrcAddr2=20
----------
X-Sun-Data-Type:  default
X-Sun-Data-Name: DataMemory.data
```

```
X-Sun-Content-Lines: 14
****************************************************************************
*  Data Memory
****************************************************************************/
00 01 02 // Data for Instruction #1  : Dest, Src1, Src2 (ADD Instruction)
00 02 01 // Data for Instruction #2 : Dest, Src1, Src2 (SUB Instruction)
00 05 10 // Data for Instruction #3 : Dest, Src1, Src2 (MUL Instruction)
00 80 02 // Data for Instruction #4 : Dest, Src1, Src2 (DIV Instruction)
00 00 00 // Data for Instruction #6 : Dest, Src1, Src2 (ADD Instruction, skipped)
00 00 00 // Data for Instruction #7 : Dest, Src1, Src2 (ADD Instruction, skipped)
00 00 00 // Data for Instruction #8 : Dest, Src1, Src2 (ADD Instruction, skipped)
00 40 04 // Data for Instruction #9 : Dest, Src1, Src2 (DIV Instruction)
00 05 0f// Data for Instruction #10: Dest, Src1, Src2 (MUL Instruction)
00 ff 01 // Data for Instruction #11: Dest, Src1, Src2 (SUB Instruction)
00 12 34 // Data for Instruction #12: Dest, Src1, Src2 (ADD Instruction)
---------

Data-Name: DataMemory2.data

03
01
02
01
02
01
50
05
10
40
80
02
00
00
00
00
00
00
00
00
00
10
40
04
4b
05
0f
fe
ff
01
46
```

```
12
34

X-Sun-Data-Name:  verilog.log

Host command:  verilog
Command arguments:
processor.v

VERILOG-XL 2.1.2  log  file  created Nov  11, 1995   15:52:56
VERILOG-XL 2.1.2    Nov  11, 1995   15:52:56


Compiling  source  file  "processor.v"
Highest  level  modules:
processor

              PROG          DEST  SRC1  SRC2  SRC1  SRC2  RESULT
     TIME RESET CNTR OPCODE  ADDR ADDR  ADDR DATA DATA DATA
----  -----  ----  ------  ----  ----  ----  ----  ----  ------

      5     1    00    xx    xx    xx    xx    xx    xx    xx
      50    1    00    xx    xx    xx    xx    xx    xx    xx
      100   1    00    xx    xx    xx    xx    xx    xx    xx
      150   1    00    xx    xx    xx    xx    xx    xx    xx
      200   1    00    xx    xx    xx    xx    xx    xx    xx
ADD INSTRUCTION:
      250  0    00    00    00   01    02    01    02    03
SUB  INSTRUCTION:
      300  0    01    01    03   04    05    02    01    01
MUL INSTRUCTION:
      350  0    02    02    06   07    08    05    10    50
      400  0    02    02    06   07    08    05    10    50
      450  0    02    02    06   07    08    05    10    50
DIV INSTRUCTION:
      500  0    03    03    09   0a    0b    80    02    40
      550  0    03    03    09   0a    0b    80    02    40
      600  0    03    03    09   0a    0b    80    02    40
BRA INSTRUCTION:
      650  0    04    04    04   00    00    80    02    40
      700  0    04    04    04   00    00    80    02    40
DIV INSTRUCTION:
      750  0    08    03    15   16    17    40    04    10
      800  0    08    03    15   16    17    40    04    10
      850  0    08    03    15   16    17    40    04    10
MUL INSTRUCTION:
      900  0    09    02    18   19    1a    05    0f    4b
      950  0    09    02    18   19    1a    05    0f    4b
      1000 0    09    02    18   19    1a    05    0f    4b
```

```
SUB INSTRUCTION:
     1050    0   0a   01   1b   1c   1d   ff   01   fe
ADD INSTRUCTION:
     1100    0   0b   00   1e   1f   20   12   34   46
     1150    0   0c   xx   xx   xx   xx   12   34   46
     1200    0   0c   xx   xx   xx   xx   12   34   46
L197 "processor.v": $finish at simulation time 1220
531 simulation events
CPU time: 0.3 secs to compile + 0.1 secs to link + 0.1 secs in simulation
End of VERILOG-XL 2.1.2 Nov 11, 1995 15:52:58
```

Example 5-37. A behavioral processor model

5.15 Exercises

1a. Using the model for the microprocessor in example, verify the functionality with given instruction and data memory (files instrm1.dat and datam.dat). Single step with "trace on" to see the simulation in detail. Obtain the trace for this.

1b. Add information to these files for testing the following instructions:

```
ADD DM[1], DM[6]
BRA   IM[8]
```

Verify the simulation and display the results for these

1c. Add code to the model for instructions of BREQ—(Branch on equal). This will have 3 operands—DM[addr1] and DM[addr2] { 2 addresses in data memory} and a Branch address:

```
BREQ addr1, addr2, iaddr
```

Verify this using the following:

```
SUB 5, 8, 9
BREQ 5,8, 1
```

2. Convert the following repeat loop into for and while loops.

```
parameter DivsLength = 31
parameter DivdLength 63
parameter QuoLenght 31
parameter HiDminimum = 32

repeat(DivsLength+1)
begin
        quotient = quotient << 1;
        dividend = dividend << 1;
        dividend[DivdLength: HiDminimum] = dividend[DivdLength:
HiDminimum] – divisor;
        if (! dividend [DivddLength])
                quotient = quotient + 1;
        else
        begin
```

```
                    dividend[DivdLength:Himinimum] =
                    dividend[DivdLength:Himinimum] + divisor;
            end
    end
```

3. Compute the results of the following blocking and non-blocking assignments:

```
module e (out);
        output out;
        reg a,b,c;
        reg d,e,f;

            initial
        begin
                $monitor ("time=%d a=%d b=%d    c=%d e=%d
\n",$time,a,b,c,d,e):
                a=0; b=1; c=0; d=0; e=1;
                #30 $finish;
        end

        always c = #5 ~c;

        always @(posedge c)
        begin
            a = b;
            b = a;
            d<= e;
            e <= d;
        end
endmodule
```

6 STRUCTURAL PRIMITIVE MODELING

6.1 Introduction

This chapter describes the predefined gates, switches, and user-defined primitives. The user-defined primitive enables extending set of built-in gates, and also allows one to model latches and flip-flops. This forms modeling with structural primitives in Verilog.

6.2 Gates

6.2.1 Introduction

Verilog defines a set of predefined modules that model the behavior of logic gates. These are common ways to describe implementation or structural details of a design. These are called built-in gates and have names like and, or, xor, not. The Table 6-1 in section 6.2.3 provides a complete list of gates and their functions. These use the 4-logic values 0-1-x-z and perform functions commonly understood for these—like and, or, nand, nor, xor, and xnor. The buf is a buffer and not is an inverter. Additional gates like bufif and notif gates use additional values of L and H. These are useful for reducing pessimism and are interpreted as:

L is either a 0 or X, but not 1

H is either 1 or X but not 0.

These gates represent TTL type transistors. Reducing pessimism can be understood to be the process of reducing number of unknowns (xs) during simulation.

6.2.2 Example

```
nand    #(10,10) nn     (x2, aIn, bIn);
nor (out, inl, in2, in3, in4);
not (outl, out2, in);
```

Example 6-1. Built-in gates modeling.

6.2.3 List of Gates and Their Functions

The list of built-in gates includes and, nand, or, nor, xor, xnor,buf, not, bufif0, bufifl, notifl, notif0, pullup and pulldown gates.

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

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

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

buf	IN	OUT
	0	0
	1	1
	x	x
	z	x

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

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

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

not	IN	OUT
	0	1
	1	0
	x	x
	z	x

pmos bufif0 rpmos		CONTROL			
		0	1	x	z
D	0	0	z	L	L
A	1	1	z	H	H
T	x	x	z	x	x
A	z	z	z	z	z

nmos bufif1 rnmos	0	1	x	z
0	z	0	L	L
1	z	1	H	H
x	x	z	x	x
z	z	z	z	z

Figure 6-1. Tables for Each Built-in Gate in Verilog

6.2.4 Syntax for Gates and Switch Declarations

gate_instantiation
    ::=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};
    | n_enable_gatetype [drive_strength] [delay3] enable_gate_instance{ , enable_gate_instance} ;
    | mos_switchtype [delay3] mos_switch_instance{ , mos_switch _instance} ;
    | pass_switchtype [delay3] pass_switch_instance{ , pass_switch_instance} ;
    | pass_en_switchtype [delay3] pass_en_switch_instance{ , pass_en_switch _instance} ;
    | cmos_switchtype [delay3] cmos_switch_instance{ , cmos_switch_instance} ;

| **pullup** [pullup_strength] pull-gate_instance{ , pull_gate _instance} ;
| **pulldown** [pulldown_strength] pull-gate_instance{ , pull_gate _instance} ;

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,
input_terminal});
enable_gate_instance ::= [name_of_ gate_instance] (output_terminal, input_terminal{, input_terminal,
enable_terminal});
mos_switch_instance ::= [name_of_ gate_instance] (output_terminal, input_terminal, enable_terminal});
pass_switch_instance ::= [name_of_ gate_instance] (inout_terminal, inout_terminal, enable_terminal});
pass_enable_switch_instance ::= [name_of_ gate_instance] (inout_terminal, inout_terminal, enable_terminal);
cmos_ switch_ instance ::= [name_of_ gate_instance] (output_terminal, input_terminal,ncontrol_terminal,
pcontrol_terminal);
pull_ gate_ instance ::= [name_of_ gate_instance] (output_terminal)
name_of_ gate_instance ::= gate_instance_identifier[range]
pullup_strength ::= (strength0, strength1)
| (strength1, strength0)
| (strength0)
input_terminal ::= scalar_expression
enable_terminal ::= scalar_expression
ncontrol_terminal ::= scalar_expression
pcontrol_terminal ::= scalar_expression
output_terminal ::= terminal_identifier | terminal_identifier[constant_expression]
inout_terminal ::= scalar_expression

n_input_gatetype ::=
and | **nand** | **or** | **nor** | **xor** | **xnor**

n_output_gatetype ::= **buf** | **not**

enable_gatetype ::= **bufif0** | **bufif1** | **notif0** | **notif1**

mos_switch_type ::= **nmos** | **rnmos** | **pmos** | **rpmos**
cmos_switch_type ::= **cmos** | **rcmos**
pass_switch_type ::= **tran** | **rtran**
pass_switch_type ::= **tranif0** | **rtranif0** | **tranif1** | **rtranif1**

delay3 ::= #delay_value | #(delay_value [,delay_value [,delay_value]])
delay2 ::= #delay_value | #(delay_value [,delay_value])
delay_value
::= unsigned_number
| parameter_identifier
| (mintypmax_expression [,mintypmax_expression] [,mintypmax_expression])

6.3 Switches

6.3.1 Introduction

This modeling level contains details of design up to the transistors and even down to capacitor or resistor. This level uses strength algebra to digitize the effects of basic charge and voltage changes in a block of logic. Some switch level designs are not convertible to gate-level models and such parts of logic are modeled using the constructs like nmos, pmos, or tranif0, tranif1. These reproduce technology dependent behavior as well. The list of switch level primitives in Verilog includes pmos, nmos, cmos, rnmos, rpmos, tran, tranif0, tranif1, rtran, rtranif0), rtranif1, pullup, pulldown. The strength values and the switch modeling is discussed in more detail in Chapter 17. The gates that begin with the letter r are resistive transistors that cause a step-down in values.

6.3.2 Syntax

The syntax here is identical to the gate-declaration and is given above along with gate-declaration. See the second line in the list of <GATETYPE>s for the complete switch-level primitive names. Drive strength of a gate was mentioned in section 6.2.4, but is expanded in this section.

```
drive_strength
    ::=(STRENGTH0), STRENGTH1)
    | (STRENGTH1 ,STRENGTH0)
```

STRENGTH0 is one of the following keywords:
supply0 strong0 pull0 weak0 highz0

STRENGTH1 is one of the following keywords:
supply1 strong1 pull1 weak1 highz1

6.4 User-Defined Primitives

6.4.1 Introduction

The User-Defined Primitive provides ways of using truth-tables as new primitives at the gate-level. It supports combinational and sequential circuits. In sequential, both level-sensitive and edge-sensitive and mixed descriptions are supported. A user-defined primitive is expected to be compiled into a single table lookup for its evaluation. Only 1 output per User-Defined Primitive is supported. Thus multiple user-defined primitives are needed, 1 per output for a meaningful logic block with multiple outputs. In the sequential designs, only 1 internal state for sequentials is supported. Everything is strictly single bit with 0, 1, x values on input and 0, 1, x, z on the output. These are the most efficient way to model; reason being the # steps in evaluations for gates, primitives, RTL, behavioral. Some idea of how things work

inside a simulator is useful to model efficiently. An automatic production is possible from other types of descriptions and by good programming.

User-Defined Primitives are commonly known as UDPs. In a udp, spaces and cases do not matter. In general, conflicting entries, where two different descriptons of the same input set has different outputs, will be detected and reported. Exception to this is the mixed descriptions of edge-sensitive and level-sensitive sequential nature. Conflicts in such entries is allowed as explained in section 6.8.

6.4.2 Examples

```
// This is a combinational primitive with 3 inputs and 1 output
    primitive carry(carryOut, carryIn, aIn, bIn);
    output      carryOut;
    input       carryIn,
            aIn,
            bIn;

        table
            0   00   :   0;
            0   01   :   0;
            0   10   :   0;
            0   11   :   1;
            1   00   :   0;
            1   01   :   1;
            1   10   :   1;
            1   11   :   1;
        endtable
    endprimitive
```

Example 6-2. A user-defined primitive definition.

6.4.3 Syntax

UDP DECLARATION and INSTANTIATION

UDP_declaration
    ::= **primitive** UDP_identifier (udp_port_list);
        UDP_port_declaration {udp_port_declaration}
        udp_body
        endprimitive

udp_port_list ::= output_port_identifier, input_port_identifier { , input_port_identifier }

udp_port_declaration ::=
    output_declaration
    | reg_declaration
    | input_declaration

udp_body ::= combination_body | sequential_body

combination_body ::= table combinational_entry { combinational_entry } endtable

sequential_body ::= [UDP_initial_statement] table sequential_entry { sequential_entry } endtable

UDP_initial_statement
    ::= initial output_terminal_name = init_val;

init_val
    ::= **1'b0**
    | **1'b1**
    | **1'bx**
    | **1'bX**
    | **1'B0**
    | **1'B1**
    | **1'Bx**
    | **1'BX**
    | **1**
    | **0**
output_terminal_name
    ::= variable

combinational_entry
    ::= level_input_list : OUTPUT_SYMBOL ;

sequential_entry
    : := seq_input_list : current_state : next_state ;

seq_input_list
    ::= level_input_list
    | 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)
    | EDGE_SYMBOL

current_state
    ::= LEVEL_SYMBOL

```
next_state
      ::= OUTPUT_SYMBOL
      | -
```

OUTPUT_SYMBOL is one of the following characters:
0 1 x X

LEVEL_SYMBOL is one of the following characters:
0 1 x X ? b B

EDGE_SYMBOL is one of the following characters:
r R f F p P n N *

```
UDP_instantiation
      ::= UDP_identifier [drive_strength] [delay2]
      UDP_instance{ ,UDP_instance};

UDP_instance
      ::= [name_of_UDP_instance] (output_port_connection, input_port_connection
{output_port_connection ,input_port_connection} )
      ::= [IDENTIFIERrange]

name_of_UDP_instance::= UDP_instance_identifier[range]
```

6.5 Combinational UDPs

6.5.1 Introduction

These can have up to 10 inputs and 1 output. There is no state column. UDPs describe the truth-table by enumerating cases. The default rule is that all unstated combinations result in x output. Idea here is to not assume anything extra within the model or the simulator. Unstated combinations must be truly don't care. Use of special symbols to increase readability and writability as follows can be done in combinational and sequential udps.

? – iterate over 0, 1, x (Don't care on input)

b – iterate over 0,1

UDP entries are case insensitive. The symbol b and B are same internally and so are the symbols x and X.

6.5.2 Example

```
// This is a combinational primitive with 3 inputs and 1 output
      primitive carry(carryOut, carryIn, aIn, bIn);
      output        carryOut;
      input         carryIn,
                  aIn,
                  bIn;
```

```
        table
            0    00    :    0;
            0    01    :    0;
            0    10    :    0;
            0    11    :    1;
            1    00    :    0;
            1    01    :    1;
            1    10    :    1;
            1    11    :    1;
        endtable
    endprimitive

    module test_carry;
                reg a, b, c;
                integer i;
                wire cout;
                  carry ci(cout, c, a, b);
                initial
                begin
                $monitor("time=%d a= %b b=%b c=%b ---------
cout=%b",$time,a,b,c,cout);
                for (i=0; i<9; i=i+1)
                begin
                    {a,b,c}=i;
                    #10;
                end
                a = 'bx;
                #10
                b= 'bx;

                end
    endmodule
```

Example 6-3. ***A combinational user–defined primitive: An adder part – carry computation.***

6.5.3 Syntax

This is included with overall UDP syntax specification in section 6.4.

6.6 Level-Sensitive Sequential UDP

6.6.1 Introduction

These are similar to combinational UDPs s, but have a state column. Some additional symbols like '-' in output column indicates no change. These are one state only primitives.

6.6.2 Example

```
primitive latch (q, clock, data);
output      q;
reg         q;
input       clock, data;
      table
//          clock       data        state       output
                0           1       : ? :       1;
                0           0       : ? :       0;
                1           ?       : ? :       -;
      endtable
endprimitive

module test_latch;
      reg a, b, c;
      integer i;
      wire q;
        latch li(q , a, b);
      initial
      begin
      $monitor("time=%d a= %b b=%b---------q=%b",$time,a,b,q);
      for (i=0; i<5; i=i+l)
      begin
            {a,b} = i;
            #10;
      end
      a = 'bx;
      #10
      b= 'bx;

      end
endmodule
```

Example 6-4. Level sensitive sequential UDP - a latch.

6.6.3 Syntax

This is included with overall UDP syntax specification in section 6.4.3.

6.7 Edge Sensitive Sequential UDPs

These user-defined primitives are similar to level-sensitive, but have edges on inputs. These have only one edged input per entry. An edge descriptor is represented as (from_value to_value) An example of edged input entry is (01). Some of the special symbols used here are:

r represents (01)

f represents (10)

p represents (0?) and (?1),

n represents (?0) and (1?)

* represents all edges

Edge sensitive is SUPERSET of level sensitive in the sense that all behavior from level-sensitive entries can be equivalently represented by edge sensitive entries, but the vice versa is not true.

6.8 Mixed Level and Edge Sensitive Sequential UDPs

These descriptions are allowed in UDPs. The motivation behind such usage is the modeling of edge-triggered flip-flops with resets and clears. Reset and Clear are level-sensitive while the rest of the active behavior is edge-sensitive. A conflict may arise and in case of reset and clear, those behaviors dominate over the clocked behavior. Thus, level-sensitive behavior takes precedence over edge-sensitive in case of conflicts in entries of these two types. For conflicts on entries of same type, an error like in other primitives (pure level or pure edge) is an error. The conflict resolution rules are meant to describe the hardware behavior correctly.

```
// Following is the model for SR edge-triggered flip-flop

primitive sr_edge_prim(q, clear, s, r, clock);
    output q;
    reg q;
    input clear, s, r, clock;
    table
            //clear   s   r   clock     qcur     q
            1   ?   ?   0     :   ?   :   0;

            0   0   0   (??)  :   ?   :   -;
            0   1   0   r     :   ?   :   1;
            0   0   1   r     :   ?   :   0;
            ?   ?   ?   f     :   ?   :   -;
            ?   *   ?   ?     :   ?   :   -;
            ?   ?   *   ?     :   ?   :   -;
    endtable
endprimitive

module sr_edge_ff(clear, s, r, clock, q, qbar);
    inout q, qbar;
            input clear, s, r, clock;
    sr_edge_prim pi (q, clear, s, r, clock);
    not (qbar, q);
endmodule

module test_sredge;
    reg s, r, clear;
    sr_edge_ff s1 (clear, s, r, clock, q, qbar);
    m555 ml (clock);
    initial
    begin
```

```
            #10
                    clear = 1;

            #200
                    clear = 0;

            #200
                    s=1; r =0;

            #200
                    s=0; r=1;

            #200
                    s=0; r=0;

            #200
                    $finish;
        end

        always
                #200
                    $display("time=%d clear=%d s=%d r=%d clock=%d q=%d
                        qbar=%d", $time, clear, s, r, clock, q, qbar);
    endmodule

module m555 (clock);
    output          clock;
    reg             clock;

    initial
            #5 clock = 1;

    always
            #50 clock = ~ clock;
endmodule
```

Example 6-5. ***Mixed edge and level sensitive sequential UDP - SR flip-flop with clear.***

6.9 UDP Instances

6.9.1 Introduction

UDPs are instantiated just like modules. A primitive name followed by instance name and the list of ports will make the instance like for modules or gates. Delays can be used for udp-output like those for built-in gates. Parameters are not used for upds.

6.9.2 Example

```
module test_k;
    wire q, clock, j, k, preset, clear;
    reg j, k;
    jkEdgeFF jk (q, clock, j, k, preset, clear);

    initial
        begin
            reset_ff;
            #50;
                    j = 1;
                k = 0;
            #50;
        end
        m555 timer (clock);
endmodule
```

Example 6-6. User-defined primitives – instances.

6.9.3 Syntax for Primitive Instance

```
UDP_instantiation
    ::= UDP_identifier [drive_strength] [delay2]
    UDP_instance{ ,UDP_instance);

UDP_instance
        ::= [name_of_UDP_instance] (output_port_connection, input_port_connection
        {output_port_connection ,input_port_connection} )
    ::= [IDENTIFIER[range]
        name_of_UDP_instance: := UDP_instance_identifier[range]
```

6.10 Exercises

1. Find all the syntactic and semantic errors in the following Verilog module:

```
module IOBuffer(bus, in, out, dir)
    inout bus;
    reg bus;

    parameter
        R_Min = 3, R_Typ = 4, R_Max = 5,
        F_Min = 3, F_Typ = 5, F_Max = 7;
        Z_Min = 12, Z_Typ = 15, Z_Max = 17;

    bufif1      #(R_Min, R_Typ, R_Max :
                F_Min, F_Typ, F_Max:
                Z_Min, Z_Typ, Z_Max)
            (bus, out, dir);
    latch_prim
            (in, clock, bus);
```

```
        m555 (clock);
    endmodule

    primitive latch (q, clock, data);
        output          q;
        reg        q;
        input           clock, data;
        wire  clock, data;
        table
        //    clock      data       states      output
              0          1          : ??:       1;
              0          0          : ? ?:      0;
              1          ?          : ? ?:      -;
        endtable
        endprimitive
```

2. Convert the following behavioral description into a user-defined primitive. After doing this, add cases to reduce pessimism on x values for bits in flag.

```
module m(out, in1, in2, in3, in4, flag);
outputout;
input in1, in2, in3, in4;
input [1:0] flag;
reg out;

always
    @ {case, in1 , in2 , in3 , in4 }
    case (flag)
        0 : out = in1;
        1 : out = in2;
        2:  out = in3;
        3 : out = in4;
    endcase
endmodule
```

3. Write results from the following Verilog model://. Following is the model for SR edge-triggered flip-flop:

```
primitive sr_edge_prim(q, clear, s, r, clock);
        output q;
    reg q;
    input clear, s, r, clock;

    table
    //clear s  r  clock  qcur q
          1  ?  ?   0  : ? : 0;
          0  0  0   (??): ?  : -;
          0  1  0   r    : ? : 1;
          0  0  1   r    : ? : 0;
          ?  ?  ?   f    : ? : -;
          ?  *  ?   ?    : ? : -;
          ?  ?  *   ?    : ? : -;
```

```
        endtable
        endprimitive

    module sr_edge_ff(clear, s, r, clock, q, qbar);
        inout q, qbar;
        input clear, s, r, clock;
        sr_edge_prim pi (q, clear, s, r, clock);
        not (qbar, q);
    endmodule

    module test_sredge;
        reg s, r, clear;
        sr_edge_ff s1 (clear, s, r, clock, q, qbar);
        m555 m1 (clock);
        initial
        begin

            #10
                clear = 1;

            #200
                clear = 0;
            #200
                s=1; r=0;
            #200
                s=0; r=1;
            #200
                s=0; r=0;
            #200
                $finish;
        end

        always
            #200
            $display("time=%d clear=%d s=%d r=%d clock=%d q=%d qbar=%d",
                $time, clear, s, r, clock, q, qbar);
    endmodule

module m555 (clock);
    output      clock;
    reg     clock;

    initial
        #5 clock = 1;

    always
        #50 clock = ~ clock;
endmodule
```

4. a. Create a user-defined primitive (UDP) for the following boolean equation:

   out = (a1 & a2 & a3) | (b1 & b2)

   This UDP has five inputs a1, a2, a3, b1, and b2 and one output out.

   b. Simulate this for the following stimulus block added to the UDP instance:

```
initial
begin
    $monitor($time, a1, a2, a3, b1, b2, out);
    for (i=0; i < 32; i=i+l)
        #10 {a1,a2,a3,b1,b2} = i;
end
```

5. Instantiate the edge-sensitive dff UDP discussed in class to create a module with q and qbar outputs with a delay of 5 units on q and 7 units on qBar.

7 MIXED STRUCTURAL, RTL, AND BEHAVIORAL DESIGN

7.1 Introduction

One of the sources of power of expressing design in Verilog comes from the ability to mix the three design styles freely in a module and across modules. This is useful for flexible design methodology, and also for system level modeling. This is also good for performing the following functions all within the realm of Verilog.

- Design
- Test-Bench
- Timing
- Synthesis.

The design modeling is discussed throughout this book. Several examples of test-benches are given along with the models in examples until now. Timing will be discussed in specify blocks. Synthesis will be discussed in synthesis chapter. The order of events in structural, RTL, and behavioral blocks happening at same time is indeterminate in general, but simulation starts only with **initial** and **always** statements. In general, all three kinds of blocks are concurrently executed.

7.2 Examples and Scenarios: 1 – Comparing Structural Adder Design with Behavioral Model

In this, we test an adder for correctness by applying random inputs and checking if the result matches RTL add operation.

```
module mixed1();
    wire[31:0] out;
    wire carryout;
    reg in1, in2, carryin;
```

```
        assign out = in1 + in2;

        fullAdder_s a (carryout, out, in1, in2, carryin);
    always @out
            if (out==='bx)
              $display("time=%d out has become x", $time, 'bx);

    initial
        begin
                repeat(10000)
                begin
                #10
                    in1 = $random;
                    in2 = $random;
                end
        end
    endmodule

module fullAdder_s(cOut, sum, aIn, bIn, cIn);
    output   cOut, sum;
    input  aIn,  bIn,  cIn;

    wire      x2;

    nand      (x2, aIn, bIn),
              (cOut, x2, x8);
    xnor      (x9, x5, x6);
    nor       (x5,x1,x3),
              (x1, aIn, bIn);
    or        (x8,x1,x7);
    not       (sum, x9),
              (x3, x2),
              (x6, x4),
              (x4, cIn),
              (x7, x6);
endmodule
```

Example 7-1. Comparisons of different levels of abstraction by mixed level design.

7.3 Examples and Scenarios: 2 – System Modeling

In the following system, we are using as asic which was developed previously using structural level design method. We also have a microprocessor developed at behavioral level that can handle the instruction set. The memory read and write cycles are RTL level models. A parity generator at switch level is used.

```
module system();
    micro mbehav();
    mem mrtl();
    graph masic();
    parity_check pswitch();
endmodule
```

Example 7-2. ***System modeling with behavioral, rtl, gates and switches mixed in one board design.***

7.4 Examples and Scenarios: 3 – Adding Behavioral Code to a Design for Checking

External models in VHDL or other simulators can be linked using Programming Language Interface and system simulation can proceed.

In the following example, we have a gate-level model of adder mixed with a small behavioral section to generate a special output that tells us whether the adder resulted in zero as sum. Thus, a special adder is created effortlessly. This part of the design can be implemented in terms of gates at a later stage.

```
module adder(in1, in2, cin, out, cout, zero_flag);
/* repeat structural adder from chapter 4; add 1 o/p*/
/* add block of code as follows */
    output out, zero_flag, cout;
    reg zero_flag;
    input in1, in2, cin;
        always@out
        if(out==0)
            zero_flag = 1;
        else
            zero_flag = 0;

                    nand        (x2, in1, in2),
                        (cout, x2, x8);
                xnor    (x9, x5, x6);
                nor     (x5, x1, x3),
                        (x1,in1,in2);
                or      (x8, x1,x7);
                not     (out, x9),
                        (x3, x2),
                        (x6, x4),
                        (x4, cin),
            (x7, x6);
    endmodule
```

Example 7-3. ***A behavioral flag-bit generation mixed with adder of rtl or structural style.***

7.5 Examples and Scenarios: 4 – Design Cycle and Project Planning Flexibility

In the example of flip-flop in Chapter 1, the reset was modeled behaviorally while the rest of the flip-flop was made of gates. This is an example of a design that is a special design with the reset being connected to the system reset for all such flip-flops and not available for connections within the design. Thus, mixing of this behavioral abstraction allows us to make special situations available locally and the get the design going quickly.

In a scenario where part of the design is synthesized and part of it is hand-designed, the two paths may have different project-plans. In this case, one can test the gate-level implementation of one type with the RTL implementation of the other. Thus, in a design cycle as one refines certain parts of the design downto details, simulating those with parts which are still RTL and behavioral is a great way to test these parts. The mixed style allows one to plan the project flexibly and still be effective in simulating various sections within a system.

7.6 Exercises

1. Add code to check for parity (assume even parity) for Example 7-3. Add another flag to report errors.

2. Synthesis will be discussed in Chapters 12 to 14. Verifying results of synthesis involves mixing gate-level and behavioral or rtl level code. Verify the results from synthesis of Examples 12-1 to 12-4 with their pre-synthesis results and compare for accuracy. You may want to do this mixed modeling exercise after you have completed studying Chapter 12.

3. Implement parts of the microprocessor in Example 5-37 using ALU of Example 3-27. Create a decoder similar to the multiplexer in Example 3-27 and instantiate this decoder. Model the control using a controller of the style of cache controller of Example 11-5. Connect the behavioral and implementation (RTL) level models of the microprocessor and verify the functional correctness using the tests of Example 5-37.

8 SYSTEM TASKS AND FUNCTIONS

8.1 Introduction

These are predefined tasks and functions built into Verilog HDL. Syntactically all system tasks and functions always begin the symbol '$'. They provide functions such as

- Input-Output from Files, Screen, and Keyboard;
- Simulation Control and Debugging;
- Timing Checks, Stochastic, and Probabilistic Analysis;
- Conversion Functions between different types

These system tasks and functions typically do not describe hardware per se, although there are a few exceptions.

Commonly used system tasks and functions are :

$display	display values
$monitor	trace value-changes
$fopen, $fclose	open, close a file
$readmem	memory read tasks
$time	simulation time
$finish, $stop	end, stop simulation
$dumpvars	dump data to file for waveform display
$setup, $hold	setup and hold timing checks

These along with similar other tasks will be described in detail in the following sections of this chapter.

8.2 Display System Tasks

8.2.1 Overview

The displaying of values and strings to the standard output is done via display tasks. These are:

$display $displayb $displayh $displayo

The $display tasks will display formatted values and strings in user-defined format while the other three display values in binary, hex, or octal form.

8.2.2 Examples

```
module disp;
reg [0:31] rval, rvall, i;
    initial
    begin
       rval =101;
       rvall = 2020;
       $display("rval=%h hex =%d decimal",rval,rval);
       $display("rval=%0h hex =%0d decimal" ,rval,rval);
       $displayh("rval = %d",rval);
       $display ("table of hex values");
       $display("rval rvall");

       $displayh(rval[0:7],  rvall [0:7]);
       $displayh(rval[8:15],   rvall[8:15]);
       $displayh(rval[16:23],rvall[16:23]);
       $displayh(rval[24:31],  rvall [24:31]);

       $display("example of displayb follows");
       $displayb(rval);
       $display("example of displayo follows");
       $displayo(rval);

       $display("Simulation time is  %t",$time);

       $display("%d", 1'bx);
       $display("“%h", 14'bx01010);
       $display("%h   %o", 12'b001xxx101x01, 12'b001xxx101x01);
               $display("Current scope is %m");
    end
endmodule
```

Example 8-1. $display usage with different methods of capturing design data.

On simulating the above example, the standard out will display the following result:

```
rval=00000065 hex =       101 decimal
rval=65 hex =101 decimal
table of hex values
rval rval1
0000
0000
0007
65e4
example of displayb follows
00000000000000000000000001100101
example of displayo follows
00000000145
Simulation time is                    0
Current scope is disp
x
xxXa
XXX lx5X
```

Thus, the normal $display is used to display values using formatting strings. The $displayh, $displayb, and $displayo are used to display values in hex, bin, and octal format and are more useful for creating output tables. The $write set of tasks is same as $display except that no newline is added at the end of this task. This helps in creating output from different places in the description and put it on one line.

8.2.3 Syntax and Format Details

display_tasks ::= display_task_name(list_of_arguments)
display_task_name **::= $display | $displayb | $displayh**
| $displayo | $write | $writeo | $writeh | $writeb

The following table shows the format specifiers.

%h hexadecimal
%b binary
%o octal
%d decimal
%c ascii character
%v net strength
%m module name (full hierarchical)
%s string
%t current time format
%e real number in exponent form
%d real number in decimal form
%g display real in the 2 formats but shortest width

The following table shows special characters used in the string part of $display("string",..). These will help special control over output.
'%o' modifier will take away heading spaces or zeroes, e.g., '%od' or '%ob'

\n NewLine
\t Tab character
\\ \ character
\" " character
\o a character in octal 1-3 digits
%% is the % character

Explanation

Amongst the % format specifiers, both %d and %t can be used to obtain time. Originally, time was 32-bits wide in Verilog HDL and %d was used to get the value. However, the simulations of complex systems were done for longer times and the time was extended to be 64 bits (also competitive with VHDL implementations). A new format specifier %t was added. Thus, the older %d specifiers continued to work as they are while %t now will support upto 20 digits (full 64-bit value).

Size considerations

The $display set of tasks allocate spaces needed for maximum value for that expression. For the example in 9.2.2, we can see the results with leading 0s for hex as the size is 32 bits or 8-hex digits. To use minimum size(with no leading spaces or 0s), use %0h, %0d, %0b and %0o format specifiers.

X And Z Considerations

When all bits are unknown, 'x' is displayed . When all bits are high impedance, 'z' is displayed. For partial unknowns, 'X' is displayed and for partial high impedances, 'Z' is displayed. See last three lines of the example 8-1 above.

8.3 Monitor System Tasks

8.3.1 Overview

A task is provided to trace or monitor the values of nets and regs as they change. When a $monitor task is invoked with one or more arguments, the simulator sets up a mechanism whereby each time a variable or expression in the argument changes value, the entire argument is displayed as in $display.

This trace can be turned on or off by special control tasks $monitoron and $monitoroff. The $monitoroff turns off displaying the monitored values and $monitoron turns it back on.

The $strobe set of tasks have functionality like $display but have the characteristic of $monitor that the values displayed are at the end of the simulation time after all changes for that time are complete.

8.3.2 Examples

```
module test_adder;
    // test a 2 bit adder using $monitor
    reg[1:0] in1, in2;
    wire [2:0] out;
    adder a (out, in1, in2);
    initial
    begin
        $monitor("time=%d out=%d in1=%d in2=%d",$time,out,in1,
            in2);
        in1=0;
        in2 = 0;
        repeat (4)
        begin
            repeat(4)
                #5
        in2 = in2+1;
        in1 = in1+1;
        end
    end

    initial
    begin
      #30 $monitoroff;
      #15 $monitoron;
      #30 $finish;
    end

endmodule

module adder(o,i1,i2);
    output [2:0]o; input [1:0]i1, i2;
    assign    o = i1 + i2;
endmodule
```

Example 8-2. ***Capturing simulation results of a design selectively with $monitor and $monitoron/off.***

This example produces the following output:

```
time=          0 out=0 in1=0 in2=0
time=          5 out=1 in1=0 in2=1
time=         10 out=2 in1=0 in2=2
time=         15 out=3 in1=0 in2=3
time=         20 out=1 in1=1 in2=0
time=         25 out=2 in1=1 in2=1
time=         45 out=3 in1=2 in2=1
time=         50 out=4 in1=2 in2=2
time=         55 out=5 in1=2 in2=3
```

```
time=            60 out=3 in1=3 in2=0
time=            65 out=4 in1=3 in2=1
time=            70 out=5 in1=3 in2=2
Exiting VeriWell for Win32 at time 75
```

From time = 0 to time = 25, $monitor records changes in its parameters. At 25, $monitoroff takes effect and no monitor outputs are produced till time 45 when the $monitoron again kicks in. The tasks of $monitoroff and Smonitoron are useful in a debugging session(especially for a long run) when one can selectively turn this on and off around problem times, saving one from unwieldy large output.

8.3.3 Syntax

```
monitor_tasks::=monitor_task_name[(list_of_arguments)];|
                $montoron;|
                $monitoroff;
monitor_task_name ::= $monitor| $monitorb| $monitorh| $monitorf
                    | $strobe | $strobeb | Sstrobeh | $strobeo
```

8.4 File Management

8.4.1 Overview

Verilog provides ability to perform input/output from files. The files are opened and closed using $fopen and $fclose system tasks. These are similar to open() and close() calls in “C", although the return values are binary and can be combined to perform ios from multiple files at the same time, unlike "C". The opened files can then be used in display and monitor tasks which have variations like $fmonitor and $fdisplay with first argument as file descriptor. Again this is similar to printf and fprintf in "C". The file descriptors in Verilog are also known as multi-channel-descriptors.

8.4.2 Examples

In the following example, we modified the Example 8-2 to redirect the output to file adderout.mon. This is specially useful in a larger design where outputs from each sub-part will be saved in a different file and then can be checked for correctness independently.

```
module test_adder;
  integer f1;

    // test a 2 bit adder using $fmonitor
    reg[1:0]in1, in2;
    wire [2:0] out;

    adder a (out, in1, in2);
    initial
    begin
```

```
            f1 =  $fopen("adder_out.mon");
            $fmonitor(fl,"time=%d out=%d inl=%d in2=%d",$time,out,
                  in1,in2);
            inl = 0;
            in2 = 0;
            .........
            // remaining example is same as before
      end
endmodule
module  adder(...);
....
endmodule
```

Example 8-3. File management in capturing results of simulating a design.

Running this will now produce no results on screen but will create a file adder_out.mon with the same output as before. $fmonitor also has the advantage of creating multiple monitors while only one $monitor can be active any time.

8.4.3 Syntax

file_open_function ::=
integer **$fopen**("filename");
file_close_task ::=
$fclose(multi_channel_descriptor);

file_output_tasks ::=
file_output_task_names(multi_channel_decriptor,
list_of_arguments);
file_output_task_name ::=
$fdisplay | $fdisplayb | $fdisplayh | $fdisplayo |
$fwrite | $ fwriteb | $fwriteh | $fwriteo |
$fstrobe | $fstrobeb | $fstrobeh | $fstrobeo|
$fmonitor | $fmonitorb | $fmonitorh | $fmonitoro

8.5 File Input Into Memories

8.5.1 Overview

The only form of input from files provided in IEEE 1364 Verilog HDL system tasks is the reading of input patterns into a memory. These tasks have names of $readmemb and $readmemh for reading binary and hex values. As the regs and memories are only bit-valued this is meaningful.

8.5.2 Example

```
module test_adder;
   reg [3:0] mem [1:16];
   // test a 2 bit adder using $fmonitor
```

```
    reg [1:0]inl,in2;
    wire [2:0] out;
    adder a (out, inl,in2);
    initial
       $readmemh("mem.dat", mem);
    initial
    begin
       $monitor("time=%d out=%d inl=%d in2=%d",$time,out,inl,in2);
       for(i=l;i<=16;i=i+l)
            #5 {inl, in2 } = mem[i];
    end
 endmodule
 // Insert the adder module definition from 9.3.2 here
 //mem.dat file contains 16 lines with 0 to f values
```

Example 8-4. Reading input from files – $readmem usage.

This will result in same output as in section 9.3.2 once the remaining lines from that example are added back here.

8.5.3 Syntax

```
load_memory_tasks ::=
    $readmemh("file_nam", memory_name, [,start_addr, end_addr]]);
    |$readmemb("file_nam", memory_name, [,start_addr, end_addr]]);
```

The data file contains data and addresses where address begin with symbol @.

8.6 Simulation Time Functions

The functions $time and $stime provide the simulation time values. These can be either displayed using %d(32 bit) or %t (64 bit) format or used in expressions. Examples of usage of $time is already shown in $monitor example in section 8.3.2 and other places.

8.7 Simulation Control Tasks

8.7.1 Overview

Verilog provides ability to break and also to end simulation in the form of tasks. This enables stopping or ending simulation based on certain conditions from within the model. This is also useful in test-bench and debugging aspects of simulation The tasks $stop breaks the simulation and brings it in interactive mode. The task $finish ends the simulation.

8.7.2 Examples

```
// Here is an example of adder which is being developed
// We perform $stop if the output goes to x. This way
// debugging can be performed on the system right away
```

```
// without having to start from the beginning again.

module  test_adder;

    // All lines here are same as in the 9.3.2
        // Add the following always block to this module
    always @out
        if (out=== 'bX)
            $stop;

endmodule

module adder(...);
        .......
endmodule
```

Example 8-5. ***Simulation control tasks – $stop and $finish.***

8.8 Waveform Interface (VCD Files)

8.8.1 Overview

Verilog provides a set of tasks to save the value-changes in a file that are accessed by a waveform viewer to display the results in the form of a waveform display. These files are called value change dump files (VCD). The tasks that relate to these are $dumpfile and $dumpvars. These files are ASCII files and can be used by other post-processing tools such as tester interface tools as well.

Like the monitor on and off controls, $dumpon and $dumpoff stop and restart the dumping operations.

8.8.2 Examples

```
// This is same example as in 9.3.2, but the dumping
// tasks are added after $monitor statement
module test_adder;
    // test a 2 bit adder using $monitor
    reg[1:0] in1, in2;
    wire [2:0] out;
    adder a (out, in1, in2);
    initial
    begin
        $monitor("time=%d  out=%d  in1=%d  in2=%d",$time,out,
            in1,in2);
        $dumpfile("ex_dump");
        $dumpvars;
        inl = 0;
        in2 = 0;
        ..........
```

```
        end

        initial
        begin
           #30 $monitoroff;
              ..........
        end

endmodule

module adder(o,il,i2);
...........
endmodule
```

Example 8-6. ***Creating a waveform data file using $dumpvars and related system tasks.***

The dumpfile ex_dump is produced by this. The files contents are shown below:

```
$date
Mon Jul 01 11:18:41 1996

$end
$version
 VeriWell for Win32 2.0.5
$end
$timescale
  1s
$end


$scope module test_adder $end
$var reg     2 !   in1 [1:0]$end
$var reg     2 "   in2 [1:0] $end
$var wire    3 #   out [2:0] $end

$scope module a $end
$var wire    2 $    i2[1:0] $end
$var wire    2 %    il[1:0]  $end
$var wire    3 &    o [2:0] $end
$upscope $end

$upscope $end

$enddefinitions    $end
#0
$dumpvars
b0 &
b0 %
b0 $
b0 #
```

```
.....
....
#10
...
...
#15
```

8.8.3 Syntax

```
dumpfile_task ::= $dumpfile("modulel.dmp");
dumpvars_task ::= $dumpvars(levels[,list_of_modules_or_variables]);
list_of_modules_or_variables ::= module_or_variable { ,
                                 module_or_variable}

module_or_variable ::= module_identifier | variable_dentifier
```

8.9 Exercises

1. What is the difference between $display and $write system tasks?
2. Write the format specifiers in $display and $monitor to :
   - indicate strength values
   - indicate current hierarchical instance and module name
   - display real numbers with shortest widths
3. Write a simulation capture module using system tasks for capturing the value changes at simulation times 25, 40 and 55 that is stabilized at that time using $fstrobe tasks. Use a filename "strobe_out" for writing the results[Use the Example 8-2 without the $monitor line]
4. Run Example 8-6 on the simulator and view the waveforms on the simulator provided with the book.

9 COMPILER DIRECTIVES

9.1 Introduction

The Compiler directives direct the pre-processor part of Verilog Parser or Simulator. These are very similar to "C" pre-processor directives which transform the input code into an output that is processed based on the directives. Some of the processing involves substitution of strings, conditional inclusion and exclusion of code and setting defaults. The character of "`" [back-quote] precedes all compiler directives. The scope of a directive is independent of module definitions; the scope extends from the point where the directive occurs to the next compiler directive that changes the prior directive. This may go across files or to the end of file or all files.

The key compiler directives are:

```
`include           ---- include another file here
`define            ---- define a macro[symbol]
`undef             ---- undefine a symbol

`ifdef             ---- These three are conditional compilation
`else                   directives
`endif

`default_nettype   ---- define a default net type
                        for the entire design

`timescale         ---- Define the timescale to be used
                        for the subsequent part of design

`resetall          ---- reset all directives back to
                        original default values
```

```
`celldefine  ---  These define a cell name used by
`endcelldefine    certain PLI routines
```

The compiler directives apply globally to the design until they are redefined or in case of undef undefined. They have an effect of separating the features from the core language, and also can save significant compilation time.

9.2 'include

9.2.1 Introduction

Verilog models can be organized into different files and then compiled together as one unit. One of the useful features in this regard is the `include compiler directive. A file whose name follows the `include compiler directive will be included during the compilation of the model in the preprocessing phase and carried over to the subsequent phases till the end of the simulation as if the other file was part of the including file.

9.2.2 Example

```
`include  "/design/library/cells.v"
`include  "/design/system/rest_of_system.v"
module  my_design(....);
....
endmodule
```

Example 9-1. ***`include compiler directive.***

In the above example, the library elements in file cells.v and the definition of rest of the system is included from other directories and files to add to the module being designed and tested.

9.2.3 Syntax

```
include_compiler_directive ::= `include "filename"
```

9.3 `define and `undef

9.3.1 Introduction

Macros or word replacements can be defined in the Verilog model using `define and `undef compiler directives.

9.3.2 Examples

```
//Define size of the word and use it for declarations
        `define WORDSIZE 64
        reg [WORDSIZE-1 : 0] data_bus;
```

```
//Define size of RAM and use both defines
`define RAMSIZE `hffffffff
reg [WORDSIZE -1 : 0] ram[RAMSIZE-l:0];


// Define a macro with parameters and use it
`define max(a,b) ((a) > (b) ? (a): (b))
.........
n = `max(p+q, r+s);

`undef RAMSIZE f
```

9.3.3 Syntax

```
text_macro_definition ::=
`define text_macro_name macro_text

text_macro_name ::=
text_macro_identifier[(list_of_formal_arguments)]

list_of_formal_arguments ::=
formal_arguments_identifier{,formal_arguments_identifier}


text_macro_usage ::=
`text_macro_identifier(    list_of_actual_arguments)

list_of_actual_arguments ::=
actual_argument{,actual_arguments}

actual_argument ::= expression

undefine_compiler_directive ::=
`undef text_macro_name
```

9.4 `ifdef,`else,`endif

9.4.1 Example

```
module and_op (a, b, c);
      output a;
      inout b, c;
      // view of design RTL/GATE is chosen using ifdef
`ifdef RTL
            wire a = b and c;
`else
and a1(a, b, c);
`endif
```

```
endmodule
```

Example 9-2. Compiling code conditionally based on prior macro definitions using `ifdef.

In the above example, the text-macro RTL must be defined prior to the module if RTL view is desired. The lines between `ifdef and `else [line starting with wire in this case] is selected and lines between `else and `endif are ignored [and line here].

9.4.2 Syntax

```
conditional_compiler_directive   ::=
        `ifdef text_macro_name
        first_group_of_lines
        [`else
        second_group_of_lines
        `endif]
```

9.5 `default_nettype

9.5.1 Example

```
`default_nettype  trireg
   module switch_sim(,,,);
          nmos(out0,  in1,  in2);
          nmos(out, in1, out0);              ..............
endmodule
```

Example 9-3. default_nettype compiler directive.

In the above example, all nets that are not declared are taken to be of type trireg. As this is defining a transistor-level module that is a capacitive network, this is appropriate.

9.5.2 Syntax

```
default_nettype net_type ::=
      `default_nettype  net_type

net_type:::=
      wire | tri | tri0 | wand | triand | tri1
      | wor | trior | trireg
```

9.6 `timescale

9.6.1 Example

```
`timescale 1ns/1ps
module test;
```

```
reg set;
parameter d = 1.55
initial
    begin
    #d set = 0;
    #d set = 1;
    end
endmodule
```

The `timescale tells the system to use 1 ns for all reporting and internally use 1 ps for resolution of time in this part of design. Thus, the value for parameter d is scaled to a delay of 1.55 ns. Had we used timescale directive `timescale 10ns/1ns, 1.55 would mean 15.5 or 16 ns. The first part of `time-scale gives the time-units and the second part gives the time-precision.

9.6.2 Syntax

```
time_scale_compiler_directive ::=
    `timescale time_unit/time_precision
```

The time unit and precision are defined using the following units:

s (seconds), **ms** (milliseconds), **us** (microseconds) **ns** (nanoseconds), **ps** (picoseconds)
fs (femtoseconds)

9.7 `resetall

This resets all compiler directives to default values. This is also a way to assure your part of design not to be affected by other designs combined with it during system simulation.

9.8 Exercises

1. The macro definitions with parameters are similar to function definitions. What is the advantage of using macros over functions?

2. Perform simulations of the example in section 7-1 of the structural and RTL descriptions by using `ifdef compiler directive. Add more directives so that one can simulate both together as well just by setting the right compiler directive.

10 INTERACTIVE SIMULATION AND DEBUGGING

10.1 Introduction

Verilog language was developed concurrently with a fully interactive simulator and debugger from the beginning. Thus, abilities to process the simulation in steps with trace and waveforms is available both in an interactive manner and also in a post-processing manner seamlessly through Verilog HDL. These facilities are provided in two ways:

a. by system tasks and functions and
b. interactive commands

10.2 System Tasks and Functions

10.2.1 Previously Covered

Some of these covered in Chapter 9 deal with tasks to perform:

- display to the output ($display)
- monitor the value-changes as textual output($monitor)
- send the display and monitor results to files ($fdisplay, $fmonitor etc.)
- stop or end the simulation in interactive mode - $stop or $finish
- generate data for waveform display - $dumpvars

These can also be used in interactive mode. In general, Verilog simulators should enable any behavioral statement to be entered on the command line in interactive simulation.

10.2.2 Additional Tasks

Some of the following are implementation dependent, but are present in several good implementations of Verilog:

$gr_waves
Like $monitor - except create waveforms

$showvars
Show the fanins of a signal.
In debugging the functioning of a multi-driver net, this is very helpful.

$keys file_name Save the keystrokes into file_name; this can then be rerun with this file.

$log file_name
Save the log into the log_file. This can be later analyzed for correctness.

10.3 Commands

Verilog simulator will accept the following commands in the interactive mode. Most simulators will provide Graphical User Interface with menu commands to cover these functions.

Single Step -';'
    This will advance the simulation one statement at a time. Ideally this should step through 1 event at a time.
Trace -','
    Display each update and evaluate events being executed by the simulator.
Break - CTRL-C
    Break the simulation and bring it in interactive mode.
sim n
    simulate for n time units
continue -.
    Continue simulation

10.4 Browser Tools

Most simulators will provide a browser that allows traversal of a design from top to bottom, and then in each part of the design get a list of signals defined there. These signals can then be selected for displaying values at command line or in the waveform window or for use with other tool within the tool-set.

10.5 Code Coverage

Several tools are available to check for code coverage of Verilog simulations. These will perform block-analysis and path-analysis apart from simple line by line analysis. This is useful for testing the adequacy of tests and is a good debugging tool.

1. Add code to check for V at various times at the output for different examples seen so far and then add $stop and debug the reasons for production of ‘x. Specially examples where different levels of abstractions are connected may produce x as transients and this should be confirmed that these indeed are expected values.
2. Several waveform tools perform data compression of vcd dump files. For larger designs, this is significant. Utilize these facilities if available in your toolset. Some software (from Veritools Inc., for example) is available on the internet on a trial basis.

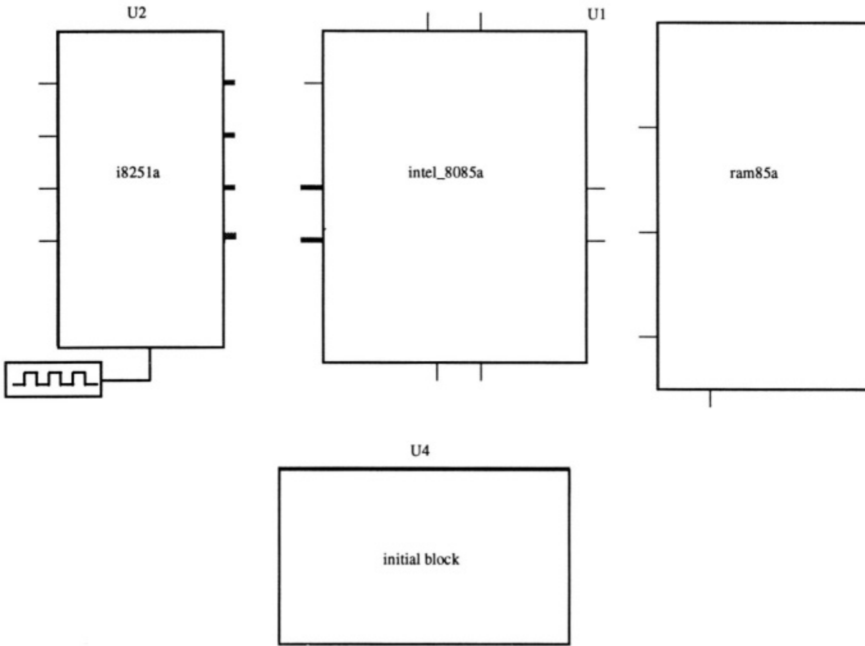
11 SYSTEM EXAMPLES

11.1 Introduction

In this chapter, we give system examples and discuss their modeling. The capabilities of Verilog HDL presented until now are well-suited to model systems where each of the system component is either behavioral or RTL or structural or external.

11.2 Example 1:8085 Based System: SioBS.V

The high level block diagram is shown on the next page. This consists of a model of Intel 8085 8-bit microprocessor, 8251 serial IO controller, a ram model and an initialization module. These are modeled at the behavioral level. The system module s85 instantiates the four modules—i8085a which is the microprocessor, i8251 which is the 8251 serial IO controller, the ram85a which is a ram model, and the initial block that initializes the system. The models for the 8085 and 8251 contain salient features of these model and the complete models can be simulated using the online model provided on the disk with the book.



All of these models are at the behavioral level.

Following are excerpts from the system with some explanations:

```
module s85; // system module
        reg [8:1]     dflags;   wire       s0, ale, rxd, txd;
        tri[7:0]      ad, a;
        tri1          read, write, iomout;
        reg           clock, trap, rst7p5, rst6p5, rst5p5,
             intr, ready, nreset, hold, pclock;
//The system has 3 main components as below
        ram85a        r0(ale,    ad, a, write, read, iomout);
        intel_8085a   i85(clock, , , , , trap, rst7p5, rst6p5, rst5p5, intr, ,
        ad, a, s0, ale, write, read,, iomout, ready, nreset,
         , , hold);
        i8251         p1(ad, rxd, , pclock, write, !iomout, a[0], read, ,
            , , , , txd, , !nreset, , , , pclock, );
        defparam p1.instance_id = 8'h01;
// The block below provides for initialization and resetting
        initial
          begin
            clock = 0;
            #500  nreset =  1;  ready = 1;
             @(posedge i85.haltff)  @i85.ec2;
               disable clockwave;    ......
          end
```

```
endmodule // s85
```

The i8085a module describes the instruction cycles for various instructions. Thus, any program in the assembly language (converted to hex or binary) can be run on this model, and various registers and nets can be observed with this. The ram loads these programs and connects with CPU to make this happen. A monitor and keyboard are connected via the serial interface providing additional input-output into the system.

The excerpts from the 8085 model are given below. These include descriptions of all blocks that form distinct modeling principles. For the instruction processing, a few examples are included here as the principles of modeling instructions are similar to all instructions but the actual code size can be quite large for the complete instructions set with all data types and all other variations. The entire model is included in the disk or CDROM included with the book.

```
/* Behavioral description of the Intel 8085a microprocessor */
module intel_8085a (clock, x2, resetff, sodff, sid, trap,
    rst7p5, rst6p5, rst5p5, intr, intaff, ad, a, s0, aleff,
      writeout, readout, s1, iomout, ready, nreset, clockff, hldaff, hold);
output  resetff, sodff, intaff, s0, aleff,
      writeout, readout, s1, iomout, clockff, hldaff;
inout[7:0] ad, a;
input  clock, x2, sid, trap, rst7p5, rst6p5, rst5p5,
      intr, ready, nreset, hold;
reg[15:0]       pc/*program counter*/, sp/*stack pointer*/,addr;
reg[8:0] intmask;  // interrupt mask and status
reg[7:0]       acc/*accumulator*/ regb/*general registers*/, regc,
      regd, rege, regh, regl,  ir /*instruction register*/, data/*data*/;
  reg  aleff, /* address latch enable*/    s0ff,    /*status line 0 */
   slff/*status line 1*/,....., cc/*carry condition code*/;

tri[7:0]  ad = dcontrol ? (s ? data: addr[7:0]): 'bz,
event      ec1 /*clock 1 event*/, ec2/*clock 2 event*/

// internal clock generation
always begin
   @(posedge clock) -> ec1;       @(posedge clock) -> ec2;
end
integer instruction; // instruction count

always  begin:run_processor
   #1 reset_sequence;
   fork
      execute_instructions;             // Instructions executed
      wait(!nreset)                  // in parallel with reset
         @ec2 disable run_processor; // control. Reset will
   join                              // disable run_processor
end                                   // and all tasks
```

```
task  reset_sequence;
begin
   wait(!nreset)
   fork
      begin
         $display("Performing 8085(%m) reset sequence");
         read = 1;
            ...........
         disable check_reset;
      end
      begin:check_reset
         wait(nreset)              // Check, in parallel with the
            disable run_processor; // reset sequence, that nreset
      end                          // remains at 0.
   join
   wait(nreset)  @ecl  @ec2 resetff = 0;
end
endtask

/* fetch and execute instructions */
task execute_instructions;
forever begin
   instruction = instruction + 1;
   @ecl /*clock cycle 1 */ addr = pc;  s = 0;  iomff = 0;
         ......
   @ec2       aleff = 0;
   @ec1 /*clock cycle 2*/ read = 0;   dcontrol = 0;
   @ec2           ready_hold;
   @ec2 /*clock cycle 3*/  read = 1; data = ad; ir = ad;
   @ec1 /*clock cycle 4*/

      if(do6cycles(ir)) begin
         // do a 6-cycle instruction fetch
         @ecl  @ec2 // conditional clock cycle 5
            if(hold) begin
                  ...
               dcontrol = 0;     @ec2 hldaff = 1;
            end
            else begin
                  ......
            end
         @ec1; // conditional clock cycle 6
      end

      if(holdff)holdit;
      checkint;
      do_instruction;
         ..........
      if(validint) interrupt;
end
```

```
endtask

function do6cycles;
      ....
endfunction

task  checkint;
begin
   if(rst6p5)
      if((intmask[3] == 1) && (intmask[l] == 0)) intmask[6] = 1;
   else
      intmask[6] = 0;
      ............................

   if({intmask[7], intmask[3:2]} == 6)
      intmask[4] = 1;
   else
      intmask[4] = 0;

   validint = (intmask[6:4] == 7) | trapff | intr;
end
endtask

/* memory read */
task memread;
output[7:0]  rdata;
input[15:0] raddr;
begin
    @ecl
      addr = raddr;
      aleff = 1;
         ............
    @ec2
      aleff =  0;
    @ecl
      dcontrol = 0;
      if(int)
         intaff =  0;
      else
         read = 0;
    @ec2
      ready_hold;
      checkint;
    @ec2
      intaff =  1;
      read = 1;
      rdata = ad;
   if(holdff) holdit;
end
endtask
```

```
task memwrite;
..............
endtask

task ioread;
 ..........
endtask

task iowrite;
..........
endtask

taskdo_instruction;
begin
   pc = pc + 1;
   @ec2 // instruction decode synchronized with clock 2 event
      case(ir[7:6])
         0:
            case(ir[2:0])
                0: newops;
                1: if(ir[3])addhl;elselrpi;
                2: sta_lda;
                3: inx_dcx;
                4: inr;
                5: dcr;
                6: movi;
                7: racc_spec;
            endcase
         1:
            move;
         2:
            rmop;
         3:
            case(ir[2:0])
                0,
                2,
                4: condjcr;
                1: if(ir[3]) decode1; else pop;
                3: decode2;
                5: if(ir[3]) decode3; else push;
                6: immacc;
                7: restart;
            endcase
      endcase
end
endtask

/* decrement register and memory contents */
task dcr;
```

```
    case(ir[5:3])
        0: dodec(regb); // DCR B
        1: dodec(regc); // DCR C
            .................
        6: // DCR M
            begin
                memread(data, {regh, regl});
                dodec(data);
                memwrite(data, {regh, regl});
            end

        7:dodec(acc); // DCR A
    endcase
endtask

/* enabled only from decrm */
task dodec;
inout[7:0] sr;
begin
    cac = sr[3:0] == 0;
    sr = sr -1;
    calpsz(sr);
end
endtask
  .................
 .....................
 endmodule
```

Example 11-1. 8085 microprocessor, ram, and 8251 serial IO controller system – behavioral model.

11.3 Example 2: R4200

```
/***************************************************************************
This is the top level module which connects the processor and the
system controller together and also provides the stimulus to test
the chips.

***************************************************************************/

module stim;

    wire [63:0] SysAD;
    wire [63:0] CS_AD;
    wire [7:0] SysADC;
    wire [7:0] CS_ADC;
    wire [8:0] SysCmd;
    wire [8:0] CS_cmd;
```

```
	wire SysCmdP,CS_cmd_P,ext_ready;
	wire ValidOut,Release,SC_req_valid,CS_req_valid;

	reg  [8:0]  syscmd;
	reg [63:0] sysad;

	reg ValidIn,ExtRqst,RdRdy,WrRdy,MasterClock,Reset;
	reg read,respond_toread,send_address,send_data;


	  always #10  MasterClock = ~MasterClock;

	  assign SysAD = ValidIn? 64'bz:sysad;
	  assign SysCmd  =  ValidIn?  8'bz:syscmd;

	  processor
p1(CS_cmd,CS_AD,CS_cmd_P,CS_ADC,CS_req_valid,SC_req_valid,MasterClock,ext_ready
,Reset);

	  sys_cntrl
s1(SysAD,SysADC,SysCmd,SysCmdP,ValidIn,ValidOut,ExtRqst,Release,RdRdy,
	     WrRdy,MasterClock,Reset,CS_req_valid,CS_AD,CS_ADC,CS_cmd,CS_cmd_P,
SC_req_valid,ext_ready);


	  initial  begin
	    ValidIn =  1;
	    ExtRqst =  1;
	    RdRdy = 0;
	    send_address=0;
	    send_data=0;
	    WrRdy = 0;
	    read  =  0;
	    respond_toread = 0;
	    MasterClock  =  0;
	    Reset = 0;
	    #3  Reset =  1;

	    $monitor("Time=%0d,SysAD  =

%x,SysADC=%x,SysCmd=%x,ValidIn=%x,ValidOut=%x,ExtRqst=%x,Release=%x,RdRdy=
%x,WrRdy=%b,Reset=%b,
	  CS_req_valid=%b,CS_AD=%x,CS_ADC=%x,CS_cmd=%x,

SC_req_valid=%b,ext_ready=%b\n",$time,SysAD,SysADC,SysCmd,ValidIn,ValidOut,ExtRq
st,Release,RdRdy,WrRdy,Reset,
	  CS_req_valid,CS_AD,CS_ADC,CS_cmd, SC_req_valid,ext_ready);
	  #700  ExtRqst  =  0;
	  WrRdy=1; RdRdy=1;
	    end
```

```
/* responding to a read request from the processor */
always @ (posedge MasterClock) begin
   if(~ValidIn && respond_toread) begin ValidIn=1; respond_toread = 0;end
   if (respond_toread) begin
                  sysad =255;
                  syscmd=257;
                  ValidIn = 0;
                end
   if (~Release && SysCmd==1) begin read = 0;respond_toread = 1;end
            end

/* Setting and writing an external write request. */

always @ (posedge MasterClock) begin
   if(~ValidIn  && send_data) begin ValidIn=1; RdRdy=0; WrRdy=0; send_data=0;end
        if (send_data) begin
                  sysad =511;
                      syscmd=2;
                      ValidIn = 0;
            end
        if(send_address) begin send_address=0;
                  sysad =0;
                      syscmd=2;
                  send_data = 1;
                      ValidIn = 0;
                    end

        if (!ExtRqst) begin
            if (~Release) begin
                  ExtRqst=1;
                  send_address=1;
                       end
                  end
                  end

endmodule



/*************************************************************************/

     /* Definition of the i/o ports of R4200 chip and implementation of the  bus interface
protocol shown in fig 1.54 and 1.55 .
     */

   /*******************************************************************
```

```
This is the system interface module. It implements the bus interface protocol and
interacts
with the processor chip.

        CS: - Cache to the System Controller.

        CS_req_valid - A valid request is in the CS_AD bus and CS_ADC command bus
                andCS_cmd_P.

        SC_req_valid - A valid request is in the CS_AD bus and CS_ADC command bus
              and CS_cmd_P and System interface is driving it.

        CS_cmd[8] - 0=Address cycle 1= Data cycle.
        CS_cmd[7:0] - 1 Read, 2 Write.
        SysCmd[7:0] -1 Read response, 2 Write request. 0 - Null

        state: Indicates the state of the system controller.
        ext_ready: Signalling the cpu_core that the external agent is ready to accept a
request.

        The rest of the inputs and outputs are the same a give in Table 1-5.

    ********************************************************************/

    module
sys_cntrl(SysAD,SysADC,SysCmd,SysCmdP,ValidIn,ValidOut,ExtRqst,Release,RdRdy,
        WrRdy,MasterClock,Reset, CS_req_valid,CS_AD,CS_ADC,CS_cmd,CS_cmd_P,
                SC_req_valid,ext_ready);

    inout [63:0] SysAD;
    inout [63:0] CS_AD;
    inout  [7:0]  SysADC;
    inout  [7:0]  CS_ADC;
    inout [8:0] SysCmd;
    inout [8:0] CS_cmd;
    inout SysCmdP;
    inout CS_cmd_P;
    input ValidIn,ExtRqst,RdRdy,WrRdy,MasterClock,Reset,CS_req_valid;
    output ValidOut,Release,SC_req_valid,ext_ready;

    wire  [63:0] SysAD;
    wire  [63:0] CS_AD;
    wire  [7:0]  SysADC;
    wire  [7:0] CS_ADC;
    wire  [8:0] SysCmd;
    wire  [8:0] CS_cmd;
    wire SysCmdP;
    wire CS_cmd_P;

    /*********************************STATES***************/
```

```
`define idle 4'h0
`define send_rd_add 4'h1
`define wait_rd_data 4'h2
`define send_wr_add 4'h3
`define send_wr_data 4'h4
`define ext_rd_response 4'h5
`define ext_wr_add 4'h6
`define ext_wr_data 4'h7
`define ext_null_req 4'h8
`define ext_req_asserted 4'h9

      reg  [3:0]  state;
reg [1:0] ready_for_read;
reg  [1:0] ready_for_write;
reg rd_pending,ext_ready;

assign    SysAD = (state= =`send_rd_add || state= =`send_wr_add ||
state= =`send_wr_data)?   CS_AD:
    64'bz;
    assign SysADC =  (state= =`send_rd_add  ||  state= =`send_wr_add  ||
state= =`send_wr_data)? CS_ADC:
    8'bz;
    assign  SysCmd =  (state= =`send_rd_add  ||  state= =`send_wr_add  ||
state= =`send_wr_data)?  CS_cmd:
    9'bz;
    assign SysCmdP = (state= =`send_rd_add || state= =`send_wr_add ||
state= =`send_wr_data)? CS_cmd_P
    :  1'bz;
    assign CS_AD = (state= =`ext_rd_response || state= =`ext_wr_add ||
state= =`ext_wr_data)?  SysAD:
    64'bz;
    assign CS_ADC =  (state= =`ext_rd_response  ||  state= =`ext_wr_add ||
state= =`ext_wr_data)?   SysADC:
    8'bz;
    assign CS_cmd = (state= =`ext_rd_response || state= =`ext_wr_add ||
state= =`ext_wr_data)? SysCmd:
    9'bz;
    assign CS_cmd_P =  (state= =`ext_rd_response  ||  state= =`ext_wr_add  ||
state= =`ext_wr_data)?
              SysCmdP:  1'bz;
    assign ValidOut = (state= =`send_rd_add || state= =`send_wr_add ||
state= =`send_wr_data)?  0:1;
    assign Release = (state= =`send_rd_add || state= =`ext_req_asserted)? 0:  1;
    assign SC_req_valid = (state= =`ext_rd_response || state= =`ext_wr_add || state = =
`ext_wr_data)?
    1:0;


initial begin
    state =  0;
```

```
    ready_for_read =  0;
    ready_for_write =  0;
    rd_pending=0;
    ext_ready = 0;
    end


  always  @(posedge MasterClock) begin

  // giving two cycle delay for the external interface to accept read and write requests
  // after asserting acceptance

    if (RdRdy) begin ready_for_read = 0; end
    else if (ready_for_read<2) ready_for_read= ready_for_read +1;

    if (WrRdy) begin ready_for_write = 0; end
    else if (ready_for_write<2) ready_for_write=ready_for_write+1;

    if (ready_for_read = = 2 && ready_for_write = = 2) begin ext_ready=1; end
    else ext_ready=0;

  end

  always  @(posedge MasterClock) begin

    case(state)
        `idle: begin
            if (CS_req_valid) begin
                if(CS_cmd= =1  && ready_for_read= =2) state=`send_rd_add;
                else  if(CS_cmd= 2  &&  ready_for_write= 2)  state  =  `send_wr_add;
                    end
            else if(!ExtRqst) state = `ext_req_asserted;
            end
        `send_rd_add: begin
                $display("!!!  SYS_CNTRL:  Sending rd address \n");
                state  =  `wait_rd_data;
                end
        `wait_rd_data: begin
                $display("!!!  SYS_CNTRL:  Waiting for data from external
interface\n");
                if(!ExtRqst)  begin
                    state = `ext_req_asserted;
                    rd_pending=1;
                        end
                else if(!ValidIn) begin
                $display("!!!  SYS_CNTRL:  External  interface  returning  data\n");
                    state = `ext_rd_response;
                        end
                end
        `send_wr_add: begin
```

```
                        state = `send_wr_data;
                        end
                `send_wr_data: begin
                        if (CS_req_valid && CS_cmd= =2)  state=`send_wr_add;
                        else state=`idle;
                        end
                `ext_rd_response: begin
                $display("!!! SYS_CNTRL: External interface going back to idle after
rdresponse\n");
                                state = `idle;
                        end
                `ext_wr_add: begin
                        $display("!!! SYS_CNTRL: External agent sending wr address\n");
                        state = `ext_wr_data;
                        end
                `ext_wr_data: begin
                        $display("!!! SYS_CNTRL: External agent sending wr data\n");
                        if (rd_pending) state = `wait_rd_data;
                        else state = `idle;
                        end
                `ext_null_req:  begin
                        if (rd_pending)  state = `wait_rd_data;
                        else state = `idle;
                        end
                `ext_req_asserted: begin
                        $display("!!!  SYS.CNTRL:  External  agent  asserted  request\n");
                        if (!ValidIn && SysCmd = =  1)  state=`ext_rd_response;
                        else if(!ValidIn && SysCmd = = 2) state = `ext_wr_add;
                        else if(!ValidIn && SysCmd = = 0) state = `ext_null_req;
                        end
                default: state = 0;

        endcase
    end

  endmodule


/***********************************************************************

      This is the processor  module. This module has the instruction and data caches inbuilt.
When a Id/st instruction is issued, it checks if the location is present in the cache, if it is
present in the cache then, load the data immediately else request it from the external memory,
through the external agent.

      The processor module reads the instructions from a file called test.mem and executes
them sequentially.


***********************************************************************/
```

```
module processor
(Dcmd,Dbus,DcmdP,DbusC,CS_req_valid,req_valid_in,MasterClock,ext_ready,Reset);

inout [8:0] Dcmd;
inout [7:0] DbusC;
inout [63:0] Dbus;
inout DcmdP;
output CS_req_valid;
inputreq_valid_in,Reset;
input MasterClock,ext_ready;

reg req_valid_out;
wire [63:0] Dbus;
wire [8:0] Dcmd;
wire CS_req_valid;

/******* Define opcodes for instructions *****************/
/** R-Format Insts **/
`define ADD 6'b100000
`define ADDU 6'b100001
`define AND 6'b100100
`define OR 6'b100101
`define SLL 6'b000000
`define SLLV 6'b000100
`define SRA 6'b000011
`define SRAV 6'b000111
`define SRL 6'b000010
`define SRLV 6'b000110
`define SLT 6'bl01010
`define SLTU 6'bl01011
`define SUB 6'b100010
`define SUBU 6'b100011
`define XOR 6'bl00110
`define NOR 6'b100111
`define MFHI 6'b010000
`define MFLO 6'b010010
`define MTHI 6'b010001
`define MTLO 6'b010011
/**/
/** I-Format Insts **/
`define ADDI 6'b001000
`define ADDIU 6'b001001
`define ANDI 6'b001100
`define ORI 6'b001101
`define XORI 6'b001110
`define SLTI 6'b001010
`define SLTIU 6'b001011
`define LUI 6'b001111
`define LB 6'b100000
```

```
`define LBU     6'b100100
`define LH      6'b100001
`define LHU     6'b100101
`define LW      6'b100011
`define LWL     6'b100010
`define LWR     6'b100110
`define SB      6'b101000
`define SH  6'b101001
`define SW      6'b101011
`define SWL  6'b101010
`define SWR     6'b101110
/**/
/** Multiply and divide **/
`define DIV     6'b011010
`define DIVU     6'b011011
`define MULT     6'b011000
`define MULTU     6'b011001
/**/
`define HALT     6'b000001
/**/
/** Jumps and Branches **/
`define BEQ     6'b000100
`define BNE     6'b000101
`define BLEZ     6'b000110
`define BGTZ     6'b000111
`define BLTZ     5'b00000
`define BGEZ     5'b00001
`define BLTZAL   5'b10000
`define BGEZAL   5'b10001
`define J       6'b000010
`define JAL     6'b000011
`define JR      6'b001000
`define JALR     6'b001001

`define fetch       0
`define decode      1
`define execute     2
`define br_exe      3
`define mem_fetch 4
`define complete    5

reg [7:0] data_cache[0:8191]; /*  8kbyte data cache */
reg [31:0] inst_cache[0:4095]; /* 16kbyte inst cache */

reg [50:0] data_tag[0:8191]; /* 8kbyte data tag */
reg data_valid[0:8191];  /*  8kbyte data cache valid bit.*/
                 /* Indicates if the data is available */


reg [31:0] inst_buffer; /* inst currently executed */
```

```
reg [63:0]  FX_GPR[0:31],FP_GPR[0:31],MULT_HI,MULT_LO,PC;
reg [31:0] FCR[0:31];
reg LL_bit,cmdP,rd_data_done,rd_pending;
reg  [7:0]  cmdC;

reg  [63:0]  s1_contents,s2_contents,result,eff_address,rd_data;
reg[2:0]   cpu_state;
reg[4:0]   sa,rs,rt,rd,dreg;
reg[15:0] imm;
reg[25:0]  target;
reg[8:0] cmd;
reg[63:0] ext_register;
integer  opcode,opcode1  ,opcode2,line_valid,send_cmd;

integer  k;
initial begin
        for  (k=0;k<32;k=k+1)  begin
                FX_GPR[k] =0;
        end
        for (k=0;k<8191;k=k+1) data_valid[k]=0;
        PC=0;
        cpu_state = 0;
        rd_pending=0;
        req_valid_out=0;
        rd_data_done=0;
        send_cmd=0;
        cmdC=0;
        $readmemh("test.mem",inst_cache);

       end

    assign Dbus = req_valid_out ? eff_address:64'bz;
    assign Dcmd =  req_valid_out  ? cmd:9'bz;
    assign DcmdP = req_valid_out ? cmdP:1'bz;
    assign DbusC = req_valid_out ? cmdC:8'bz;
    assign CS_req_valid = req_valid_out;

    always  @(!Reset)begin
            cpu_state = 0;
            end
    always @(req_valid_in)
       begin
       if (rd_pending) begin
            if(Dcmd==257)begin
            rd_data = Dbus;
            rd_data_done = 1;
                    end
            else if(Dcmd==258) ext_register = Dbus;
```

```
                end
        end
      always  @ (posedge MasterClock)
           begin
             if (Reset) begin
             case(cpu_state)
                  `fetch:     fetch_inst;
                  `decode: decode_inst;
                  `execute: execute_inst;
                  `br_exe:  exe_br_inst;
                  `mem_fetch:  exe_mem_inst;
                  `complete: complete_inst;
                  default: cpu_state=0;
             endcase
                    end
        end


    task fetch_inst;
        begin
    $display("**********************FETCH**************************\n");
        $display("!!!!!!PC=%0d,Inst = %x\n",PC,inst_cache[PC]);
        inst_buffer = inst_cache[PC];
        cpu_state = `decode;
        end
    endtask

    task  decode_inst;
        begin
             $display("----------------------DECODE---------------------- \n");
             opcode = inst_buffer[31:26];
             rs = inst_buffer[25:21];
             rt = inst_buffer[20:16];
             imm = inst_buffer[15:0];
             target =  inst_buffer[25:0];
             sa =  inst_buffer[10:6];
             rd = inst_buffer[15:11];
             opcode1 = inst_buffer[5:0];
             opcode2 = rt;
             s1_contents = FX_GPR[rs];
             s2_contents = FX_GPR[rt];

             /*  Checking for branches in opcodes   */
                  if (opcode =`BEQ || opcode=`BNE || opcode=`BLEZ ||
opcode=`BGTZ ||
                  opcode=`J || opcode=`JAL)
                  begin
                  cpu_state = `br_exe;
                  end
```

```
            else if((opcode ==1)&& (rt==`BLTZ || rt==`BGEZ || rt==`BLTZAL ||
rt==`BGEZAL))
                begin           /* Checking for branches in BCOND codes */
                cpu_state = `br_exe;
                opcode = rt;
                end
            else if((opcode == 0) && (opcodel ==`JR || opcode1 ==`JALR))
                begin
                cpu_state = `br_exe;
                opcode=opcode1;
                end
                else
                begin
                cpu_state = `execute;
                end

        case (inst_buffer[31:26])
            `HALT:begin
                print;
                $finish;
                end
            /*0:   * Special*
                case(inst_buffer[5:0])

                    `SLL,`SRL,`SRA,`SLLV,`SRLV,`SRAV,
                    `ADD,`ADDU,`AND, `SUB,`SUBU,`OR,
                    `SLT,`SLTU,`XOR,`NOR,`MFHI,
                    `MFLO, `MTLO, `MTHI, `MULT,`MULTU,
                    `DIV,`DIVU:
                            //opcode = inst_buffer[5:0];
                endcase*/
        endcase


        end
    endtask

    task execute_inst;
    integer j,signs;
    reg[127:0] mult_result;
        begin

    $display("~~~~~~~~~~~~~~~~~~~~~~~EXECUTE~~~~~~~~~~~~~~~~~~~~\n");
            cpu_state = `complete;
        case(opcode)
        0:  /* Special */
            case(opcode1)
            `SLL: result=s2_contents<<sa;
            `SLLV: result=s2_contents<<s1_contents;
            `SRL: result=s2_contents>>sa;
```

```
`SRLV:result=s2_contents>>s1_contents;
 `SRA:
        begin
               result=s2_contents;
               for(j=0;j<sa;j=j+1)
                      begin
                      result=result>>1;
                     result={s2_contents[63],result[62:0]};
                      end
        end
`SRAV:
        begin
               result=s2_contents;
              for(j=0;j<s1_contents;j=j+1)
                      begin
                  result=result>>1;
                 result={s2_contents[63],result[62:0]};
                  end
    end
`ADD:
                   begin
                   signs={s1_contents[63],s2_contents[63]};
                   case(signs)
                          0: result=s1_contents+s2_contents;

                          1: begin
                                 s2_contents=0-s2_contents;
                                 result=s1_contents-s2_contents;
                             end
                          2: begin
                                 s1_contents=0-s1_contents;
                                 result=s2_contents-s1_contents;
                             end
                          3: begin
                                 s1_contents=0-s1_contents;
                                 s2_contents=0-s2_contents;
                                 result=s2_contents+s1_contents;
                                 result=0-result;
                             end
                   endcase
             end
`ADDU:  begin
             result =  s1_contents  + s2_contents;
             $display("********ExecutingADDU**********\n");
      end
`SUB,`SUBU:
                   begin
             $display("********ExecutingSUBU**********\n");
                   signs={s1_contents[63],s2_contents[63]};
                   case(signs)
```

```
                0: result=s1_contents-s2_contents;
                1: begin
                      s2_contents=0-s2_contents;
                      result=s1_contents+s2_contents;
                   end
                2: begin
                      s1_contents=0-sl_contents;
                      result=s2_contents+s1_contents;
                      result=0-result;
                   end
                3: begin
                      s2_contents=0-s2_contents;
                      result=s2_contents-s1_contents;
                   end
            endcase
            end
`SLTU:
        begin
        $display("********Executing SLTU**********\n");
            if(s1_contents<s2_contents)
            result=1;
            else
            result=0;
        end
`SLTIU:
        begin
        $display("********Executing SLTIU**********\n");
            s2_contents = {{48{imm[15]}},imm[15:0]};
            rd = rt;
            if(s1_contents<s2_contents)
            result=1;
            else
            result=0;
        end
`SLT:
        begin
        $display("********Executing SLT**********\n");
          signs={s1_contents[63],s2_contents[63]};
          case(signs)
                0:begin
                      if(s1_contents<s2_contents) result=1;
                      else result=0;
                  end
                1:result=0;
                2: result=1;
                3:begin
                      if(s1_contents<s2_contents) result=0;
                      else result=1;
                  end
          endcase
```

```
                end
    `SLTI:
                begin
                $display("********ExecutingSLTI**********\n");
                 rd = rt;
                 s2_contents =  {{48{imm[15]}},imm[15:0]};
                 signs={  s1_contents[63],s2_contents[63]};
                 case(signs)
                         0:begin
                              if(s1_contents<s2_contents) result=1;
                              else  result=0;
                           end
                         1: result=0;
                         2: result=1;
                         3:begin
                              if(s1_contents<s2_contents)result=0;
                              else result=1;
                           end
                 endcase
                 end
    `AND: result=s1_contents&s2_contents;
    `OR  : result=s1_contentsls2_contents;
    `XOR  : result=s1_contentsAs2_contents;
    `NOR:   begin
                result=s1_contentsls2_contents;
                result=~result;
            end
  `MULTU:
          begin
          mult_result=s1_contents *s2_contents;
          MULT_HI = mult_result[127:64];
          MULT_LO = mult_result[63:0];
          end
  `MULT: begin
              signs={s1_contents[63],s2_contents[63]};
              case(signs)
                    0:    mult_result=s1_contents*s1_contents;

                     1: begin
                           s2_contents=0-s2_contents;
                           mult_result=s1_contents*s2_contents;
                           mult_result=0-mult_result;
                        end

                    2: begin
                           s1_contents=0-s1_contents;
                           mult_result=s1_contents*s2_contents;
                           mult_result=0-mult_result;
                        end
                    3: begin
```

```
                    s1_contents=0-s1_contents;
                    s2_contents=0-s2_contents;
                    mult_result=s1_contents*s2_contents;
                    mult_result=0-mult_result;
                  end
              endcase
          MULT_HI = mult_result[127:64];
          MULT_LO = mult_result[63:0];
          end

    `DIVU: result=s1_contents/s2_contents;
    `DIV: begin
              signs={s1_contents[63],s2_contents[63]};
              case(signs)
                 0: result=s1_contents/s2_contents;
                 1: begin
                    s2_contents=0-s2_contents;
                    result=s1_contents/s2_contents;
                    result=0-result;
                  end

                 2: begin
                    s1_contents=0-sl_contents;
                    result=s1_contents/s2_contents;
                    result=0-result;
                  end
                 3: begin
                    s1_contents=0-s1_contents;
                    s2_contents=0-s2_contents;
                    result=s1_contents/s2_contents;
                    result=0-result;
                  end
              endcase
          end
        endcase
     `ADDI:
                  begin
                  rd = rt;
                  s2_contents = {{48{imm[15]}},imm[15:0]};
                  signs={s1_contents[63],s2_contents[63]};
                  case(signs)
                     0: result=s1_contents+s2_contents;

                     1: begin
                        s2_contents=0-s2_contents;
                        result=s1_contents-s2_contents;
                      end
                     2: begin
                        s1_contents=0-s1_contents;
                        result=s2_contents-s1_contents;
```

```
                                end
                              3: begin
                                   s1_contents=0-s1_contents;
                                   s2_contents=0-s2_contents;
                                   result=s2_contents+s1_contents;
                                   result=0-result;
                                end
                          endcase
                      end

           `ADDIU:        begin
                          rd = rt;
                          s2_contents = {{48{imm[15]}},imm[5:0]};
                          result = s1_contents + s2_contents;
                      $display("********ExecutingADDIU**********\n");
                      end
           `ANDI:
                      begin
                      rd = rt;
                      s2_contents = {{48{imm[15]}},imm[15:0]};
                      result=s1_contents&s2_contents;
                      end
           `ORI  :    begin
                      $display("********ExecutingORI**********\n");
                      rd = rt;
                      s2_contents = {{48{imm[15]}},imm[15:0]};
                      result=s1_contentsls2_contents;
                 end
           `XORI  :   begin
                      rd = rt;
                      s2_contents = {{48{imm[15]}},imm[5:0]};
                      result=s1_contents^s2_contents;
                      end

           `LUI:        result=s2_contents<<16;

         `LB,`LBU,`LH,`LHU,`SW,
         `SW,`SH,`SB,`LW:
                          begin
                          cpu_state = `mem_fetch;
                          rd = rt;
                        eff_address = s1_contents + {{48{imm[15]}},imm[15:0]};
                          end

      endcase
      end

   endtask

   task  complete_inst;
```

```
        begin
    $display("~~~~~~~~~~~~~~~~~~~~~~~~~~COMPLETE~~~~~~~~~~~~~~~~~~~~~~\n");
            PC=PC+1;
            case(opcode)
                    `MULT,`MULTU: cpu_state = `fetch;
                    default:
                        begin
                        FX_GPR[rd] = result;
                        cpu_state = `fetch;
                        end
            endcase
        end
    endtask

    task exe_br_inst;

    integer j;
    reg[63:0] target_address;

    begin
        $display("-----------------------BRANCH EXECUTE------------------\n");
        case(opcode)
            `BEQ,`BNE,`BLEZ,`BGTZ,
              `BLTZ,`BGEZ,`BLTZAL,`BGEZAL:
                    begin
                     target_address = {{48{imm[15]}},imm[15:0]};
                    end

              `J,`JAL    :
                    begin
                        target_address = {{38{target[25]}},target[25:0]};
                    end
           endcase

        case(opcode)
            `BEQ:
                begin
            $display("***** Executing BEQ *****\n");
                if (s1_contents==s2_contents)
                    begin
                    if(target_address[63]=1)
                    begin
                    target_address=0-target_address;
                    target_address=target_address/4;
                    PC=PC-target_address;
                    end
                    else
                        begin
                        target_address=target_address/4;
```

```
                        PC=PC+target_address;
                        end
                    end
                else PC=PC+1;
            end
        `BGEZ,`BGEZAL:
            begin
                if (s1_contents>=0)
                    begin
                    if(target_address[63]==1)
                    begin
                    target_address=0-target_address;
                    target_address=target_address/4;
                    PC=PC-target_address;
                    end
                    else
                        begin
                        target_address=target_address/4;
                        PC=PC+target_address;
                        end
                    end
                else  PC=PC+1;
            end
        `BGTZ:
            begin
                if (s1_contents>0)
                    begin
                    if(target_address[63]==1)
                    begin
                    target_address=0-target_address;
                    target_address=target_address/4;
                    PC=PC-target_address;
                    end
                    else
                        begin
                        target_address=target_address/4;
                        PC=PC+target_address;
                        end
                    end
                else  PC=PC+1;
            end
        `BLEZ:
            begin
                if  (s1_contents<=0)
                    begin
                    if(target_address[63]==1)
                    begin
                    target_address=0-target_address;
                    target_address=target_address/4;
                    PC=PC-target_address;
```

```
                end
                else
                    begin
                    target_address=target_address/4;
                    PC=PC+target_address;
                    end
                end
            else  PC=PC+1;
        end
    `BLTZ,`BLTZAL:
        begin
            if  (s1_contents<0)
                begin
                if(target_address [63]==1)
                begin
                target_address=0-target_address;
                target_address=target_address/4;
                PC=PC-target_address;
                end
                else
                    begin
                    target_address=target_address/4;
                    PC=PC+target_address;
                    end
                end
            else  PC=PC+1;
        end
    `BNE:
        begin
            if (s1_contents!=s2_contents)
                begin
                if(target_address[63]==1)
                begin
                target_address=0-target_address;
                target_address=target_address/4;
                PC=PC-target_address;
                end
                else
                    begin
                    target_address=target_address/4;
                    PC=PC+target_address;
                    end
                end
            else PC=PC+1;
        end
    `J:  PC=target_address/4;
    `JAL: begin
        $display("***** Executing JAL *****\n");
        FX_GPR[31]=PC+1;
        PC=target_address/4;
```

```
                end
          `JR,`JALR: begin PC=s1_contents;
                $display("***** Executing JR *****\n"); end
          endcase
          cpu_state=`fetch;
     end
     endtask
```

/* If the request is not pending already, then check if the effective address is present in the cache, else make a read request to the external agent to bring the data in. After the data is brought in, if the request is a load request, read it, else if it is a store then store it in the cache and also write it back to the external agent. */

```
     task exe_mem_inst;

     begin
          $display("-------------------COMPLETING LD/STINST----------------\n");

          if(send_cmd)begin
                req_valid_out=0; send_cmd=0; end

          if (rd_pending) begin
                $display("!!!!!!Waiting for the system controller to return data...\n");
                      send_cmd=1;
                      end

          if (rd_pending=0)
                      begin


//$display("data_valid=%x,data_tag=%x\n",data_valid[eff_address[12:0]],data_tag[eff_addres
s[12:0]]);
                            if(~data_valid[eff_address[12:0]] II
                                  (data_tag[eff_address[12:0]] != eff_address[50:0]))
                                  begin
                $display("!!!!Data not valid in the cache... Getting it from the main
memory...\n");
                                  if (ext_ready) begin
                                             cmd=1;
                                             cmdP=0;
                                             rd_pending=1;
                                             req_valid_out=1;
                                             end
                                  end
                      else begin line_valid=1; end
                      end
          else if(rd_pending && rd_data_done) begin
                $display("!!!!!!Received  data from the system controller....\n");
                            rd_pending = 0;
```

```
                rd_data_done = 0;
                line_valid=1;
                data_cache[eff_address[12:0]] = rd_data[7:0];
                data_valid[eff_address[12:0]] = 1;
                data_tag[eff_address[12:0]] = eff_address[63:13];
                data_cache[eff_address[12:0]+l] = rd_data[15:8];
                data_valid[eff_address[12:0]+l] = 1;
                data_tag[eff_address[12:0]+l] = eff_address[63:13];
                data_cache[eff_address[12:0]+2] = rd_data[23:16];
                data_valid[eff_address[12:0]+2] = 1;
                data_tag[eff_address[12:0]+2] = eff_address[63:13];
                data_cache[eff_address[12:0]+3] = rd_data[31:24];
                data_valid[eff_address[12:0]+3] = 1;
                data_tag[eff_address[12:0]+3] = eff_address[63:13];
                data_cache[eff_address[12:0]+4] = rd_data[39:32];
                data_valid[eff_address[12:0]+4] = 1;
                data_tag[eff_address[12:0]44] = eff_address[63:13];
                data_cache[eff_address[12:0]+5] = rd_data[47:40];
                data_valid[eff_address[12:0]+5] = 1;
                data_tag[eff_address[12:0]+5] = eff_address[63:13];
                data_cache[eff_address[12:0]+6] = rd_data[55:48];
                data_validteff_address[12:0]+6] = 1;
                data_tag[eff_address[12:0]+6] = eff_address[63:13];
                data_cache[eff_address[12:0]+7] = rd_data[63:56];
                data_valid[eff_address[12:0]+7] = 1;
                data_tag[eff_address[12:0]+7] = eff_address[63:13];
                    end
if (line_valid) begin
    line_valid = 0;
    case(opcode)
        `LW: begin
                result[63:32]=0;
                result[31:24]=data_cache[eff_address+3];
                result[23:16]=data_cache[eff_address+2];
                result[15:8]=data_cache[eff_address+l];
                result[7:0]=data_cache[eff_address];
                FX_GPR[rd] = result;
            end
        `LB: begin
                result[7:0]=data_cache[eff_address];
                result[63:8]={56{result[7]}};
            end
        `LBU: begin
                result[7:0]=data_cache[eff_address];
                result[63:8]=0;
                FX_GPR[rd] = result;
            end
        `LH: begin
                result[7:0]=data_cache[eff_address+l];
                result[15:8]=data_cache[eff_address];
```

```
                    result[63:16]={48{result[15]}};
                    FX_GPR[rd] = result;
                end
            `LHU: begin
                    result[7:0]=data_cache[eff_address+l];
                    result[15:8]=data_cache[eff_address];
                    result[63:16]=0;
                    FX_GPR[rd] = result;
                end
            `SW: begin
                    data_cache[eff_address]=s2_contents[31:24];
                    data_cache[eff_address+l]=s2_contents[23:16];
                    data_cache[eff_address+2]=s2_contents[15:8];
                    data_cache[eff_address+3]=s2_contents[7:0];
                end
            `SH: begin
                    data_cache[eff_address]=s2_contents[15:8];
                    data_cache[eff_address+1]=s2_contents[7:0];
                end
            `SB: begin
                    data_cache[eff_address]=s2_contents[7:0];
                end
        endcase
        $display("!!!!!!Completing the request.....\n");
        cpu_state =`fetch;
        PC=PC+1;
            end
  end
  endtask

  task print;
  integer i;
  begin

      $display ("!!!!!!inst = %x ---HALT----\n",inst_cache[PC]);
      $display ("**REGISTER CONTENTS AFTER THE EXECUTION OF THE
PROGRAM \"test.mem\"**\n");
      $display
("______________________________________________________________________\n");
      for (i=0;i<32;i=i44)
            begin
            $display ("FX_GPR[%0d] = %0d, FX_GPR[%0d] = %0d, FX_GPR[%0d]
= %0d, FX_GPR[%0d] =
  %0d\n",i,FX_GPR[i],i+1,FX_GPR[i+1],i+2,FX_GPR[i+2],i+3,FX_GPR[i+3]);
            end
      $display
("______________________________________________________________________\n\n");

  end
  endtask
```

```
endmodule

/****************************************************************************/




/*******************************************************************************/
```

The following is the test program run on the processor module. To run the model save the following test program on to a file called "test.mem".

```
34010010  //ori  r1,  r0,  16  ; r1 =16
00012021  //addu r4,   r0,   r1   ;  r4 = 16
27bd0017  //addiu r29, r29, 23      ;  r29 = 23
03ale023  //subu r28, r29, $1         ;  r28 = 7
340e0001  //ori  r14,  r0,  1  ; r14 = 1
340f0002 //ori  r15,  r0,  2        ; r15  = 2
0101102a //slt r2, r8,     r1     ;  r2 =1
0c000024 //jal 0x00000024    ; jmp lp: r31= pc+1 = 8;
04000000 //HALT
27b00010  //addiu $16, $29,  16      ; lp: r16= 23+16=39
8fb90008  //lw $25, 8($29)        ; r25 = mem[23+8] = 255;
 1020000c  //beq$1,$0,12
8fb20010  //lw $18,  16($29)        ; r18 = mem[23+16] = 255;
03e00008  //jr $31               goto Halt.
```

```
/*******************************************************************************/
```

VERILOG-XL 2.2.1   Apr 21,1996 12:33:34

Copyright (c) 1994 Cadence Design Systems, Inc. All Rights Reserved.
Unpublished -- rights reserved under the copyright laws of the United States.

Copyright (c) 1994 UNIX Systems Laboratories, Inc. Reproduced with Permission.

THIS SOFTWARE AND ON-LINE DOCUMENTATION CONTAIN CONFIDENTIAL INFORMATION
AND TRADE SECRETS OF CADENCE DESIGN SYSTEMS, INC. USE, DISCLOSURE, OR
REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR EXPRESS WRITTEN PERMISSION OF
CADENCE DESIGN SYSTEMS, INC.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to
restrictions as set forth in subparagraph (c)(l)(ii) of the Rights in
Technical Data and Computer Software clause at DFARS 252.227-7013 or
subparagraphs (c)(l) and (2) of Commercial Computer Software -- Restricted
Rights at 48 CFR 52.227-19, as applicable.

Cadence Design Systems, Inc.
555 River Oaks Parkway
San Jose, California 95134

For technical assistance please contact the Cadence Response Center at
1-800-CADENC2 or send email to crc_customers@cadence.com

For more information on Cadence's Verilog-XL product line send email to
talkverilog@cadence.com

```
Compiling source file "stim.v"
Compiling source file "sys_cntrl.v"
Compiling source file "processor.v"
Highest level modules:
stim

Time=3,SysAD =

zzzzzzzzzzzzzzzz,SysADC=zz,SysCmd=Ozz,ValidIn=1,ValidOut=1,ExtRqst=1,Release=1,RdR
dy=0,WrRdy=0,Reset=1,
    CS_req_valid=0,CS_AD=zzzzzzzzzzzzzzzz,CS_ADC=zz,CS_cmd=zzz,
SC_req_valid=0,ext_ready=0

******************FETCH****************************

!!!!!!PC=0,Inst = 34010010

------------------------DECODE------------------------

Time=30,SysAD =

zzzzzzzzzzzzzzzz,SysADC=zz,SysCmd=0zz,ValidIn=1,ValidOut=1,ExtRqst=1,Release=1,RdR
dy=0,WrRdy=0,Reset=1,
    CS_req_valid=0,CS_AD=zzzzzzzzzzzzzzzz,CS_ADC=zz,CS_cmd=zzz,
SC_req_valid=0,ext_ready=1

~~~~~~~~~~~~~~~~~~~~~~~~EXECUTE~~~~~~~~~~~~~~~~~~~~~

********Executing ORI**********

~~~~~~~~~~~~~~~~~~~~~~~~COMPLETE~~~~~~~~~~~~~~~~~~~~
```

```
**********************FETCH***************************
!!!!!!PC=1,Inst = 00012021
----------------------DECODE----------------------
~~~~~~~~~~~~~~~~~~~~~~~~EXECUTE~~~~~~~~~~~~~~~~~~~~
********Executing ADDU**********
~~~~~~~~~~~~~~~~~~~~~~~~COMPLETE~~~~~~~~~~~~~~~~~~~~
**********************FETCH***************************
!!!!!!PC=2,Inst=27bd0017
----------------------DECODE----------------------
~~~~~~~~~~~~~~~~~~~~~~~~EXECUTE~~~~~~~~~~~~~~~~~~~~
********Executing  ADDIU**********
~~~~~~~~~~~~~~~~~~~~~~~~COMPLETE~~~~~~~~~~~~~~~~~~~~
**********************FETCH***************************
!!!!!!PC=3,Inst  =  03ale023
----------------------DECODE----------------------
~~~~~~~~~~~~~~~~~~~~~~~~EXECUTE~~~~~~~~~~~~~~~~~~~~
********Executing  SUBU**********
~~~~~~~~~~~~~~~~~~~~~~~~COMPLETE~~~~~~~~~~~~~~~~~~~~
**********************FETCH***************************
!!!!!!PC=4,Inst = 340e0001
----------------------DECODE----------------------
~~~~~~~~~~~~~~~~~~~~~~~~EXECUTE~~~~~~~~~~~~~~~~~~~~
********Executing QRI**********
~~~~~~~~~~~~~~~~~~~~~~~~COMPLETE~~~~~~~~~~~~~~~~~~~~
**********************FETCH***************************
```

```
!!!!!!PC=5,Inst = 340f0002
------------------------DECODE-------------------------
~~~~~~~~~~~~~~~~~~~~~~~~~EXECUTE~~~~~~~~~~~~~~~~~~~~
********Executing ORI**********
~~~~~~~~~~~~~~~~~~~~~~~~COMPLETE~~~~~~~~~~~~~~~~~~~~
**********************FETCH***************************
!!!!!!PC=6,Inst = 0101102a
------------------------DECODE------------------------
~~~~~~~~~~~~~~~~~~~~~~~~~EXECUTE~~~~~~~~~~~~~~~~~~~~~
********Executing SLT**********
~~~~~~~~~~~~~~~~~~~~~~~~~COMPLETE~~~~~~~~~~~~~~~~~~~~
**********************FETCH***************************
!!!!!!PC=7,Inst = Oc000024
------------------------DECODE------------------------
-----------------------BRANCH EXECUTE-------------------
***** Executing JAL *****
**********************FETCH***************************
!!!!!!PC=9,Inst=27b00010
-----------------------DECODE--------------------------
~~~~~~~~~~~~~~~~~~~~~EXECUTE~~~~~~~~~~~~~~~~~~~~
********Executing ADDIU**********
~~~~~~~~~~~~~~~~~~~~~COMPLETE~~~~~~~~~~~~~~~~~~~~
Time=703,SysAD =
zzzzzzzzzzzzzzzz,SysADC=zz,SysCmd=0zz,ValidIn=1,ValidOut=1,ExtRqst=0,Release=1,RdR
dy=1,WrRdy=1,Reset=1,
    CS_req_valid=0,CS_AD=zzzzzzzzzzzzzzzz,CS_ADC=zz,CS_cmd=zzz,
SC_req_valid=0,ext_ready=1
```

************************FETCH***************************

!!!!!!PC=10,Inst = 8fb90008

Time=710,SysAD =

zzzzzzzzzzzzzzzz,SysADC=zz,SysCmd=0zz,ValidIn=1,ValidOut=1,ExtRqst=0,Release=0,RdRdy=1,WrRdy=1,Reset=1,
CS_req_valid=0,CS_AD=zzzzzzzzzzzzzzzz,CS_ADC=zz,CS_cmd=zzz,
SC_req_valid=0,ext_ready=0

!!! SYS_CNTRL: External agent asserted request

--------------------------DECODE---------------------------

Time=730,SysAD =

zzzzzzzzzzzzzzzz,SysADC=zz,SysCmd=0zz,ValidIn=1,ValidOut=1,ExtRqst=1,Release=0,RdRdy=1,WrRdy=1,Reset=1,
CS_req_valid=0,CS_AD=zzzzzzzzzzzzzzzz,CS_ADC=zz,CS_cmd=zzz,
SC_req_valid=0,ext_ready=0

!!! SYS_CNTRL: External agent asserted request

~~~~~~~~~~~~~~~~~~~~~~~~EXECUTE~~~~~~~~~~~~~~~~~~~~

Time=750,SysAD =

0000000000000000,SysADC=zz,SysCmd=002,ValidIn=0,ValidOut=1,ExtRqst=1,Release=l,
RdRdy=1,WrRdy=1,Reset=1,
CS_req_valid=0,CS_AD=0000000000000000,CS_ADC=zz,CS_cmd=002,
SC_req_valid=1,ext_ready=0

!!! SYS_CNTRL: External agent sending wr address

------------------COMPLETING LD/ST INST--------------------

!!!!Data not valid in the cache... Getting it from the main memory...

Time=770,SysAD =

zzzzzzzzzzzzzzzz,SysADC=zz,SysCmd=0zz,ValidIn=1,ValidOut=1,ExtRqst=1,Release=1,RdRdy=0,WrRdy=0,Reset=1,
CS_req_valid=0,CS_AD=zzzzzzzzzzzzzzzz,CS_ADC=zz,CS_cmd=0zz,
SC_req_valid=1,ext_ready=0

!!! SYS_CNTRL: External agent sending wr data

-------------------COMPLETING LD/ST INST-----------------
~~~~~~~~~~~~~~~~~~~~~~~~

!!!!Data not valid in the cache... Getting it from the main memory...

Time=790,SysAD =

zzzzzzzzzzzzzzzz,SysADC=zz,SysCmd=0zz,ValidIn=1,ValidOut=1,ExtRqst=1,Release=1,RdRdy=0,WrRdy=0,Reset=1,
CS_req_valid=0,CS_AD=zzzzzzzzzzzzzzzz,CS_ADC=zz,CS_cmd=zzz,
SC_req_valid=0,ext_ready=1

--------------------COMPLETING LD/ST INST-------------------

!!!!Data not valid in the cache... Getting it from the main memory...

Time=810,SysAD =

zzzzzzzzzzzzzzzz,SysADC=zz,SysCmd=0zz,ValidIn=1,ValidOut=1,ExtRqst=1,Release=1,RdRdy=0,WrRdy=0,Reset=1,
CS_req_valid=l,CS_AD=000000000000001f,CS_ADC=00,CS_cmd=001,
SC_req_valid=0,ext_ready=1

--------------------COMPLETING LD/ST INST------------------

!!!!!!Waiting for the system controller to return data....

Time=830,SysAD =

000000000000001f,SysADC=00,SysCmd=001,ValidIn=1,ValidOut=0,ExtRqst=1,Release=0,
RdRdy=0,WrRdy=0,Reset=1,
CS_req_valid=1,CS_AD=000000000000001f,CS_ADC=00,CS_cmd=001,
SC_req_valid=0,ext_ready=1

!!! SYS_CNTRL: Sending rd address

--------------------COMPLETING LD/ST INST------------------

!!!!!!Waiting for the system controller to return data....

Time=850,SysAD =

zzzzzzzzzzzzzzzz,SysADC=zz,SysCmd=0zz,ValidIn=1,ValidOut=1,ExtRqst=1,Release=1,RdRdy=0,WrRdy=0,Reset=1,
CS_req_valid=0,CS_AD=zzzzzzzzzzzzzzzz,CS_ADC=zz,CS_cmd=zzz,
SC_req_valid=0,ext_ready=1

!!! SYS_CNTRL: Waiting for data from external interface

!!! SYS_CNTRL: External interface returning data

--------------------COMPLETING LD/ST INST------------------

!!!!!!Waiting for the system controller to return data....

Time=870,SysAD =

0000000000000ff,SysADC=zz,SysCmd=101,ValidIn=0,ValidOut=1,ExtRqst=1,Release=1,RdRdy=0,WrRdy=0,Reset=1,
CS_req_valid=0,CS_AD=0000000000000ff,CS_ADC=zz,CS_cmd=101,
SC_req_valid=1,ext_ready=1

!!! SYS_CNTRL: External interface going back to idle after rdresponse

--------------------COMPLETING LD/ST INST------------------

!!!!!!Waiting for the system controller to return data....

!!!!!!Received data from the system controller....

!!!!!!Completing the request......

Time=890,SysAD =

zzzzzzzzzzzzzzzz,SysADC=zz,SysCmd=0zz,ValidIn=1,ValidOut=1,ExtRqst=1,Release=1,RdRdy=0,WrRdy=0,Reset=1,
CS_req_valid=0,CS_AD=zzzzzzzzzzzzzzzz,CS_ADC=zz,CS_cmd=zzz,
SC_req_valid=0,ext_ready=1

**********************FETCH****************************

!!!!!!PC=11,Inst = 1020000c

------------------------DECODE---------------------------

-----------------------BRANCH EXECUTE-----------------------

***** Executing BEQ *****

**********************FETCH****************************

!!!!!!PC=12,Inst=8fb20010

-----------------------DECODE--------------------------

~~~~~~~~~~~~~~~~~~~~~~~~~EXECUTE~~~~~~~~~~~~~~~~~~~~

--------------------COMPLETING LD/ST INST------------------

!!!!Data not valid in the cache... Getting it from the main memory...

Time=1030,SysAD =
~~~~~~~~~~~~~~~~~~~~~~~~~

zzzzzzzzzzzzzzzz,SysADC=zz,SysCmd=0zz,ValidIn=1,ValidOut=1,ExtRqst=1,Release=1,RdRdy=0,WrRdy=0,Reset=1,
CS_req_valid=1,CS_AD=0000000000000027,CS_ADC=00,CS_cmd=001,
SC_req_valid=0,ext_ready=1

--------------------COMPLETING LD/ST INST------------------

!!!!!!Waiting for the system controller to return data....

Time=1050,SysAD =

0000000000000027,SysADC=00,SysCmd=001,ValidIn=1,ValidOut=0,ExtRqst=1,Release=0,RdRdy=0,WrRdy=0,Reset=1,
CS_req_valid=1,CS_AD=0000000000000027,CS_ADC=00,CS_cmd=001,
SC_req_valid=0,ext_ready=1

!!! SYS_CNTRL: Sending rd address

--------------------COMPLETING LD/ST INST------------------

!!!!!!Waiting for the system controller to return data....

Time=1070,SysAD =

zzzzzzzzzzzzzzzz,SysADC=zz,SysCmd=0zz,ValidIn=1,ValidOut=1,ExtRqst=1,Release=1,RdRdy=0,WrRdy=0,Reset=1,
CS_req_valid=0,CS_AD=zzzzzzzzzzzzzzzz,CS_ADC=zz,CS_cmd=zzz,
SC_req_valid=0,ext_ready=l

!!! SYS_CNTRL: Waiting for data from external interface

!!! SYS_CNTRL: External interface returning data

--------------------COMPLETING LD/ST INST------------------

!!!!!!Waiting for the system controller to return data....

Time=1090,SysAD =

00000000000000ff,SysADC=zz,SysCmd=101,ValidIn=0,ValidOut=1,ExtRqst=1,Release=1,RdRdy=0,WrRdy=0,Reset=1,
CS_req_valid=0,CS_AD=00000000000000ff,CS_ADC=zz,CS_cmd=101,
SC_req_valid=l,ext_ready=l

!!! SYS_CNTRL: External interface going back to idle after rdresponse

--------------------COMPLETING LD/ST INST------------------

!!!!!!Waiting for the system controller to return data....

!!!!!!Received data from the system controller....

!!!!!!Completing the request.....

Time=1110,SysAD =

zzzzzzzzzzzzzzzz,SysADC=zz,SysCmd=0zz,ValidIn=1,ValidOut=1,ExtRqst=1,Release=1,RdRdy=0,WrRdy=0,Reset=1,
CS_req_valid=0,CS_AD=zzzzzzzzzzzzzzzz,CS_ADC=zz,CS_cmd=zzz,
SC_req_valid=0,ext_ready=l

**********************FETCH******************************

!!!!!!PC=13,Inst = 03e00008

------------------------DECODE---------------------------

-----------------------BRANCH EXECUTE-----------------------

***** Executing JR *****

***********************FETCH*****************************

!!!!!!PC=8,Inst = 04000000

-------------------------DECODE-------------------------

!!!!!!inst = 04000000 ---HALT----

REGISTER CONTENTS AFTER THE EXECUTION OF THE PROGRAM "test.mem"

FX_GPR[0] = 0, FX_GPR[1] = 16, FX_GPR[2] = 1, FX_GPR[3] = 0

FX_GPR[4] = 16, FX_GPR[5] = 0, FX_GPR[6] = 0, FX_GPR[7] = 0

FX_GPR[8] = 0, FX_GPR[9] = 0, FX_GPR[10] = 0, FX_GPR[11] = 0

FX_GPR[12] = 0, FX_GPR[13] = 0, FX_GPR[14] = 1, FX_GPR[15] = 2

FX_GPR[16] = 39, FX_GPR[17] = 0, FX_GPR[18] = 255, FX_GPR[19] = 0

FX_GPR[20] = 0, FX_GPR[21] = 0, FX_GPR[22] = 0, FX_GPR[23] = 0

FX_GPR[24] = 0, FX_GPR[25] = 255, FX_GPR[26] = 0, FX_GPR[27] = 0

FX_GPR[28] = 7, FX_GPR[29] = 23, FX_GPR[30] = 0, FX_GPR[31] = 8

L227 "processor.v": $finish at simulation time 1210
12172 simulation events + 5 accelerated events
CPU time: 0.3 secs to compile + 0.2 secs to link + 0.1 secs in simulation
End of VERILOG-XL 2.2.1 Apr 21,1996 12:33:34

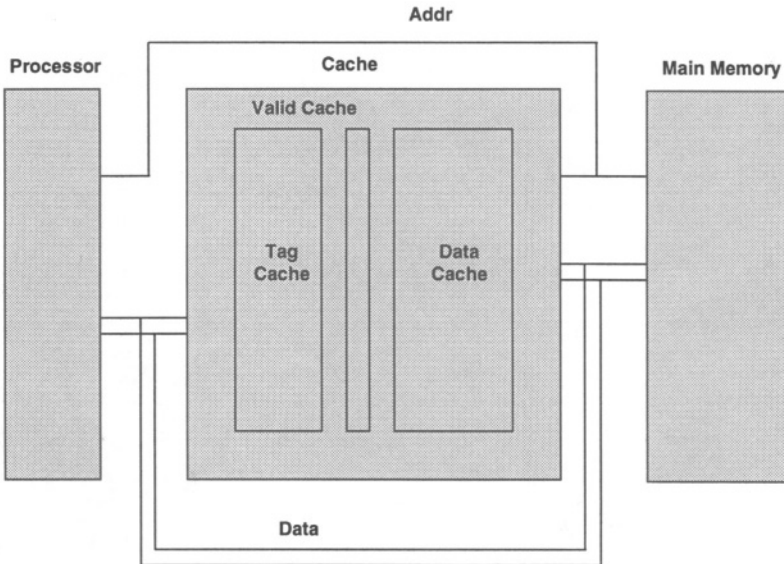
**/

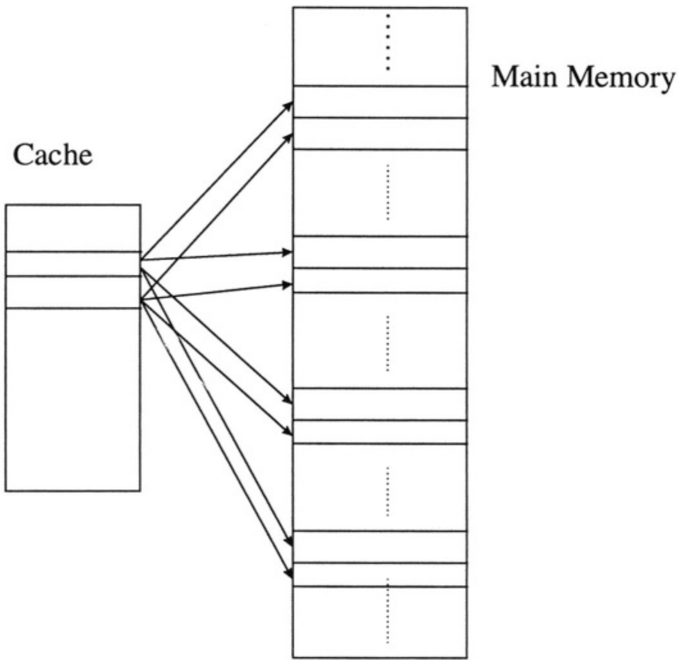
Example 11-2. R4200 microprocessor with instructions, bus-cycles, and registers – behavioral model.

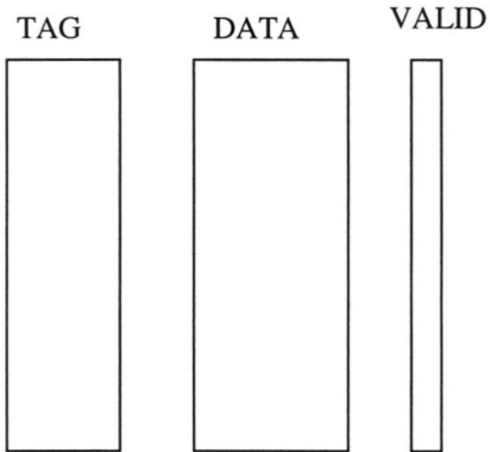
11.4 Example 3: Cache Design

11.4.1 Cache System: Architecture with State Diagram

The following page describes a cache design with an architectural state diagram.




Main Memory
Cache


TAG
DATA
VALID

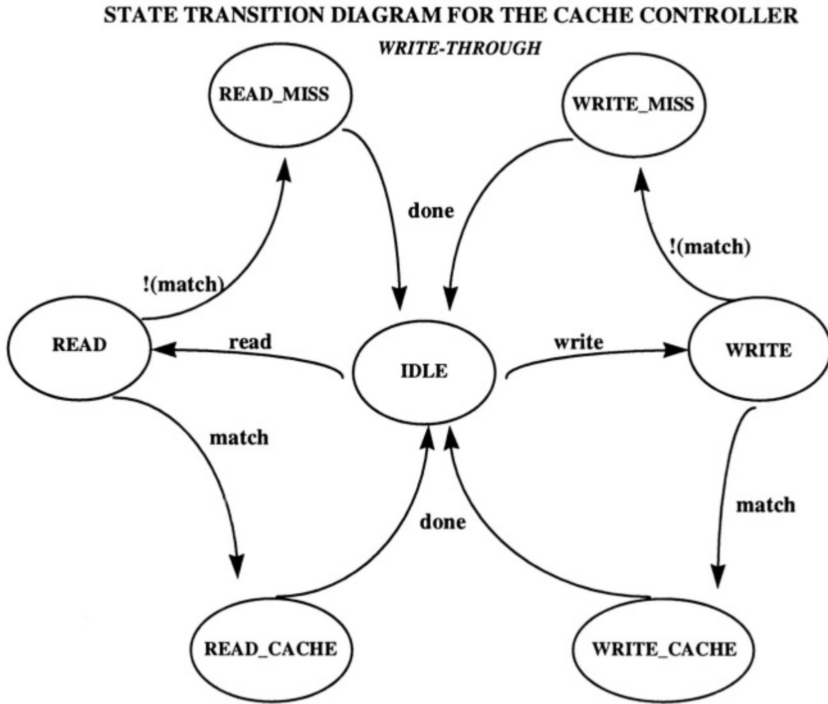


Figure 11-1. State Diagram for Cache Controller with Write-Back Policy

11.4.2 Cache System: Behavioral Model with Write-Through Policy

```
// Digital Design with Verilog HDL - Summer 1996 - cache controller model with write-
through policy

// 2 K Cache

`define CACHE_SIZE 2*1024
// This is limited by maximum size in this simulator
`define MEM_SIZE  128*1024

`define ADDR_SIZE  17
`define TAG_SIZE  6
//define states
`define IDLE 0
`define READ 1
`define WRITE 2
`define READ_MISS 3
```

```
`define READ_CACHE 4
`define WRITE_MISS 5
`define WRITE_CACHE 6

`define DATA_SIZE 64
`define CP 100
`define CACHE_DRV 1
`define NONCACHE_DRV 0

module cache(reset, addr, data, read, write, clock,
                                    buscntrl, done);
      input [`ADDR_SIZE-1:0] addr;
      inout [`DATA_SIZE-1:0] data;
      input read, write, clock, reset;
      input buscntrl;
      output done;     // indicates completion of cache operation

      reg [`TAG_SIZE-1:0] tagCache[`CACHE_SIZE-1:0];
      reg [`ADDR_SIZE-`TAG_SIZE-1:0] dataCache[`CACHE_SIZE-1:0];
      reg [`CACHE_SIZE-1:0] validCache;

      reg match;
      reg [7:0] state;
      integer i;
      integer index, tag;

   reg [`DATA_SIZE-1:0] MainMemory[0:`MEM_SIZE-l];
   reg [`DATA_SIZE-1:0] dataOut;

   reg done;

   assign data = (buscntrl== `CACHE_DRV) ? dataOut: 'bz;

   function get_caching_scheme;
   input [`ADDR_SIZE-1:0] addr;
   begin
      get_caching_scheme = 1;
   end
   endfunction

      /* Write-Through Algorithm:
            Read Operation:
                  Divide address bits into tag bits and index bits.
                  Match the address in tagCache[index] with tag.
                  If matched,
                        read from Cache;
                  Else
                        read from memory and if not read-miss then
                        copy that into cache.
            Write Operation:
```

```
        Always write into main memory.
        If a cacheable address then also write into cache.
        Update tag and data cache as well as valid indicators.
*/

        initial
        for (i=0; i <`CACHE_SIZE; i=i+1)
                validCache[i] = 0;

        always @ (read or write)
        begin
        done = 0;

        if (read)
                state = `READ;
        else
        if (write)
                state = `WRITE;

        while (state != `IDLE)
        begin
                @ (posedge clock)
                if (reset)
                begin
                                // Clear all validCache bits
                        for (i=0; i <`CACHE_SIZE; i=i+1)
                                validCache[index] = 0;
                end
                else
                begin
                        index = addr[`ADDR_SIZE-`TAG_SIZE-1:0];
                        tag = addr[`ADDR_SIZE-1 : `ADDR_SIZE-`TAG_SIZE];
                        if ((validCache[index]) &&(tagCache[index] == tag))
                                match = 1;
                        else
                                match = 0;

                        case(state)

                        `READ:
                        begin
                        // Match Found in cache
                                if (match)
                                        dataOut = dataCache[index];
                                else
                                        //a few possibilities here
                                        // read data from memory and also
                        // copy in cache or not copy in cache; determining
                        // this is part of another policy; obtain this info
                        // from another task; LRU algotihm means bring this in
```

```
        // OK
                if (get_caching_scheme(addr) == 0)
                    state = `READ_MISS;
                else
                    state = `READ_CACHE;
    end

        `READ_MISS :
        begin
            dataOut = MainMemory[addr];
            done = 1;
            state = `IDLE;
        end

        `READ_CACHE:
        begin
            dataOut = MainMemory[addr];
            dataCache[index] = data;
            tagCache[index] = tag;
            validCache[index] = 1;
            state = `IDLE;
        end

        `WRITE:
        begin
            if (get_caching_scheme(addr) == 0)
                state = `WRITE_MISS;
            else
                state = `WRITE_CACHE;
        end

        `WR1TE_MISS:
        begin
            MainMemory[addr] = data;
            if (match)
            // Do not maintain this location in cache any more
            validCache[index] = 0;
            state = `IDLE;
        end

        `WRITE_CACHE:
        begin
            MainMemory[addr] = data;
            validCache[index] = 1;
            dataCache[index] = data;
            tagCache[index] = tag;
            state = `IDLE;
        end

    endcase
```

```
            end
            end

        end

// TEST PORTION OF THE MODULE
        initial
        begin
            $readmemh("cache_t.dat",MainMemory);

        end

endmodule

`define NEW 1

`ifdef NEW

module text;

reg reset, clock, read, write;
reg [`ADDR_SIZE-1:0] addr;
reg buscntrl;
reg [`DATA_SIZE-1:0] data_reg;
wire [`DATA_SIZE-1:0] data;

cache c(reset, addr, data, read, write, clock, buscntrl, done);

assign data = (buscntrl == `CACHE_DRV) ? 'bz : data_reg;

always #50 clock = ~clock;

initial clock = 0;

initial
begin
  // $dumpfile("cache.dmp");
   $monitor($time,,clock,, reset,, read,, write,, data, addr,, c.state,,
                  c.match);

        #(`CP+1)
            reset = 1;
        #`CP
            reset = 0;
        #(5*`CP)
            addr = 2;
            read = 1;
            buscntrl = `CACHE_DRV;

        #(5*`CP)
```

```
        addr = 4;
        read = 0;
        write = 1;
        data_reg = 1000;

        buscntrl=`NONCACHE_DRV;
    #(5*`CP)
        addr = 2;
        read = 1;
        write = 0;
        buscntrl = `CACHE_DRV;
    #(10*`CP)
    $finish;
end

endmodule
`endif
```

Example 11-3. A cache system with a write-through policy; behavioral abstraction.

11.4.3 Cache System: Behavioral Model Modified for Write-Back Policy

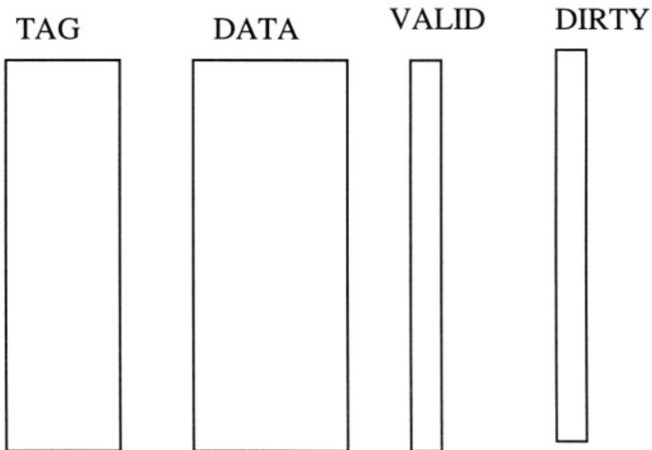


Figure 11-2. Block Diagram for the Cache Controller with Write-Back Policy Containing Dirty Bits

11.4.3.1 State Diagram

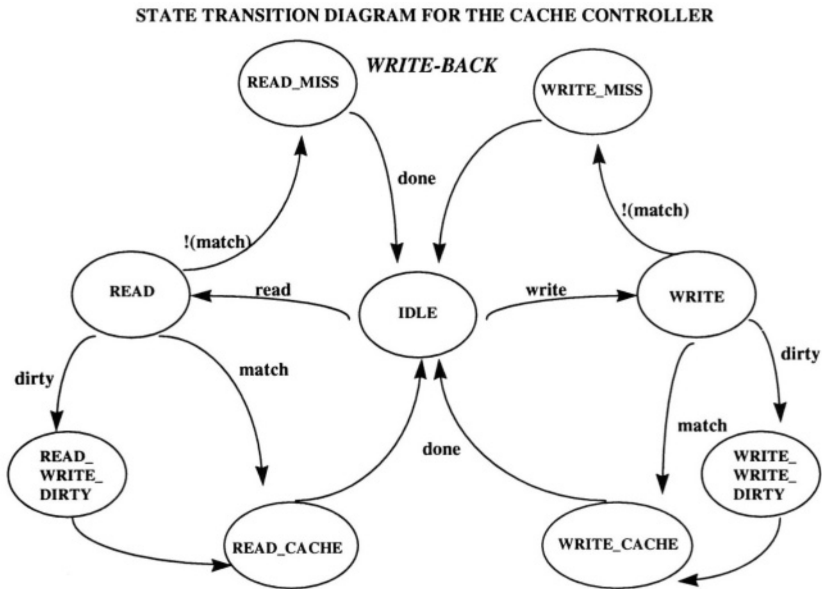


Figure 11-3. State Transition Diagram for the Cache Controller with Write-Back Policy

11.4.3.2 Verilog Source for the Cache Controller with Write-Back Policy

```
// Digital Design with Verilog HDL - Summer 1996 - cache controller model with write-back
policy

`define TAG_SIZE 6
// 256 K Cache

`define CACHE_SIZE 2*1024
`define ADDR_SIZE 17
`define IDLE 0
`define READ 1
`define WRITE 2
`define READ_MISS 3
`define READ_CACHE 4
`define WRITE_MISS 5
`define WRITE_CACHE 6
`define READ_WRITE_DIRTY 7
`define WRITE_WRITE_DIRTY 8

`define DATA_SIZE 64
```

```
// This is limited by maximum size in this simulator
`define MEM_SIZE 128*1024

`define CP 100
`define CACHE_DRV 1
`define NONCACHE_DRV 0

module cache(reset, addr, data, read, write, clock, buscntrl, done);
      input [`ADDR_SIZE-1:0] addr;
      inout [`DATA_SIZE-1:0] data;
      input read, write, clock, reset;
      input buscntrl;
      output done;     // indicates completion of cache operation

      reg [`TAG_SIZE-1:0] tagCache[`CACHE_SIZE-1:0];
      reg [`ADDR_SIZE-`TAG_SIZE-1:0] dataCache[`CACHE_SIZE-1:0];
      reg [`CACHE_SIZE-1:0] validCache;

      reg [`CACHE_SIZE-1:0] dirtyCache;

      reg [`ADDR_SIZE-1:0] wb_addr;

      reg match;
      reg [7:0] state;
      integer i;
            integer index, tag;

            reg [`DATA_SIZE-1:0] MainMemory[0:`MEM_SIZE-1];
            reg [`DATA_SIZE-1:0] dataReg;

            reg done;
            integer f;
            initial
            f = $fopen("state.out");

            assign data = (buscntrl== `CACHE_DRV) ? dataReg : 'bz;

function get_caching_scheme;
   input [`ADDR_SIZE-1:0] addr;
   begin
      get_caching_scheme = 1;
   end
endfunction

     task print_state;
      input [7:0] state;
            case(state)
                  `IDLE:
                        $fdisplay(f, "STATE=IDLE");
                   `READ: $fdisplay(f,"STATE=READ");
```

```
            `WRITE: $fdisplay(f, "STATE=WRITE");
            `READ_MISS:  $fdisplay(f,  "STATE=READ_MISS");
            `READ_CACHE: $fdisplay(f, "STATE=READ_CACHE");
            `WRITE_MISS: $fdisplay(f, "STATE=WRITE_MISS");
            `WRITE_CACHE: $fdisplay(f, "STATE=WRITE_CACHE");
            `READ_WRITE_DIRTY: $fdisplay(f, "STATE=READ_WRITE_DIRTY");
            `WRITE_WRITE_DIRTY:   $fdisplay(f,  "STATE=WRITE_WRITE_DIRTY");
        endcase
    endtask

    /* Write-Back Algorithm:
        Read Operation :
            Divide address bits into tag bits and index bits.
            Match the address in tagCache[index] with tag.
            If matched,
                read from Cache;
            Else
                read from memory and if not read-miss then
                copy that into cache.
        Write Operation:
            Divide address bits into tag bits and index bits.
            If dirty bit is on for this index, write the original
            data back to memory. Update the cache with the new
            data.                               */

        always  @(read or write)
        begin
        done = 0;

        if (read)
            state = `READ;
        else
        if (write)
            state = `WRITE;

        while (state  != `IDLE)
        begin
            @(posedge clock)
            if (reset)
            begin
                        // Clear all validCache bits
                for (i=0; i <`CACHE_SIZE; i=i+1)
                begin
                    validCache[index] = 0;
                    dirtyCache[i] = 0;
                end
            end
            else
            begin
                index =  addr[`ADDR_SIZE-`TAG_SIZE-1:0];
```

```
        tag = addr[`ADDR_SIZE-1: `ADDRSIZE-`TAG_SIZE];
        if ((validCache[index]) &&(tagCache[index] == tag))
            match = 1;
        else
            match = 0;
    print_state(state);
        case(state)

        `READ:
        begin
        // Match Found in cache
            if (match)
            begin
                dataReg = dataCache[index];
                done= 1;
                state = `IDLE;
            end
            else
                //a few possibilities here
                // read data from memory and also
        // copy in cache or not copy in cache; determining
        // this is part of another policy; obtain this info
        // from another task; LRU algotihm means bring this in
        // OK
                if (get_caching_scheme(addr) == 0)
                    state = `READ_MISS;
                else
                begin
                    if (dirtyCache[index])
                        state = `READ_WRITE_DIRTY;
                    else
                        state = `READ_CACHE;
                end
end

        `READ_MISS:
        begin
            dataReg = MainMemory[addr];
            done = 1;
            state = `IDLE;
        end

        `READ_CACHE:
        begin
            dataReg = MainMemory[addr];
            dataCache[index] = dataReg;
            tagCache[index] = tag;
            validCache[index] = 1;
            state = `IDLE;
        end
```

```
                `WRITE:
                begin
                        if (get_caching_scheme(addr) == 0)
                                state = `WRITE_MISS;
                        else
                        if  (dirtyCache[index])
                                state  = `WRITE_WRITE_DIRTY;
                        else
                        begin
                                state = `WRITE_CACHE;
                        end

                end

                `WRITE_MISS:
                begin
                        MainMemory[addr] = data;
                        if (match)
                        // Do not maintain this location in cache any more
                        validCache[index]  = 0;
                        state = `IDLE;
                end

                `WRITE_CACHE:
                begin
//                      MainMemory[addr] = data;
// In the writeback scheme, we do not update memory until a location
// in cache is being rewritten.

                        dirtyCache[index] = 1;
                        validCache[index]  = 1;
                        dataCache[index]  = data;
                        tagCache[index]  = tag;
            done =  1;
                        state  = `IDLE;
                end

                `READ_WRITE_DIRTY  :
                begin
                        wb_addr =  {tag, tagCache[index]};
                        MainMemory[wb_addr] = dataCache[index];
                        dirtyCache[index]  = 0;
                        state = `READ_CACHE;
                end

                `WRITE_WRITE_DIRTY  :
                begin
                        wb_addr =  {tag,  tagCache[index]};
                        MainMemory[wb_addr]  =  dataCache[index];
```

```
                    state=`WRITE_CACHE;
                end
            endcase
        end
        end

    end

// TEST PORTION OF THE MODULE
    initial
    begin
        $readmemh("cache_t.dat",MainMemory);

    end

endmodule

`define NEW 1

`ifdef  NEW

module test;

reg reset, clock, read, write;
reg [`ADDR_SIZE-1:0] addr;
reg buscntrl;
reg [`DATA_SIZE-1:0] data_reg;
wire [`DATA_SIZE-1:0] data;

integer  i;
integer f;

        cache c(reset, addr, data, read, write, clock, buscntrl, done);

        assign data = (buscntrl == `CACHE_DRV) ? 'bz : data_reg;

        always #50 clock = ~clock;

        initial clock = 0;

        initial
        begin
          f = $fopen("fmon.out");
         // $dumpfile("cache.dmp");
          $fmonitor(f,$time,,clock,, reset,, read,, write,, data, addr,, c.state,,
            c.match);

    #(`CP+1)
        reset = 1;
    #`CP
```

```
        reset = 0;

    #(5*`CP)
        addr = 2;
        read = 1;
        buscntrl = `CACHE_DRV;

    #(5*`CP)
        addr = 4;
        read = 0;
        write = 1;
        data_reg = 1000;
        buscntrl = `NONCACHE_DRV;

    #(5*`CP)
        write = 0;
        #1
        write = 1;
        addr = 4;
        read = 0;
        write = 1;
        buscntrl = `NONCACHE_DRV;
        data_reg = 1010;

    #(10*`CP)
    for (i=0; i< 500; i=i+1)
    begin
        #(5*`CP)
            addr = $random;
            read = $random;
            write = ~read;
            buscntrl = (read) ? `CACHE_DRV : `NONCACHE_DRV;
            if (write)
                data_reg = $random;
    end
    $finish;
end

endmodule
`endif
```

Example 11-4. ***Cache system with a write-back policy: behavioral model – refinement of Example 11-3.***

11.4.4 Cache System: Implementation: Write-Through Policy

11.4.4.1 Block Diagram

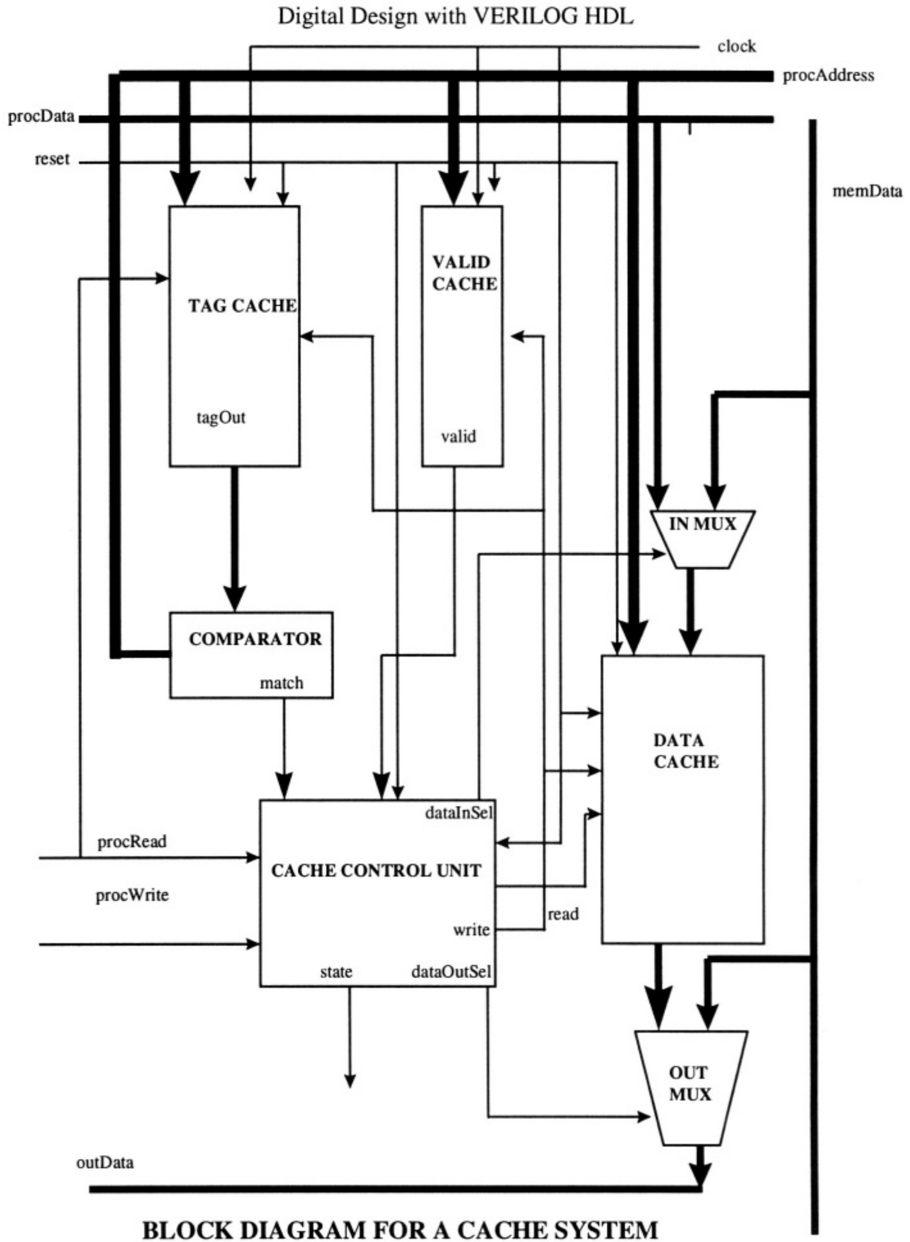


Figure 11-4. Block Diagram for a Cache System

11.4.4.2 Verilog Source

```
`define TAG_SIZE  6
// 256 K Cache

`define CACHE_SIZE 2*1024
`define ADDR_SIZE 17
`define IDLE 0
`define READ 1
`define WRITE 2
`define READ_MISS 3
`define READ_CACHE 4
`define WRITE_MISS 5
`define WRITE_CACHE 6
`define READ_WRITE_DIRTY  7
`define WRITE_WRITE_DIRTY  8

`define READ_MISS1 9
`define READ_MISS2 10
`define WRITE_MEM 11

`define DATA_SIZE 64

// This is limited by maximum size in this simulator
`define MEM_SIZE  128*1024

`define CP  100
`define CACHE_DRV 1
`define NONCACHE_DRV 0
`define FULL 1

`ifdef FULL
module Processor(procRead, procWrite, procAddress, procData, procClock, reset);
    output procRead, procWrite;
    inout  [`ADDR_SIZE-1:0]procAddress;
    inout  [`DATA_SIZE-1:0] procData;
    input  procClock;
    input reset;
endmodule

module MainMemory(memRead, memWrite, memAddress, memData, memClock,reset);
    input memRead, memWrite, memClock, reset;
    input [`ADDR_SIZE-1:0] memAddress;
    input [`DATA_SIZE-1:0] memData;
endmodule
`endif

module Cache(procRead, procWrite, procAddress, procData,
                memRead, memWrite, memAddress, memData, reset, clock);
    input procRead, procWrite, reset, clock;
```

```
    input [`ADDR_SIZE-1:0] procAddress;
    output memRead, memWrite;
    output [`ADDR_SIZE-1:0] memAddress;
    inout [`DATA_SIZE-1:0] memData, procData;

    wire [`DATA_SIZE-1:0] dataIn, outData, dataOut;
    wire [`TAG_SIZE-1:0] tagOut;


    tagCache tc(procAddress, tagOut, clock, write, procRead, reset);
    validCache vc(procAddress, valid, clock, write, procRead, reset);
    dataCache dc(procAddress, dataIn, dataOut, clock, write, read);
    comparator c(tagOut, procAddress[`ADDR_SIZE-1:`ADDR_SIZE-`TAG_SIZE], match);
    cacheControl cc(procRead, procWrite, match, valid, read, write, memWrite,
                    memRead, dataOutSel, dataInSel, clock, reset);
    dataMux dmIn(procData, memData, dataInSel, dataIn);
    dataMux dmOut(dataOut, memData, dataOutSel, outData);

endmodule

module System();

    wire [`ADDR_SIZE-1:0] memAddress;
    wire [`DATA_SIZE-1:0] memData;
    wire [`ADDR_SIZE-1:0] procAddress;
    wire [`DATA_SIZE-1:0] procData;
    Processor p(procRead, procWrite, procAddress, procData, procClock,
        reset);
    MainMemory m(memRead, memWrite, memAddress, memData, memClock,
            reset);
    Cache c(procRead, procWrite, procAddress, procData,
            memRead, memWrite, memAddress, memData, reset, clock);

endmodule


/**** Modules within cache */
/* In the previous level of abstraction, most of these were abstracted to memories. The
reading and writing to these memories or registers was modeled at a higher level without the
structural details. In this, we refine these to more detail where functional units are organized
in the level of detail that has internal units and synchronization between these is also
implemented.
*/
/* Assume that read or write is a pulse asserted for clock cycle when
that operation begins */

module tagCache(address, tagOut, clock, write, read, reset);

    input [`ADDR_SIZE-1:0] address;
    input clock, write, read, reset;
```

```
    output [`TAG_SIZE-1:0] tagOut;

    reg [`TAG_SIZE-1:0] tagOut;
    reg [`TAG_SIZE-1:0] tagCacheMem[`CACHE_SIZE-1:0];

    always @write
        if (write)
            @(negedge clock)
                                        tagCacheMem[address[`ADDR_SIZE-TAG_SIZE-
                                1:0]] = address[`ADDR_SLZE-1 : `ADDR_SIZE-`TAG_SIZE];


    always @read
        if (read)
            @(posedge clock)
                tagOut = tagCacheMem[address[`ADDR_SIZE-`TAG_SIZE-1:0]];

endmodule

module validCache(address, validOut, clock, write, read, reset);
    input [`ADDR_SIZE-1:0] address;
    input clock, write, read, reset;
    output validOut;
    reg validOut;
    reg [`CACHE_SIZE-1:0] validCacheMem;

    always @reset
            if (reset)
            validCacheMem =`CACHE_SIZE'bO;

    always @write
        @ (negedge clock)
            if(!reset)
                validCacheMem[address[`ADDR_SIZE-`TAG_SIZE-1:0]]=1;


    always @read
        @(posedge clock)
            validOut = validCacheMem[address[`ADDR_SIZE-`TAG_SIZE-1:0]];

endmodule


module dataCache(address, dataIn, dataOut, clock, write, read);
    input [`ADDR_SIZE-1:0] address;
    input clock, write, read;
    output [`DATA_SIZE-1:0] dataOut;

    reg [`DATA_SIZE-1:0] dataOut;
```

```
    input [`DATA_SIZE-1:0] dataIn;

    reg  [`DATA_SIZE-1:0]  dataCache[`CACHE_SIZE-1:0];

    always @write
        if  (write)
         @(posedge clock)
            dataCache[address[`ADDR_SIZE-`TAG_SIZE-1:0]]=dataIn;

    always @read
        if (read)
            @(posedge clock)
                dataOut   =   dataCache[address[`ADDR_SIZE-`TAG_SIZE-1:0]];

endmodule


module comparator(in1, in2, match);
    input  [`TAG_SIZE-1:0]  in1,  in2;
    output match;

    assign match = (in1 == in2);
endmodule

module cacheControl(procRead, procWrite, match, valid, read, write, memWrite,
                    memRead,dataOutSel,dataInSel, clock, reset);


    input procRead, procWrite, match, valid, clock, reset;
    output memWrite, memRead,dataOutSel, dataInSel, read, write;
    reg memWrite, memRead,dataOutSel, dataInSel, read, write;
    reg  [7:0]  state, nextState;

    always  @(posedge clock)
        state = reset ?  `IDLE  :   nextState;

    always  @(state or procWrite or procWrite or match or valid)
    case(state)

    `IDLE :  if (procRead)
                    nextState = `READ;
                else if (procWrite)
                    nextState = `WRITE;
                else
                    nextState = `IDLE;

    `READ : if (match && valid)
                    nextState = `IDLE;
                else
                    nextState = `READ_MISS1;
```

```
`READ_MISS1:/*   Read from main memory takes 2 cycles */
            nextState = `READ_MISS2;

`READ_MISS2:    nextState = `IDLE;

`WRITE: /* Check principles of write-hit from Mips documentation*/
            /* Current plan : always save in cache */
            /* Outputs here consist of writing to cache */
            nextState = `WRITE_MEM;

`WRITE_MEM :
            nextState = `IDLE;

endcase

/* Output combinational logic from controller */
/* The signals in the output list are
memRead, memWrite, dataInSel, dataOutSel, read, write
*/
always @state
      case (state)

      `READ :
            begin
             read = 1;
             dataOutSel = ~(match & valid);
            end

      `READ_MISS1:
            begin
            memRead = 1;
            dataInSel= 1;
            end


      `READ_MISS2:
            begin
            memRead <= #`CP 0;
            write = 1;
            write <= #`CP 0;
            end



      `WRITE:
            begin
            memWrite = 1;
            write  = 1;
            dataInSel = 0;
            end
```

```
        `WRITE_MEM:
            begin
            memWrite = 0;
            write = 0;
            memWrite <= #`CP 0;
            dataInSel <= #`CP 1;
            end
        endcase
    endmodule


/* This mux will output data onto processor for read */
module dataMux(data0, data1, dataSel, outData);
    input [`DATA_SIZE-1:0] data0, data1;
    input dataSel;
    output [`DATA_SIZE-1:0] outData;
    // cache controller must generate a 1 for read_cache and
    // 0 for read_miss states for dataSel lines.

    assign outData = dataSel ? data0 : data1;

endmodule
```

Example 11-5. A register transfer level model of the cache system with write-through policy: with blocks.

11.5 Memory Model with Bus Cycle Timing and with Timing Checks

```
module ram_example(addr, we_, io, ce1_, ce2_, oe_, vcc, vss);
/*
 64K x 4 RAM
*/
//    DECLARE PORTS
`define    mem_size 65536
`define    addr_size 16
`define data_size 4

input    [(`addr_size-1):0] addr;
inout    [(`data_size-1):0] io;
input    ce1_, ce2_, we_, oe_;
input    vcc, vss;
// Define SIZE Constants

`define    asize `addr_size
`define    dsize `data_size
`define    size `mem_size
```

```
//DEFINE TIMING CONSTANTS
//     READ CYCLE
`define    tRC 25    // Minimum read cycle time
`define    tAA 25    // Minimum address to data valid
`define    tOHA      3      // Maximum Output Hold from Address Change
`define    tACE      25    //Maximum CE_Bar low to data valid
`define    tDOE      15    //Maximum OE_Bar Low to data valid
`define    tLZOE    3      // Maximum OE_Bar Low to data valid
`define    tHZOE    15   //  Maximum CE_Bar Low to Power up
`define    tLZCE    5      //  Minimum CE_Bar High to High Z
`define    tHZCE    10    // Maximum CE_Bar High to High Z
`define    tPU  0      //Minimum CE_Bar Low to Power up
`define    tPD  25    //Maximum CE_Bar High to Power Down

//     WRITE CYCLE
`define tWC    20    //      Minimum Write Cycle Time
`define    tSCE      20    // Minimum CE_Bar Low to Write End
`define    tAW20    // Minimum Address setup to write end
`define    tHA 0      // Minimum Address Hold from Write End.
`define    tSA  0      // Minimum CE_Bar Low to Power up
`define    tPWE      20    // Minimum CE_Bar Low to Power up
`define    tSD  13    // Minimum CE_Bar Low to Power up
`define    tHD 0      // Minimum CE_Bar Low to Power up
`define    tLZWE    3      // Minimum CE_Bar Low to Power up
`define    tHZWE    7      // Minimum CE_Bar Low to Power up


wire ce_ = ce1_ && ce2_;


// DECLARE MEMORY, OUTPUT BUFFER
reg  [(`data_size-1):0]  mem[(`mem_size-1):0];
reg  [(`data_size-1):0]  out_buf;


//DECLARE IO AS TRI BY BUS_CONTROL FLAG
reg  bus_control;
tri  [(`data_size-1):0] io = bus_control ? out_buf : 4'bzzzz:

// DECLARE TIME VARIABLES TO HOLD ABSOLUTE TIME OF PREVIOUS EVENT
time       addr_t, io_t, ce_0_t, ce_1_t, we_0_t, we_1_t, oe_0_t, curr_t;
time read_cycle_start_time,  write_cycle_start_time;
time delay_to_output_x, delay_to_output_valid;

//DECLARE REG TO INDICATE START OF READ CYCLE
reg read_flag;

// DETECT SIGNAL CHANGES
always
      @(addr)       addr_t = $time;
```

```
always
      @(negedge ce_)     ce_0_t = $time;

always
      @(posedge ce_)     ce_1_t = $time;

always
      @(negedge oe_)     oe_0_t = $time;

always
      @(posedge we_)    we_0_t = $time;


// **** READ CYCLE MODEL
// DETECT START OF A READ CYCLE
always
      @(addr or negedge(ce_) or negedge(oe_) )
      begin :  read_cycle_start_block
      if ((we_== 1) && (ce_ == 0) && (oe_ == 0))
            // FLAG READ_CYCLE_1 BLOCK
            read_flag = 1;
      end

// INITIATE READ CYCLE
always
      begin : read_block
            wait (read_flag == 1)
                  read_cycle_start_time = $time;
            // DELAY UNTIL OUTPUT IS ACTIVE
            find_del_to_op_x;
            del_to_op_valid_n_chk_sig_chg;
            out_buf   = mem[addr];
            // WAIT FOR THIS READ CYCLE TO END
            fork : read_end_fork
                        @(addr)
                        disable read_end_fork;
                  @(ce_)
                        begin
                        #`tHZCE bus_control = 0;
                        disable read_end_fork;
                        end
                  @(oe_)
                  begin
                        #`tHZOE bus_control = 0;
                              disable read_end_fork;
                  end
                  @(we_)
                        begin       // No delay specified for we_ to end of read
                        bus_control = 0;
```

```
                    disable read_end_fork;
                    end

      join
            // CHECK READ CYCLE TIME
            if ( ($time - read_cycle_start_time) < `tRC)
            begin
                  $display("Error : Read cycle time of %d violated at time
%d\n",read_cycle_start_time,$time);
                  disable read_block;
            end
      read_flag = 0;
      end

// TASKS TO DETERMINE DELAY TIME DURING READ CYCLE
task find_del_to_op_x;
begin
      delay_to_output_x = addr_t + `tOHA;
      if (delay_to_output_x < (ce_0_t + `tLZCE))
            delay_to_output_x = ce_0_t + `tLZCE;
      if (delay_to_output_x < (ce_0_t + `tLZOE))
            delay_to_output_x = ce_0_t + `tLZOE;
      delay_to_output_x = delay_to_output_valid - $time;
end
endtask

task  find_delay_to_output_valid;
begin
      delay_to_output_valid = addr_t + `tAA;
      if (delay_to_output_valid < (ce_0_t + `tACE))
            delay_to_output_valid = ce_0_t + `tACE;
      if (delay_to_output_valid < (oe_0_t  + `tDOE))
            delay_to_output_valid = oe_0_t + `tDOE;
      delay_to_output_valid =  delay_to_output_valid  - $time;
end
endtask



task del_to_op_x_n_chk_sig_chg;
   begin: del_to_out_x_fork
      fork
      // DELAY TO OUTPUT X
      #delay_to_output_x       disable del_to_out_x_fork;
      // CHECK FOR CHANGES ON CONTROL SIGNALS
       @(ce_)   $display("Warning : CE_ changed before read data active");
       @(oe_)   $display("Warning : OE_ changed before read data active");
       @(we_)  $display("Warning : WE_ changed before read data active");
      // TEST OUT IF, @(ce_), the disable below might affect the display
      // ABOVE BY PLACING A BUNCH OF DISPLAYS IN THE @(ce_) CHECK ABOVE
```

```
        @(addr)
              begin
                    $display("Warning : Addr changed before read data active");
                    // read_flag remains 1 so that new read cycle starts
                    bus_control = 0;     // MAY NOT OCCUR IMMEDIATELY
                    disable read_block;
              end

      join
      end
endtask

task del_to_op_valid_n_chk_sig_chg;
      fork : del_to_out_valid_fork
            #delay_to_output_valid disable del_to_out_valid_fork;
            //check for changes on control signals
            @(ce_)    $display("Warning : CE_ changed before read data valid");
            @(oe_)    $display("Warning : OE_ changed before read data valid");
            @(we_)  $display("Warning : WE_ changed before read data valid");

            @(we_)  $display("Warning : WE_ changed before read data valid");
            @(ce_ or oe_ or we_)
                  begin
                        read_flag = 0;
                        bus_control = 0;
                        disable read_block;
                  end
            @(addr)
            // CHECK FOR EVENT ON ADDRESS BUS
                  begin
                        $display("Warning : Addr changed before read data active");
                        // read_flag remains 1 so that new read cycle starts
                        bus_control = 0; // MAY NOT REALLY OCCUR IMMEDIATELY
                        disable read_block;
                  end
      join
endtask

task del_to_out_valid_n_chk_for_sig_chg;
    begin: del_to_valid_fork
      fork
      // DELAY TO OUTPUT X
      #delay_to_output_valid disable del_to_valid_fork;
      // CHECK FOR CHANGES ON CONTROL SIGNALS
       @(ce_) $display("Warning : CE_ changed before read data valid");
       @(oe_) $display("Warning : OE_ changed before read data valid");

       @(we_) $display("Warning : WE_ changed before read data valid");
       @(ce_ or oe_ or we_)
```

```
            begin
                  read_flag = 0;
                  bus_control = 0; // MAY NOT REALLY OCCUR IMMEDIATELY
                  disable read_block;
            end

      join
       end
endtask

always
      begin : write_block
            bus_control = 0;
            wait ((we_ == 0) && (ce_ == 0))
                  write_cycle_start_time = $time;
                  if ((write_cycle_start_time - addr_t) < `tSA)
                        begin
                              $display("Warning : Address Set-Up to Wrote Start time violated at
%d", $time);
                              disable write_block;
                        end
                  fork : write_end_fork
                        @(ce_or we_)
                        begin
                              // CHECK CE_ LOW TO WRITE END
                              if (($time - ce_0_t) < `tSCE)
                                    $display("Warning :  CE_ low to write end time= %d.", $time);
                              else
                              if ( ($time - we_0_t) < `tPWE)
                                    $display("Warning : WE_ Pulse Width (`tPWE) time violation at
time=%d",
                                                $time);
                              else
                              if (( $time - addr_t) < `tAW)
                                    $display("Warning :  Address setup (`tPWE time violation at
time=%d",
                                                $time);
                              else
                              if (( $time - io_t) < `tSD)
                                    $display("Warning :  Data Setup (`tSD time violation at
time=%d", $time);
                              else
                                    mem[addr] = io;
                              $display("the write cycle is being exec., io=%h, addr=%h", io, addr);
                              disable write_block;
                        end

                        @(addr)
                              begin
```

```
                        $display("Warning : Address changed before end of write cycle");
                            disable write_block;
                        end
                    @(io)
                        begin
                        $display("Warning : Data changed before end of write cycle");
                        disable write_block;
                        end
                join
        end
endmodule
```

Example 11-6. Detailed model of a cache controller with write-through policy.

11.6 Exercises

1. Write a model for write-back policy cache controller similar to the Example 11-5 containing RTL level description and details of synchronization and combinational logic and state machine controller description. Start with the behavioral model in Examples 11-4 and the 11-5 to obtain this model.

2. In the Example 11-1, identify the concurrent processing of reset and instructions and the usage of disable. How do the other interrupts take place in this model? Can one use functions to model task execute_instructions? Check the steps in decoding of instructions for the following instructions—dcr b; dcr m; add b; add m.

3. For Example 11-2, add pipelining features for the instruction fetch modeled in the always loop at the beginning of the processor module. See the speedup in the processor execution.

12 SYNTHESIS WITH VERILOG

12.1 Logic Synthesis and Behavioral Synthesis

Synthesis converts Verilog HDL models of hardware down to gate-level implementations automatically and maps these into target technology. Synthesis also optimizes the design for a given set of constraints related to area and speed. The synthesis techniques and tools are commonly classified into logic and behavioral synthesis. These techniques apply to HDL descriptions at the logic level and at the behavioral level, respectively. Synthesis allows mapping of same HDL description into multiple target technologies without any change in the design.

12.2 Design Flow with Synthesis

The flow for a typical Verilog based design includes logic synthesis as shown in the Figure 12-1.

12.2.1 Typical Design Flow with Verilog

The Figure 12-1 illustrates a typical design flow with Verilog. A top-down design starts with a behavioral description and is finally sent to the fab after complete placement, layout and final verification as shown in this diagram.

1. Write a high-level behavioral description of the planned design. This step starts with concepts and ends up with a high level description in the Verilog language. This description can have various levels of detail and essentially has architectural elements and algorithmic elements. This may be used with behavioral synthesis for some specialized parts but in general will be simulated for verifying the parameters, algorithms and architecture. Example here includes the cache controller models for write-through and write-back schemes (Example 11.3–11.5). Some level of tests are generated at this point.

2. Next we perform stepwise refinement to the RTL level. This is again simulated and verified for functional correctness. We also check for the RTL synthesis subset during this process. Here we first use the tests developed in step 1 and add tests for the details added at this level. For example, for a cache controller, all communicating wires and registers are modeled here as opposed to higher level models of the blocks in step 1. Thus, correctness of all signals at the (logic) synthesizable blocks are tested in this step.

3. Synthesize the HDL description with the synthesizer. In a typical Synthesizer like Synopsys, this step is divided into two parts—HDL Compilation and the Design Compilation. Synthesizer performs architectural optimizations, then creates an internal representation of the design. Use the Synthesis Design Compiler to produce an optimized gate level description in the target ASIC library. You can optimize the generated circuits to meet the timing and area constraints wanted. This optimization step must follow the translation to produce an efficient design.

4. The output of a synthesizer is a gate-level Verilog description. This netlist-style description uses ASIC components as the leaf-level cells of the design. The gate-level description has the same port and module definitions as the original high-level Verilog description 1. The gate-level Verilog description from step 3 is now passed through the Verilog simulator. You can use the original Verilog simulation drivers from step 1 and 2 because module and port definitions are preserved through the translation and optimization processes. Compare the output of the gate-level simulation (step 4) against the output of the original Verilog description simulation (step 3) to verify that the implementation is correct.

5. The synthesis tools can be used at behavioral and at the RTL level. The RTL level is synthesized using techniques that are commonly known as logic synthesis. In this book, the major components of this flow will be discussed. The various representations in Verilog like behavioral, RTL and structural occur at different places in this design cycle and will be discussed fully. Simulation aspects will be discussed for each of those and as a whole as well with the semantic model adding to the depth of this understanding. Synthesis with Verilog will be discussed in various sections and then in Chapter 12. Timing descriptions that are especially important for post-layout verification will be discussed in Chapter 13 on specify blocks,

In the Figure 12-1, we see the design flow within the synthesis. Synthesis has two aspects—technology independent and technology dependent. This is explained in Figure 12-2. In the Figure 12-3, we show how behavioral synthesis complements the logic synthesis. The typical design flow is explained below:

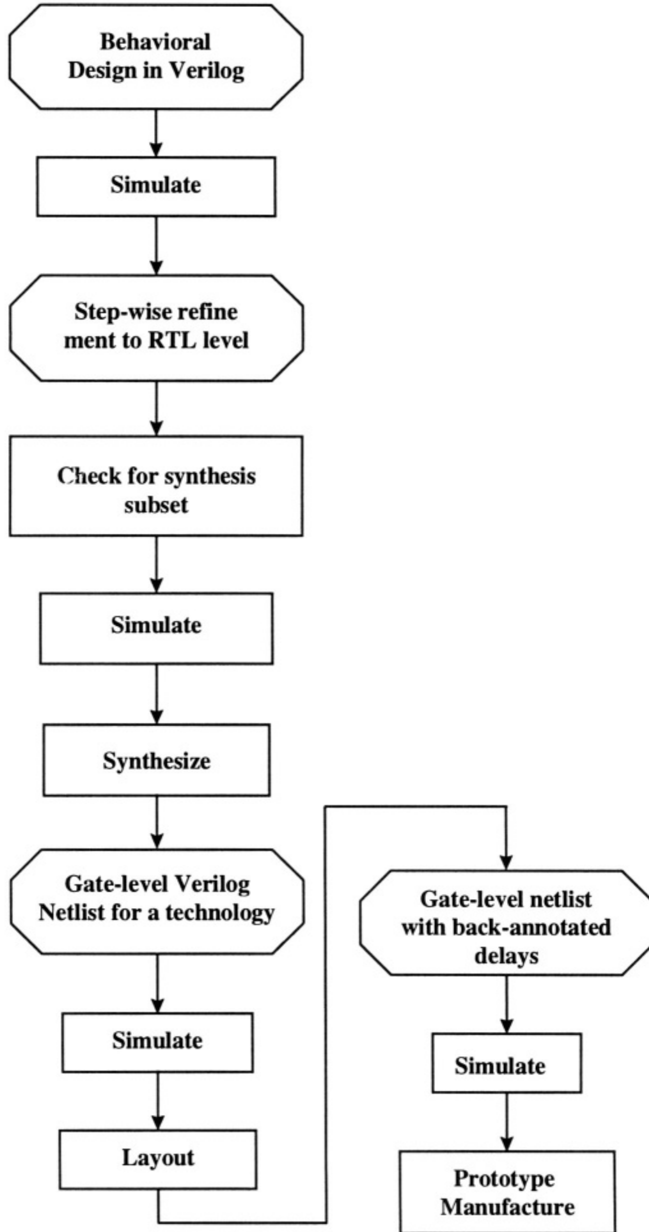


Figure 12-1. Typical Design Flow with Verilog Including Synthesis

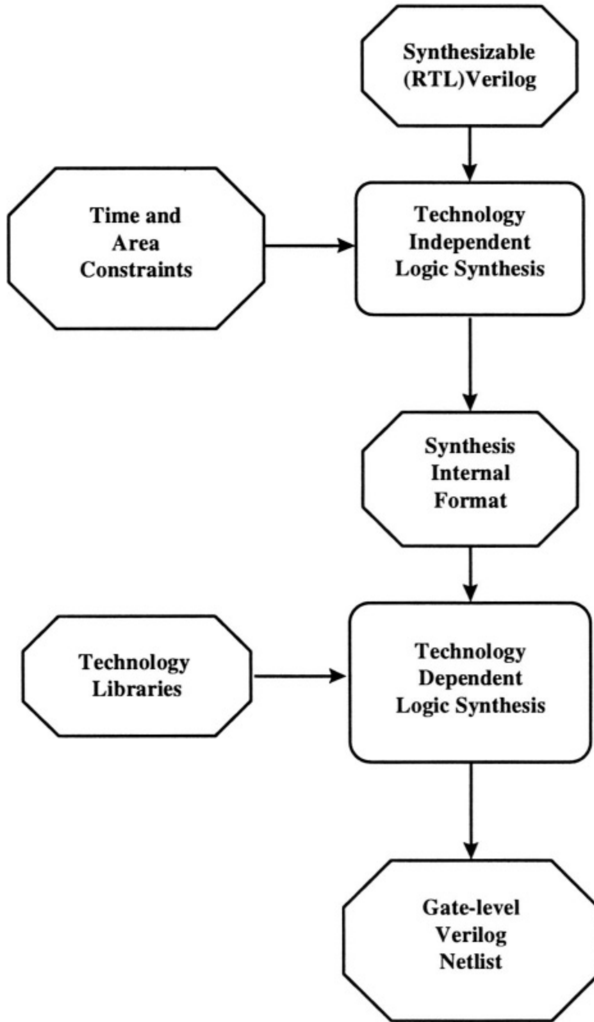


Figure 12-2. Logic Synthesis Components of Verilog Based Synthesis

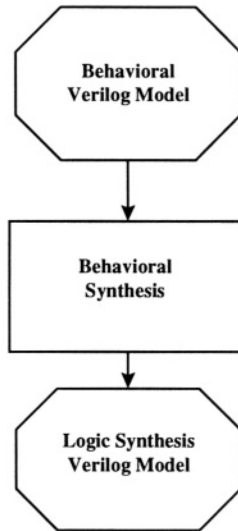


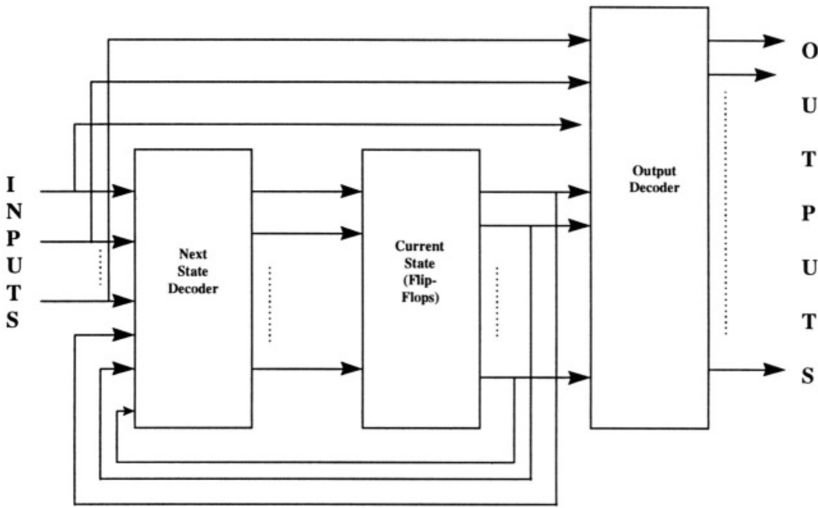
Figure 12-3. Components of Behavioral Synthesis with Verilog

12.3 Logic Synthesis View

Logic synthesis view of design is similar to the traditional view of sequential system whereby the system is separated into the 'next-state' generation logic, current state description and output generation logic. The first block diagram shows the traditional view (also known as Class A or Mealy machine). The second diagram shows the machine as a Verilog description (Sagdeo machine!). This is the modern view of the design that is now amenable to logic synthesis. A group of state machine and combination logic blocks as shown in figure. Each of these blocks is synthesized using state-machine synthesis and combinational synthesis techniques. Examples of this class of machines include the cache controller described in section 11.2 and the traffic light controller in Example 12-3 described below. Examples of pure combinational logic that is commonly synthesized include the datapath design of section on RTL descriptions.

Synthesis has two major steps: technology independent and technology dependent. The mapping is then done into target technologies via libraries supported in synthesis like asics by LSI Logic, VLSI, etc., and fpgas from XILINX, PLDs from Altera, etc.

The datapath synthesis will be done using: RTL and gate level descriptions. Some simple behavioral logic that either models a state machine or a combinational logic block is supported. State Machine Synthesis will be done from state machine descriptions via simple behavioral descriptions (1 state machine per module).



CLASS A FINITE STATE MACHINE (MEALY M/C)

[Maps to Logic Synthesis View of a Design]

Figure 12-4. Traditional View (Class A or Mealy Machine) of a Sequential Design

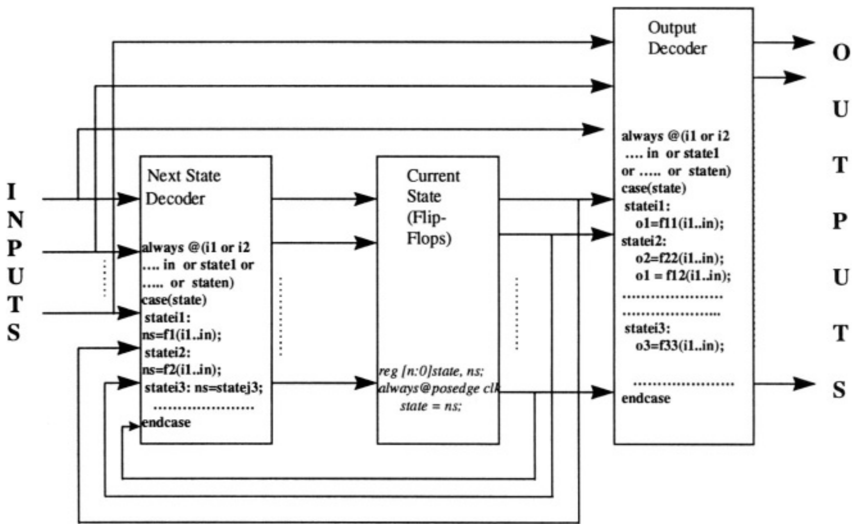


Figure 12-5. Modern View (Class A – Sagdeo Machine) of a Sequential Design

The datapath synthesis will be done using:

- RTL and gate level descriptions
- Some simple behavioral logic is supported

State Machine Synthesis will be done from state machine descriptions via simple behavioral descriptions (1 state machine per module)

12.4 Examples

Adder(Combinational Logic example):

```
module m(out,in 1,in2);
      output [4:0] out;
      input [3:0] inl, in2;

      assign out = inl + in2;
endmodule
```

Example 12-1. Synthesizable combinational adder.

In the Example 12-1 above, an adder is described using rtl construct—the continuous assignment. This will be synthesized into a combinational adder of gates like the one in Chapter 5.

```
     // Multiplexor(Combinational Logic example):
`define DATA0 0
`define DATA1 1
`define DATA2 2
`define DATA3 3

module mux42(out, data, sel);

input [3:0] data;
input [1:0] sel;
output out;
      reg out;

always @(data or sel)
      case (sel)

            `DATA0 : out = data[0];
            `DATA1 : out = data[1];
            `DATA2 : out = data[2];
            `DATA3 : out = data[3];

      endcase
endmodule
```

Example 12-2. Synthesizable combinational multiplexor.

In the Example 12-2 above, a combinational multiplexer is described using behavioral modeling with case construct as the key construct.

```
// State Machine Example - Traffic light controller

`define GREEN  0
`define YELLOW 1
`define RED   2

module TLC(R, G, Y, clock,  reset);
            output R,  Y, G;   // Red, Yellow, Green Signal lines
            input clock, reset;
            reg next_state;
            reg R, Y, G;
            always @(posedge clock)
            begin
                  case (next_state)

            `GREEN:
                  begin
                        next_state = `YELLOW;
                  G = 1;
                  R = 0;
                  end

                  `YELLOW: begin
                  next_state = `RED;
                  Y = 1 ;
                  G = 0;
            end

                        `RED: begin
                        next_state = `GREEN;
                  R = 1;
                  Y = 0;
            end
      endcase
   end

always @(reset)
      next_state = `RED;
endmodule
```

Example 12-3. Synthesizable sequential design – traffic light controller.

In the Example 12-3 above, a traffic light controller state machine is described.

```
`define TAG_SIZE  6
// 256 K Cache

`define CACHE_SIZE 2*1024
`define ADDR_SIZE  17
`define IDLE 0
```

```
`define READ 1
`define WRITE 2
`define READ_MISS 3
`define READ_CACHE 4
`define WRITE_MISS 5
`define WRITE_CACHE 6
`define READ_WRITE_DIRTY 7
`define WRITE_WRITE_DIRTY 8

`define READ_MISS1 9
`define READ_MISS2 10
`define WRITE_MEM 11

`define DATA_SIZE 64

// This is limited by maximum size in this simulator
`define MEM_SIZE 128*1024

`define CP 100
`define CACHE_DRV 1
`define NONCACHE_DRV 0

module comparator(in1, in2, match);
    input [TAG_SIZE-1:0] in1, in2;
    output match;

    assign match = (in1 == in2);
endmodule

module cacheControl(procRead, procWrite, match, valid, read, write, memWrite,
                    memRead,dataOutSel, dataInSel, clock, reset);
    input procRead, procWrite, match, valid, clock, reset;
    output memWrite, memRead,dataOutSel, dataInSel, read, write;
    reg memWrite, memRead,dataOutSel, dataInSel, read, write;
    reg [7:0] state, nextState;

    always @(posedge clock)
        state = reset ? `IDLE : nextState;

    always @(state or procWrite or procWrite or match or valid)
    case(state)

    `IDLE: if (procRead)
                nextState = `READ;
            else if (procWrite)
                nextState = `WRITE;
            else
                nextState = `IDLE;

    `READ : if (match && valid)
```

```
                nextState = `IDLE;
            else
                nextState = `READ_MISS1;
    `READ_MISS1:/*   Read from main memory takes 2 cycles */
            nextState = `READ_MISS2;

    `READ_MISS2:      nextState = `IDLE;

    ` WRITE: /* Check principles of write-hit from Mips documentation*/
            /* Current plan : always save in cache */
            /* Outputs here consist of writing to cache */
            nextState = `WRITE_MEM;

    `WRITE_MEM:
            nextState = `IDLE;

    endcase

    /* Output combinational logic from controller */
    /* The signals in the output list are
    memRead, memWrite, dataInSel, dataOutSel, read, write
    */
    always @(state)
        case (state)

        `READ:
            begin
             read = 1;
             dataOutSel = ~(match & valid);
            end

        `READ_MISS1:
            begin
            memRead = 1;
            dataInSel= 1;
            end

        `READ_MISS2:
            begin
            memRead = #1 0;
            write = 1;
            write = #1 0;
            end

        `WRITE:
            begin
            memWrite = 1;
            write = 1;
            dataInSel = 0;
            end
```

```
        `WRITE_MEM:
            begin
            memWrite = 0;
            write = 0;
            memWrite = #1 0;
            dataInSel = #l 1;
            end
        endcase
    endmodule

/* This mux will output data onto processor for read */
module dataMux(data0, datal, dataSel, outData);
    input [`DATA_SIZE-1:0] data0, data1;
    input dataSel;
    output [`DATA_SIZE-1:0] outData;
        // cache controller must generate a 1 for read_cache and
        // 0 for read_miss states for dataSel lines.

        assign outData = dataSel ? data0: data1;

endmodule
```

Example 12-4. ***Synthesizable parts of the cache system - cache control (sequential), mux, compare.***

12.5 Exercises

1. Classify the following Verilog statements as synthesizable and non-synthesizable in the logic synthesis subset.
   (i) disable main_loop;
   (ii) x = x * y;
   (iii) if (count == 8) red = 1;
   (iv) wait((clk ==1) and (state == `YELLOW);
   (v) $log2(in, out); /* $log2 is a task defined via PLI */
2. Write a synthesizable design in Verilog for state machine for a coin-operated sodapop machine with the following characteristics.

   Coke costs 25c;
   Fruit Punch Costs 30c;
   Orange Juice costs 35c.

   Assume that the inputs are Quarters, Nickels and Dimes followed by one of the three choices above. Assume that exact change is always input. Outputs are the three signals (indicating what comes out of the machine).
3. In the typical design flow of Figure 12-1, synthesizer is invoked before higher level simulation. Why is this done?

4. Run the four examples via the synthesis tool provided with the book and obtain the characteristics of the gate-level circuit.

13 VERILOG SUBSET FOR LOGIC SYNTHESIS

13.1 Introduction

A design methodology, a description style and a set of constructs define the subset of Verilog HDL that is synthesizable via logic synthesis. The design methodology has been discussed in the various design flows as in Figures 12-1, 12-2, and 12-3. Some details of these will be discussed during the discussion of the subset in the following few sections. The description style has been described in the previous sections as logic synthesis view as in Figure 12-4 and 12-5. Some aspects of these are discussed in this section in the following paragraphs.

In describing the combinational logic, the three styles and corresponding Verilog constructs can be clearly used with the restrictions of no feedback or storage. While describing the sequential logic part of the design, in the logic synthesis view of the design in Chapter 12 [Figure 12-4 and 12-5], the sequential logic corresponds to the states only. These are typically synthesized into a set of flip-flops. There are two possible representations of this—structural or rtl.

Structurally, one can directly instantiate registers into a Verilog description, selecting from any element in an ASIC library. Clocking schemes can be arbitrarily complex. You can choose between a flip-flop and a latch-based architecture. This forms the structural style of describing states. In this method, the Verilog description is specific to a given technology because you choose structural elements from that technology library. However, you can isolate the portion of your design with directly instantiated registers as a separate component (module), then connect it to the rest of the design. The description is more difficult to write.

Alternately, you can use the state machine using the model in Figure 12-5 where states are specified as reg with clock edges determining the changes in state. The advantages to this approach directly counter the disadvantages of the previous approach. With register inference, the Verilog description is much easier to write, and it is technology independent. This method allows the synthesizer to select the

type of component inferred based on constraints. As seen in the next chapter, a mixed style whereby the flip-flops are inferred in your model but they are technology independently described can also provide a method of generating sequential logic.

The preceding paragraph's description of combinational and sequential logic descriptions leads to a set of constructs in Verilog that can be synthesized or a logic synthesis subset. This is described in the remaining sections of this chapter and in the next chapter which focuses on special elements like registers, latches, 3-states and multiplexor descriptions.

Some special sequential circuits beyond state machines can be synthesized either as direct mapping to presence of such circuits in the target technology or by translation or by synthesis. One such example is that of counters. Such elements are also described in the next chapter.

Note: In using constructs outside this subset, the synthesizers deal with this in two ways: either to ignore the construct and continue synthesis process or report an error in input syntax and abort the synthesis process.

During simulation and top-down design process, it is necessary to use constructs such as system tasks and functions as well as provide more details than are usable in synthesis (for example, delays). Such constructs are ignored by the synthesizers. Such constructs do not describe the core functionality of the design. On the other hand, there are constructs that describe functionality. An example of this will be using arbitrary events based model like that of a microprocessor when instructions are modeled. Such constructs are flagged as errors during analysis for synthesis. A complete formal description of this classification is provided in Appendix B.

13.2 Structural Descriptions – Modules

The following structural constructs are synthesizable.

- Modules
- Macromodules
- Port Definitions
- Module Instantiations
- Parameters

In section 3.10, we saw the Verilog high-level structural constructs. As seen in he above list, all of those constructs are supported for synthesis. Thus, the hierarchical top-down or bottom descriptions of modules is supported in synthesis. Module definitions are fully supported. The port definitions are fully supported in all the forms—the normal, and the named, with support for expressions at the port declarations including concatenations. The module instantiations can be used freely in synthesizable descriptions. Both types of port associations—positional and named notations are supported. The parameter support depends partially on synthesis implementations—the defparam statement is not supported and parametrization may depend on intervention with the later stages in synthesis and with the libraries of

design being analyzed. section 13.7 describes the usage of parametrized designs during synthesis.

The synthesis typically preserves the module boundaries, although controls are available to merge modules and optimize across modules. The partitioning problems is still best solved by the designers.

Macromodules

The macromodule construct makes simulation more efficient by merging the macromodule definition with the definition of the calling (parent) module. However, the synthesizer treats the macromodule construct as a module construct. Whether you use module or macromodule, the synthesis process, the hierarchy it creates, and its end result are the same. Example 3-2 shows how to use the macromodule construct.

The support for the other constructs is same as that discussed before in Chapter 3 with the exception of the parameters.

13.3 Declarations – Overview

The Synthesizer front end recognizes the following Verilog statements and constructs when they are used in a Verilog module:

- parameter declarations
- wire, wand, wor, tri, supply0, and supply1 declarations
- reg declarations
- input declarations
- output declarations
- inout declarations

13.4 Module Items – Overview

- Continuous assignments
- Module instantiations
- Gate instantiations
- Function definitions
- always blocks
- task statements

13.5 Synthesizing Net Types

Synthesizable wire types include wire, wand, wor, tri, supply0, and supply1. These have been explained in section 2.5. You can define an optional range for all the data types presented in this section. You can assign a delay value in a wire declaration, and you can use the Verilog keywords scalared and vectored for simulation. The synthesizer accepts the syntax of these constructs, but both these constructs are ignored when the circuit is synthesized.

Note: You can use delay information for modeling, but Synthesizer ignores delay information. If the functionality of your circuit depends on the delay information, Synthesizer may create logic whose behavior does not agree with the behavior of the simulated circuit.

In the Verilog language, an undriven wire defaults to a value of Z (high impedance). However, synthesis leaves undriven wires unconnected. Multiple connections or assignments to a wire simply short the wires together.

13.6 Continuous Assignments

Synthesis support includes both methods of continuous assignments—the implicit and the explicit continuous assignments. The left side of a continuous assignment can be:

- A wire, wand, wor, or tri.
- One or more bits selected from a vector.
- A concatenation of any of these.

The right side of the continuous assignment statement can be any supported Verilog operator or any arbitrary expression that uses previously declared variables and functions.

Verilog allows you to assign drive strength for each continuous assignment statement. Synthesizer accepts drive strength, but it does not affect the synthesis of the circuit. Thus, when using drive strength in your Verilog source, this can be a factor. Assignments are done bit-wise, with the low bit on the right side assigned to the low bit on the left side. If the number of bits on the right side is greater than the number on the left side, the high-order bits on the right side are discarded. If the number of bits on the left side is greater than the number on the right side, operands on the right side are zero-extended

Typically combinational logic circuits can be built using this construct. Example 12-1 of adder is a typical example of this usage.

13.7 Module instantiations – Parametrized Designs

The Verilog language allows you to create parameterized designs by overriding parameter values in a module during instantiation. In Verilog, you can do this with the defparam statement or with the following syntax.

```
module_name #(parameter_value,parameter_value,...) instance_name (terminal_list)
```

Synthesizer does not support the defparam statement, but does support the syntax above. The module in Example 13-1 contains a parameter declaration.

```
module m(a,b,c);
        parameter width = 8;
        input [width-1:0] a,b;
        output [width-1:0] c;
```

```
        assign c = a & b;
endmodule
```

Example 13-1. Parametrized design.

In Example 13-1, the default value of the parameter width is 8, unless you override the value when the module is instantiated. When you change the value, you build a different version of your design. This type of design is called a parameterized design. Parameterized designs are read into dc_shell as templates with the read command. These designs are stored in an intermediate format so that they can be built with different (nondefault) parameter values when they are instantiated.

If you use parameters as constants that never change, do not read in your design as a template. One way to build a template into your design is by instantiating it in your Verilog code. Example 13-2 shows how to do this.

```
module param (a,b,c);
        input [3:0] a,b;
        output [3:0] c;
        foo #(4) U1(a,b,c);
        //instantiate foo
endmodule
```

Example 13-2. Instantiating a parameterized design in your Verilog code.

Example 13-2 instantiates the parameterized design, foo, which has one parameter, assigned the value 4. Since module foo is defined outside the scope of module param, errors such as port mismatches and invalid parameter assignments are not detected until the design is linked. When Synthesizer links module param, it searches for template foo in memory. If foo is found, it is automatically built with the specified parameters. Synthesizer checks that foo has at least one parameter and three ports, and that the bit-widths of the ports in foo match the bit-widths of ports a, b, and c. If template foo is not found, the link fails.

13.8 Structural Descriptions – Gate-Level Modeling

Verilog provides a number of basic logic gates that enable modeling at the gate level. Gate-level modeling is a special case of positional notation for module instantiation that uses a set of predefined module names. Synthesizer supports the following gate types:

and

nand

or

nor

xor

xnor

buf

not

tran

The details of these and other gate type is given in section 6.2 on Gates. Synthesizers typically optimize boolean functions and the above set of gates represents the set of boolean functions. In addition tri-stating is also supported using higher-level constructs than transistors—either gates or rtl/behavioral descriptions.

Connection lists for instantiations of a gate-level model use positional notation. In the connection lists for and, nand, or, nor, xor, and xnor gates, the first terminal connects to the output of the gate, and the remaining terminals connect to the inputs of the gate. You can build arbitrarily wide logic gates with as many inputs as you want.

Connection lists for buf, tran, and not gates also use positional notation. You can have as many outputs as you want, followed by only one input. Each terminal in a gate-level instantiation can be a 1-bit expression or signal. In gate-level modeling, instance names are optional. Drive strengths and delays are allowed, but they are ignored in synthesis. Example 13-3 shows two gate-level instantiations.

```
not (out1,in);
or o4(out2, in1, in2, in3, in4);
xor x3(out3, in1, in2, in3);
```

Example 13-3. Gate-level instantiations.

Note: Delay options for gate primitives are parsed but ignored by Synthesizer. Because a synthesizer typically ignores the delay information, it may create logic whose behavior does not agree with the simulated behavior of the circuit.

Three-State Buffers

Synthesizer supports the following gate types for instantiation of three-state gates:

bufif0 (active low enable line)

bufif1 (active high enable line)

notif0 (active low enable line; output inverted)

notif1 (active high enable line; output inverted)

Connection lists for bufif and notif gates use positional notation. Specify the order of the terminals as follows: The first terminal connects to the output of the gate. The second terminal connects to the input of the gate. The third terminal connects to the control line. Example 13-4 shows a three-state gate instantiation with an active high enable and no inverted output.

```
module three_state (in1,out1,cntrl1);
        input in1,cntrl1;
        output out1;
```

```
        bufif1 (out1,in1,cntrl1);
endmodule
```

Example 13-4. Three-state gate instantiation.

13.9 Expressions

In Verilog, expressions consist of a single operand or multiple operands separated by operators. Use expressions where a value is required in Verilog. The expressions are described in detail in section 3.4.4. Here we discuss the expressions that are supported under synthesis.

Operand Types

Synthesis does not support real numbers or time or event types. All operations on bits and reals are supported for synthesis. Thus the supported types include:

numbers

wires and regs

bit-selects

part-selects

function calls

Constant Valued Expressions

These are optimized during synthesis and their values are computed at compile-time and no corresponding hardware is generated for these. Thus, it is beneficial to add as many constant valued expressions as possible. Usage of parameters and constants rather than reg types will help in such situations. Constants are also propagated to see if the next level of expressions also is a constant valued expressions indirectly specified as such.

Handling Comparisons to X or Z

Comparisons to an X or a Z are always ignored. If your code contains a comparison to an X or a Z, a warning message is displayed indicating that the comparison is always evaluated to false, which might cause simulation to disagree with synthesis.

For example, the variable B in the following code (from a file called test2.v) is always assigned to the value 1, since the comparison to X is ignored.

```
always begin
        if(in1==1'bx)
                out = 0;
        else
                out= 1;
        end
```

Example 13-5. Synthesis and comparisons to X.

When a synthesizer reads this code, the following warning message is generated. Warning: Comparisons to a "don't care" are treated as always being false in routine test 2 line 10 in file `test2.v'.

This may cause simulation to disagree with synthesis. For an alternate method of handling comparisons to X or Z, one can direct the synthesizer to not synthesize this but translate this by hand. Directives like the // synthesis translate_off directive // synthesis translate_on directives exist for this operation. Inserting these directives might cause simulation to disagree with synthesis.

13.10 Behavioral Modeling for Synthesis

Using Behavioral Constructs

Behavioral statements can be used to describe combinational circuitry or sequential logic. Combinational logic is described using always blocks or continuous assignments with functions providing the behavioral descriptions.

To describe combinational logic, you write a sequence of statements and operators to generate the outputs you want. For example, suppose the + operator is not supported, and you want to create a combinational adder where the bit-by-bit structure is determined by the functional description.. The easiest way to describe this circuit is as a cascade of full adders, as in Example below. The example has four full adders, with each adder following the one before. From this description, Synthesizer generates a fully combinational adder.

```
module f_add(add, i1, i2);
        input [3:0] i1, i2;
        output [4:0] add;
        function [4:0] FourBitAdd;
                input [3:0] a, b;
                  reg c;
                  integer i;
                  begin
                    c = 0;
                    for (i = 0; i <= 3; i = i + 1) begin
                      FourBitAdd[i] = a[i]^ b[i]^ c;
                      c = a[i] & b[i] | a[i] & c | b[i] & c;
                   end
                   adder[4] = c;
                  end
        endfunction

        assign add = adder(i1, i2);
    endmodule
```

Example 13-6. Synthesizable combinational 4-bit adder using functions and continuous assignments.

The Examples 12-2 containing a combinational multiplexor described behaviorally. It is reproduced below:

```
// Multiplexor(Combinational Logic example):

  `define DATA0 0
  `define DATA1 1
  `define DATA2 2
  `define DATA3 3


  module mux42(out, data, sel);

      input [3:0] data;
          input [1:0] sel;
      output out;
              reg out;
      reg [3:0] data;
          reg [1:0] sel;

      always @(data or sel)
      case (sel)

          `DATA0 : out = data[0];
          `DATA1 : out = data[1];
          `DATA2 : out = data[2];
          `DATA3 : out = data[3];

      endcase
  endmodule
```

Example 13-7. Synthesizable combinational multiplexor.

The functions can also be used with the always statement. For example, the continuous assignment in the ripple carry adder above can be replaced with the following always statement:

```
always @(i1 or i2)
    add = adder(i1,i2);
```

Thus, descriptions with functions can be seen as a clear way of describing combinational logic behaviorally while describing the rules of writing combinational logic outside of functions will be a little more involved as seen in the following pages.

Any feedback is considered sequential but there are certain simple situations where a synthesis tool is able to determine whether the feedback results in combinational or sequential logic. Consider the following situation:

```
        out = b;
        if (var)
                out = out + a;
```

Example 13-8. A behavioral description with feedback that is not real.

In this case, var1 gets a value of (a+b) if var2 is non-zero. Thus, there is no real feedback. Thus synthesizer determines values assigned to a variable in each state and then checks for feedback for the complete expression.

13.11 Function Declarations

Function declarations are one of the two primary methods for describing combinational logic. The other method is the always block, described later in this chapter. You must declare and use Verilog functions within a module. You can call functions from the structural part of a Verilog description by using them in a continuous assignment statement, or as a terminal in a module instantiation. You can also call functions from other functions or from always blocks. Synthesizer supports the following Verilog data declarations in behavioral descriptions and in functions:

input declarations

reg declarations

memory declarations

parameter declarations

integer declarations

13.11.1 Data Declarations in Functions – Reg, Input, Memory, Parameter, and Integers

reg Declarations

The synthesizable Verilog language allows you to assign a value to a reg variable only within a function or an always block. In the Verilog simulator, reg variables can hold state information. A reg can hold its value across separate calls to a function. In some cases, Synthesizer emulates this behavior by inserting flow-through latches. In other cases, this behavior is emulated without a latch.

Memory Declarations

The memory constructs are converted to a bank of registers in synthesis. Sample memory declarations are shown in Example 13-9 below:

```
reg [31:0] bus;
reg [31:0] memory [255:0];
```

Example 13-9. Memory declarations – synthesized as bank of registers.

In Example 13-9, word is a 32-bit register and memory is an array of 256 registers of 32-bit width. You can index the memory to access individual registers, but you cannot access individual bits of a memory directly. Instead, you must copy the appropriate register into a one-dimensional register. For example, the following code accesses the 1st bit of the 1st location of memory.

```
bus = memory [0];
bit = bus [0];
```

Example 13-10. Accessing bits within a memory block.

Parameter Declarations

You can declare parameter variables as local to a function. However, you cannot use a local variable outside that function. Parameter declarations in a function are identical to parameter declarations in a module. The function in Example 13-11 contains a parameter declaration. This is used as a constant expression. These parameters cannot be changed like the parameters in the module and are only useful as a symbolic representation of constants.

```
function carry
        parameter width = 8;
        input [width-l:0] a,b;
        reg [width:0] add;
         add = 0+(a + b);
        carry = add[width];
endfunction
```

Example 13-11. Parameter declaration in a function.

Integer Declarations

Integer variables are local or global variables that hold numeric values You can declare integer variables locally at the function level or globally at the module level. The default size for integers is 32 bits. Synthesizers determines bit-widths, except in the case of a don't-care resulting during compile. Example below illustrates integer declarations.

```
integer a;        //single 32 bit integer
integer b, c;     //two integer declarations
```

13.12 Behavioral Statements Support for Logic Synthesis – Overview

The behavioral statements supported are:

- procedural assignments
- RTL assignments
- begin … end block statements

- if... else statements
- case, casex, and casez statements
- for loops
- while loops
- forever loops
- disable statements

Full Verilog HDL support for these constructs was discussed in Chapter 5. The following paragraphs provide details of synthesizable code.

13.13 Procedural Assignments

Delays are completely ignored. The blocking and non-blocking will produce same results for combinational logic; however for sequential logic, difference can be seen as below:

Blocking Assignment

```
always @(posedge clk)
begin
        rega = data;
        regb = rega;
end
```

In this case the synthesized netlist consists of:

```
dff a (rega, data, clk, .....);
dff b (regb, data, clk, ...);
```

Non-Blocking Assignment

```
always @(posedge clk)
 begin
        rega <= data;
        regb <= rega;
end
```

In this case the synthesized netlist consists of :

```
dff a (rega, data, clk,.....);
dff b (regb, rega, clk, ...);
```

The following restrictions apply to assignments: You cannot use procedural assignments with blocking and non-blocking assignments at the same time. The following example is not allowed.

```
reg b,c;
always
 begin
     // non-blocking assignment
         b <= #4a;
          //blocking assignment
         c = #3b;
end
```

13.14 if Statement

Synthesis supports if—else statements and these primarily create combinational multiplexors. The if...else statement may cause registers to be synthesized. Registers are synthesized when you do not assign a value to the same reg in all branches of a conditional construct. The latch creation in if statements is explained in section 13.15. Synthesizer synthesizes multiplexer logic (or similar select logic) from a single if statement. The conditional expression in an if statement is synthesized as a control signal to a multiplexor which determines the appropriate path through the multiplexor. For example, the statements in Example 13-12 below create multiplexor logic controlled by c and places either a or b in the variable x.

```
if(c)
        x = a;
else
        x = b;
```

Example 13-12. ***if statement that synthesizes multiplexor logic.***

Example 13-13 below illustrates how if and else can be used to create an arbitrarily long if... else if... else structure.

```
if (instruction == ADD)
   begin
      carry_in = 0;
      complement_arg = 0;
   end
else if (instruction == SUB)
   begin
      carry_in = 1;
      complement_arg = 1;
   end
else
   illegal_instruction = 1;
```

Example 13-13. ***if … else if… else structure with several mutually exclusive conditions.***

Example 13-14 below shows how to use nested if and else statements.

```
if(select[1])
   begin
      if (select[0]) out = in[3];
                  else out = in[2];
   end
else
   begin
      if (select[0])
             out = in[1];
       else  out = in[0];
   end
```

Example 13-14. Nested if and else statements.

13.15 Conditional Assignments

Synthesizer may synthesize a latch for a conditionally assigned variable. A variable is conditionally assigned if there is a path that does not explicitly assign a value to that variable. See the section on "Latch Inference" for more information. In Example 13-15 below, the variable value is conditionally driven. If c is not true, value is not assigned and retains its previous value.

```
always
begin
        if ( cond )
        begin
                value = in;
        end
        out = value; //causes a latch to be synthesized for value
end
```

Example 13-15. Synthesizing a latch for a conditionally-driven variable.

13.16 Case Statements

Full Case and Parallel Case

Synthesizer automatically determines whether a case statement is full or parallel. A case statement is called full case if all possible branches are specified. If you do not specify all possible branches, but you know that one or more branches can never occur, you can use a directive // Synthesis_full_case. To avoid creating latches, assign a value to all variables or use the “default” statement that automatically assigns the value specified in that statement to all unstated cases.

Synthesizers provide ways of controlling this using directives like // Synthesis_full_case (Appendix C). This, just like the default case provides the

method of creating multiplexors without feedback or latches for unstated cases. However, coding this in Verilog with default statement is much better method since this assures compatibility with simulation and timing analysis and other tools that do not recognize the compiler directives.

Synthesizer synthesizes optimal logic for the control signals of a case statement. If Synthesizer cannot statically determine that branches are parallel, it synthesizes hardware that includes a priority encoder. Thus, it is a good idea to specify the case options as constant expressions that are mutually exclusive. If Synthesizer can determine that no cases overlap (parallel case), a multiplexor is synthesized, since a priority encoder is not necessary. You can also declare a case statement as parallel case with the //synthesis_parallel_case directive (see appendix C).

Example 13-16 below does not result in either a latch or a priority encoder.

```
input [1:0] a;
always @(cntrl or data1 or data2 or data3 or data4)
begin
case (cntrl)
          2'b11:
             out = data1;
          2'b10:
             out = data2;
          2'b01:
             out = data3 ;
          2'b00:
             out = data4;
endcase
end
```

Example 13-16. A case statement that is both full and parallel.

Example 13-17 below shows a case statement that is missing branches for the cases 2'b01 and 2'b10. Example 13-17 below infers a latch for out.

```
input [1:0] cntrl;
always @(cntrl or data1 or data2) begin
          case (cntrl)
          2'b11:
             out = data1 ;
          2'00:
             out = data2;
          endcase
end
```

Example 13-17. A non-full but parallel case statement – synthesized with latches

The case statement in below is not parallel or full because the values of inputs w and x cannot be determined. However, if you know that only one of the inputs equals 2'b11 at a given time, you can use the // Synthesis parallel_case directive to avoid

synthesizing a priority encoder. If you know that either w or x always equals 2'b11 (a situation known as a one-branch tree), you can use the "// Synthesis full_case" directive to avoid synthesizing a latch.

```
always @( in1 or in2)
begin
        case(2'b11)
        in1:
           out = 10;
        in2:
           out = 01 ;
        endcase
end
```

Example 13-18. ***A case statement that is neither parallel, nor full.***

13.17 For Loops

The for loop repeatedly executes a single statement or block of statements. The repetitions are performed over a range determined by the range expressions assigned to an index. Two range expressions are used in each for loop: low_range and high_range. Note that in the syntax lines that follow, high_range is greater than or equal to low_range. Synthesizer recognizes both incrementing and decrementing loops. The statement to be duplicated is surrounded by begin and end statements.

Note: Synthesizer allows four syntax forms for a for loop.

They are:

```
for (index= low_range;index < high_range;index= index + step)

for (index= high_range;index > low_range;index= index - step)

for (index= low_range;index <= high_range;index= index + step)

for (index= high_range;index >= low_range;index= index - step)
```

Example 13-19 below shows a simple for loop.

```
for(i = 0; i <= 31; i = i+1)
begin
   sum[i] = in1[i] ^ in2[i] ^ cin;
   cout = in1[i] & in2[i] | a[i] & cin |
        in2[i] & cin;
end
```

Example 13-19. ***Example of synthesizable simple for loop.***

Note that for loops can be nested, as shown in Example 13-20 below.

```
for (i = 6; i >=0; i = i - 1)
   for (j = 0; j <= i; j =j + 1)
      if (value[j] > value[j+1]) begin
         temp = value[j+1];
         value[j+l] = value[j];
         value[j] = temp;
      end
```

Example 13-20. Example of nested for loops that can be synthesized.

You can use for loops as duplicating statements. Example 13-20 below shows a for loop that is expanded into its longhand equivalent.

```
for (i=0; i < 8; i=i+1)
   out[i] = in1[i] & in2[7-i];
```

Example 13-21. For loop used for duplicating statements.

```
out[0] = in1[0] & in2[7];
out[1] = in1[1] & in2[6];
out[2] = in1[2] & in2[5];
out[3] = in1[3] & in2[4];
out[4] = in1[4] & in2[3];
out[5] = in1[5] & in2[2];
out[6] = in1[6] & in2[1];
out[7] = in1[7] & in2[0];
```

Example 13-22. Equivalent expansion of for loop in Example 13-20.

While Loops

The while loop executes a statement until the controlling expression evaluates to false. A while loop creates a conditional branch that must be broken by an @ (posedge clock) or @ (negedge clock) statement to prevent combinational feedback. Synthesizer supports while loops, if you insert an @ (posedge clock) or @ (negedge clock) expression in every path through the loop. Example below shows an unsupported while loop that has no event-expression. This follows with the policy of feedback going via states, that is the only storage must be into states.

```
always
        while (in1 < in2)
                out = in1 + in3;
```

Example 13-23. Unsupported while loop.

If you add @ (posedge clock) expressions after the while loop in Example 13-23 above, you get the supported version shown in example below:

```
always
begin @ (posedge clock)
        while (x < y)
        begin
                @ (posedge clock);
                        x = x + z;
        end
end;
```

Example 13-24. Supported while loops.

13.18 Forever Loops

Infinite loops in Verilog use the keyword forever. You must break up an infinite loop with an @ (posedge clock) or @ (negedge clock) expression to prevent combinational feedback, as shown in Example 13-27 below:

```
always
forever
begin
        @ (posedge clock);
                out = out + in;
end
```

Example 13-25. Supported forever loop.

You can use forever loops with a disable statement to implement synchronous resets for flip-flops. The disable statement is described in the next section Example 13-27 shows usage of forever combined with disable to specify synchronous reset condition.

13.19 Disable Statements

Synthesizer supports the disable statement when you use it in named blocks. When a disable statement is executed, it causes the named block to terminate. A comparator description that uses disable is shown in Example 13-26 below.

```
module equal(out, in1, in2);
    output out;
    reg out;
    input [7:0] in1, in2;
    reg [3:0] I;
    always @ (in)
            begin: loop
                    for (i = 7; i >= 0; i = i - 1)
                    begin
```

```
                out = in1[I]!=in2[I];
                    if (out)
                    //Inequality found; comparison is done and time to stop
            looping
                        disable loop;
            end
        end
    endmodule
```

Example 13-26. Loop exiting using disable – a comparator model.

Note that the example above describes a combinational comparator. Although the description appears sequential, the generated logic runs in a single clock cycle. You can also use a disable statement to implement a synchronous reset, as shown in Example 13-27 below:

```
always
forever
begin: reset_label
 @ (posedge clock);
        if (reset) disable reset_label;
        task1;;
 @ (posedge clock);
        if (reset) disable reset_label;
        task2;
end
```

Example 13-27. Synchronous reset of state register using disable in a forever loop.

The disable statement in Example 13-27 above causes the block reset_label to terminate immediately and return to the beginning of the block. Therefore, the first state in the loop is synthesized as the reset state.

13.20 Task Statements

Task statements are similar to functions in Verilog, except they can have output and inout ports. You can use the task statement to structure your Verilog code so that a portion of code can be reused.

In Verilog, tasks can have timing controls and they can take a nonzero time to return. However, Synthesizer ignores all timing controls, so synthesis might disagree with simulation if the timing controls are critical to the function of the circuit. Example 13-28 below shows how a task construct is used to define an adder function.

```
module task_example (a,b,c);
        input [7:0] a,b;
        output [7:0] c;
        reg [7:0] c;
```

```
        task adder;
                input [7:0] a,b;
                output [7:0] adder;
                reg c;
                integer i;
                begin
                        c = 0;
                        for (i = 0; i <= 7; i = i+1) begin
                                adder[i] = a[i] ^ b[i] ^ c;
                                c = (a[i] & b[i]) | (a[i] & c) | (b[i] & c);
                        end
                end
        endtask

        always
                adder (a,b,c);  //c is a reg
endmodule
```

Example 13-28. Using tasks to describe synthesizable combinational logic.

13.21 Always Blocks

An always block can imply latches or flip-flops, or it can specify purely combinational logic. An always block can contain logic triggered in response to a change in a level or the rising or falling edge of a signal. The syntax of an always block is:

```
always @ ( event-expression [or event-expression*])   begin
   ... statements ...
end
```

The event-expression declares the triggers, or timing controls. The word or groups several triggers. The Verilog language specifies that if triggers in the event-expression occur, the block is executed. Only one trigger in a group of triggers needs to occur for the block to be executed. However, synthesizer ignores the event-expression unless it is a synchronous trigger that infers a register. In the next chapter, we discuss the details of this issue. A simple example of an always block with triggers is:

```
always @ (in1 or in2 or in3 or in4)
begin
        out = in1 & in2 | ~in3 ^ in4
end
```

Example 13-29. A simple always block describing synthesizable combinational logic.

In Example 13-29 above, a, b, and c are asynchronous triggers. If any.triggers change, the simulator resimulates the always block and recalculates the value of f. Synthesizer ignores the triggers in this example since they are not synchronous. However, you must indicate all variables that are read in the always block as triggers

For a synchronous always block, Synthesizer does not require all variables to be listed. An always block is triggered by **any of the following types of event-expressions:**

1. Event expression entailing change in the value of a reg or a net:

```
always @ ( identifier ) begin
  ... statements ...
end
```

Example 13-30. Event expression indicating sensitivity or input to a block.

In the example above, the trigger simply indicates that the identifier is an input to the block synthesized from the statements within the begin-end block above.

2. The rising edge of a clock – As shown in the Example 13-31, event-expressions can model rising clock-edges.

```
always @ (posedge event) begin
  ... statements ...
end
```

Example 13-31. Event expression indicating rising edge of clock.

3. The falling edge of a clock – As shown in the Example 13-32, event-expressions can model falling clock edges.

```
always @ ( negedge event )
begin
  ... statements ...
end
```

Example 13-32. Falling clock edge modeled in event-expression.

4. A clock combined with an asynchronous reset condition.

```
always @ ( posedge CLOCK or negedge reset )
begin
  if !reset begin
    ... statements ...
  end
  else begin
    ... statements ...
  end
end
```

Example 13-33. Event expression modeling clock-edge combined with resetting condition.

When the event-expression does not contain posedge or negedge, combinational logic (no registers) is usually generated, although flow-through latches can be generated as seen before for if and case statements.

Functions can not contain event controls in Verilog. Tasks support such modeling in Verilog but synthesis does not support such modeling. The event controls must be contained in the always blocks as in all the examples above.

13.22 Incomplete Event (Sensitivity) Specification

An always block can be misinterpreted if you do not list all signals entering an always block in the event specification prior to the evaluation. As expected, synthesizer builds a 4-input gate-logic for the description in the example below.

```
always @ (in1 or in2 or in3)
begin
        out = in1 & in2 | ~in3 ^ in4
end
```

Example 13-34. Incomplete event list – simulation and synthesis may mismatch.

When this description is simulated, f is not recalculated when c changes because c is not listed in the event-expression. The simulated behavior is not that of a 3-input AND gate. The simulated behavior of the description in example below is correct because it includes all signals in the event-expression.

```
always @ (in1 or in2 or in3 or in4)
 begin
        out = in1 & in2 | ~in3 ^ in4
end
```

Example 13-35. Complete event list.

In some cases, you cannot list all signals in the event specification. Example below illustrates this situation. Here the behavior is like one of edge-sensitive flip-flop.

```
always @ (posedge c or posedge p)
if (p)
        out = data;
else
        out = resetval;
```

Example 13-36. Incomplete event list for edge-triggered flip-flop with asynchronous reset.

13.23 Exercises

1. What does a logic synthesizer do with the following constructs: delayed assignments, drive-strength, switch-level primitives, force statement, continuous assignments.
2. Run Example 3-27 in section 3.9 through the synthesis tool provided with the book. Compare the pre- and post-synthesis design simulation results.
3. Does a synthesis tool handle dynamic loops where the loop size is a variable?
4. What is the difference between if-else-if and case statement synthesis especially when the parallel-case and full-case directives are present?

14 SPECIAL CONSIDERATIONS IN SYNTHESIZING VERILOG

A synthesizer in general maps the logic into gate-level descriptions with boolean functions. A few special types of constructs are recognized at higher level and mapped into logic-blocks. These include registers (latches and flip-flops), multiplexers, and three-state devices. These are recognized from the source description using some modeling guidelines. This chapter discusses methods of inferring these special devices—different types of registers, multiplexers and three-state devices. This part of special considerations is discussed in sections 14.1 to 14.3.

A technique used in synthesis at higher levels of abstraction is known as resource sharing. This allows the same resource to be shared amongst different blocks of hardware functional units thereby optimizing the number of resources needed in the design. The special considerations that are done while modeling in Verilog for effective resource sharing is discussed in section 14.4 to 14.5.

14.1 Inferring Registers

14.1.1 Introduction

A register inference allows you to use sequential logic in your design descriptions and keep your designs technology independent. A register is a simple, one-bit memory device, either a latch or a flip-flop A latch is a level-sensitive memory device. A flip-flop is an edge-triggered memory device.

14.1.2 Latch Inference

Because variables can hold state over time in simulation, synthesizer needs to duplicate this condition in hardware. It does this by inserting a D-type flow-through latch. The latch allows a variable to hold its value (state) until that value is

reassigned. A variable must hold its state when its previous value may change, because of a condition in an if statement. When the condition is true, the value is reassigned. Since the condition might be false, the variable must be able to hold its state. Therefore, a latch is created to hold the previous value of the variable. For example:

```
module latch(out, clock, data);
        output out;
    input clock, data;

            always @ (clock or reset)
            begin
              if (clock) begin
                out = data;
              end
endmodule
```

Example 14-1. Creating a latch.

In Example 14-1 above, the variable out is not assigned a new value when clock is false. A latch is synthesized with its data input connected to out.

A latch may also be created when you use a case statement. For example, the code in Example 14-2 creates a latched binary-coded decimal (BCD) decoder.

```
module bcd_decode(b, d);
        input [3:0] b;
        output [9:0] d;
        reg [9:0] d;
        always @(b) begin
                case(b)
                        4'h0: d= 10'b1;
                        4'h1: d= 10'b10;
                        4'h2: d= 10'bl00;
                        4'h3: d= 10'b1000;
                        4'h4: d= 10'b10000;
                        4'h5: d= 10'b100000;
                        4'h6: d= 10'b1000000;
                        4'h7: d= 10'b10000000;
                        4'h8: d= 10'bl00000000;
                        4'h9: d= 10'bl000000000;
                    // default: d=10'b0; Uncommenting this line will stop
                    latch-creation
                endcase
        end
endmodule
```

Example 14-2. Creating a latch with a case statement.

The four bits from the input are passed to the case statement. The case statement assigns an appropriate binary expression of the input's decimal value to output decimal and latches that value in register decimal.

To avoid creating latches in combinational logic, the unintended feedback created by “no change” in value during recomputation in a case of if-then-else statement should be avoided. Assign a value to all options in a case or use the “default” statement that automatically assigns value to all unstated cases. The code in Example 14-2 does not create latches if the line before endcase is uncommented. Variables declared within a function do not hold their values over time because every time a function is called, its variables are reinitialized. Therefore, synthesizers do not infer latches for these variables. In Example 14-3, no latches are inferred. This is another reason to use functions for combinational logic

```
function my_func;
        input data, gate;
        reg state;
        begin
                if (gate)
                begin
                        state = data;
                end
                my_func = state;
        end
endfunction
```

Example 14-3. Variable declared within afunction-no latches inferred.

14.1.3 Simple Flip-Flop Inference

A flip-flop is implied when you use the posedge or negedge clock constructs in an always block, as shown below.

```
always @ (posedge clock) begin
   ...
end
```

A variable that is assigned a value in this always block is synthesized as a D-type edge-triggered flip-flop. The flip-flop is clocked on the rising (or falling) edge of the signal (clock) following the posedge (or negedge) keyword. With simple flip-flops (with no asynchronous set or reset), the block's event-expression may contain only one posedge (or negedge) statement, as shown in Example 14-4.

```
always @ (posedge clock)
 begin
        out = data;
end
```

Example 14-4. Creating an edge-triggered Dflip-flop.

This code is synthesized into a D-type positive-edge-triggered flip-flop with the data input connected to data, the output connected to out and the clock input connected to net clock.

14.1.4 Modeling Flip-Flops with Resets

The clock used for flip-flops is derived from the event-expression for the always block. As shown in Example 14-4 above, the clock-edge is modeled using the posedge clock event-expression. Resets can be modeled by adding test for posedge or negedge of reset signal anded with your clock.

These conditions also require corresponding posedge and negedge entries in the event-expression at the beginning of the always block. The last else clause has no condition to test. The clocked event is assumed. The flip-flop is clocked on the rising (falling) edge of the signal following the posedge (negedge) keyword in the event-expression at the beginning of the always block.

```
module dff_with_reset(data, clock, r, out);
        input clock, r, data;
        output out;
        reg out;

        always @ (posedge clock or posedge r)
         begin
                if(r)
                //asynchronous reset
                        out = 0;
                else
                //posedge clock is assumed
                        out = data;
        end
endmodule
```

Example 14-5. Flip-flop with asynchronous reset.

```
module dff_with_reset(data, clock, r, out);
        input clock, r, data;
        output out;
        reg out;

        always @ (posedge clock)
         begin
                if(r)
                //synchronous reset
                        out = 0;
                else
                //posedge clock is assumed
                        out = data;
```

```
        end
endmodule
```

Example 14-6. Flip-flop with synchronous reset.

14.1.5 Synthesis Checks During Register Inference

During synthesis of sequential devices, a particular design will follow a set of rules for the registers used in that design. Synthesizers can assist the designer in checking the design rules governing these devices. Several of these rules apply to the set-reset signals and their relationship to each other and other signals. Typically while specifying the conditions under which a register is set or reset, synchronous and asynchronous behavior of these is maintained throughout a design or portion of the design. This can be checked by the synthesizer by the following ways.

Check asynchronous nature of set and reset for all registers in the design

- // Synthesis asynchronous_set_reset Check asynchronous nature of set and reset for all registers in the design
- // Synthesis synchronous_set_reset

Another aid provided in synthesizing registers consists of setting the precedence of set and reset signals. Normally, designers know that these two signals have the same priority. However, in a Verilog HDL model of these, a priority encoder can be easily generated since one has to specify one of these signals before the other. Avoiding generation of these is done using directives -// Synthesis one_hot and // Synthesis one_cold.

14.1.5.1 asynchronous_set_reset – The asynchronous_set_reset check causes Synthesizer to check specified objects for the asynchronous set or reset of a latch or flip-flop The syntax of asynchronous_set_reset is:

```
// Synthesis asynchronous_set_reset "object_name,..."
    module m(r, s, data, control, outl, out2);
        input r, s, control, data;
        output outl, out2;
        // Synthesis async_set_reset "reset, set"
        reg y, t;

        always @ (reset or set)
        begin: direct_set_reset

            if (reset)
              y = 1'b0; //asynchronous reset
             else if (set)
              y = 1'b1; //asynchronous set
        end
```

```
        always @ (gate or reset)  //for set: (gate or set)
          if (reset)    //for set: if (set)
            t=1'b0; //forset:t=1'b1
          else if (gate)
            t = d;

endmodule
```

Example 14-7. Asynchronous set/reset on a design.

14.1.5.2 synchronous_set_reset – The synchornous_set_reset check causes Synthesizer to check specified objects for synchronous set or reset of a flip-flop. This directive takes one argument of a double-quoted list of single-bit signals separated by commas.

The syntax of synchronous_set_reset is

```
// Synthesis synchronous_set_reset "object_name,..."

module sync_set_reset(clk, reset, set, dl, d2, y, t);
        input clk, reset, set, dl, d2 ;
        output y, t;
        // Synthesis sync_set_reset "reset, set"
        reg y, t;

        always @ (posedge clk)
        begin: synchronous_reset
          if (reset)
            y = 1'b0; //synchronous reset
          else
            y = d1;
        end

        always @ (posedge clk)
        begin: synchronous_set
          if (set)
            t = 1'b1; //synchronous set
          else
            t = d2;
        end
endmodule
```

Example 14-8. Synchronous set/reset on a design.

14.1.5.3 One_hot – To prevent D flip-flops with asynchronous set and reset signals from being implemented with priority-encoded logic that would prioritize the set over the reset or vice versa, use the indications of one_hot and one_cold are used. Not using these may cause priority-encoded implementations to occur because the if...else construct in the HDL description specifies prioritization. The one_hot and

one_cold directives tell Synthesizer that only one of the objects in the list is active at one time. To define active high signals, use one_hot. To define active low, use one_cold. Each directive has two objects specified.

The one_hot dindicator takes one argument of a double-quoted list of objects separated by commas. This directive indicates that the group of signals (set and reset) are one_hot, i.e., no more than one signal is active high (has a Logic 1 value) at any one time. Users are responsible to make sure that the group of signals is really one_hot. Synthesizer does not produce any logic to check this assertion. The syntax of one_hot is:

```
// Synthesis one_hot "object_name,..."
```

This directive is only used for set and reset signals on sequential devices. For a general group of signals, do not use this directive for specifying that only one signal is hot.

```
module one_hot_example (reset, set, , out, data);
        input reset, set;
        output out;

        // Synthesis one_hot "reset, set"
        reg y , t ;

        always @ (reset or set)
                begin: direct_set_reset
                        if (reset)
                           y = 1'b0; //asynchronous reset by "reset"
                          else if (set)
                           y = 1'bl; //asynchronous set by "set"
                end
                // code for normal clocking cases here
                ..........................
endmodule
```

Example 14-9. Using one_hot for set and reset.

14.1.5.4 one_cold – As with the one_hot, one_cold indicator takes one argument of a double-quoted list of objects separated by commas. This directive indicates that the group of signals (set and reset) are one_cold, that is, no more than one signal is active low (has a Logic 0 value) at any one time. Users are responsible to make sure that the group of signals is really one_cold. Synthesizer does not produce any logic to check this assertion. The syntax of one_hot is:

```
// Synthesis one_hot "object_name,..."
```

This directive is only used for set and reset signals on sequential devices. For a general group of signals, do not use this directive for specifying that only one signal is hot.

```
module one_cold(reset, set, out; in)
            input reset, set;
            output out;
            input in;

           // Synthesis one_cold "reset, set"
            reg out;
            always @ (reset or set)
            begin: direct_set_reset
               if (~reset)
                y = 1'b0; // asynchronous reset by "~reset"
               else if (~set)
                y = 1'bl; // asynchronous set by "~set"
            end
    endmodule
```

Example 14-10. Using one_coldfor set and reset.

14.1.6 Bus Latch

The following example creates a bus latch.

```
module bus_latch(reset, set, control, out, in);
        input reset, set, control;
        input [0:1] in;
        output [0:1] out

        always @ (reset or set or control or in)
        begin:
                if (reset)
                        y = 0;
                else if (set)
                        y = 2;
                else if (control)
                        out = in;
        end
endmodule
```

Example 14-11. Creation of a bus-latch.

14.2 Multiplexers

Multiplexers or MUXes are widely used by hardware designers to implement conditional assignments to signals. The Verilog model of an 8-to-1 multiplexer is given below. This will be synthesized as a multiplexer (MUX) as the directive:

//Synthesis infer_mux "mux_blk"

```
module mux8tol(DIN,SEL,DOUT);
          input  [7:0]  DIN;
          input  [2:0]  SEL;
          output DOUT;
          reg DOUT;
          //Synthesis infer_mux  "mux_blk"

          always  @ (SEL or DIN)
          begin: mux_blk
                    case (SEL)
                              3'b000 : DOUT<=DIN[0];
                              3'b001 : DOUT<=DIN[l];
                              3'b010 : DOUT<=DIN[2];
                              3'b011 :DOUT<=DIN[3];
                              3'b100 : DOUT <=DIN[4];
                              3'bl01 : DOUT<=DIN[5];
                              3'b110 : DOUT<=DIN[6];
                              3'b111 : DOUT<=DIN[7];
                    endcase
          end
endmodule
```

Example 14-12. ***An 8-1 multiplexer modeled behaviorally and synthesized to a mux.***

14.3 Three-State Inference

Synthesizer can infer three-state gates from the value (high impedance) in the Verilog language which maps to the target technology library. When a variable is assigned the value z, the output of the three-state gate is disabled.

Example 14-13 shows the HDL for a three-state gate.

14.3.1 Modeling a Tri-State Gate for Synthesis

```
module simple_threestate (enable,in,out);
input in,enable;
          output out;
          reg   out;

          always @ (enable or in)
          begin
```

```
        if (enable)
          out = in;
        else
          out=1'bz; //assigns high-impedance
      end
endmodule
```

Example 14-13. A 3-state gate created behaviorally.

One Three-state Gate

```
always @ (sela or selb or a or b) begin
        t=1'bz;
    if(sela)
          t = a;
       if (selb)
          t = b;
end
```

Example 14-14. A 3-state gate with 2 drivers.

The value z can also appear in function calls, return statements, and aggregates. Although it is valid to use z in an expression such as if (value == 1'bz) Expressions that compare a value to z are always evaluated as false during synthesis. This evaluation might cause a difference between pre-synthesis and post-synthesis simulations. The code in Example 14-15 below infers two three-state gates.

14.3.2 Modeling Two Three-State Gates

```
always @ (select1 or in1)
  if (select)
    out = in1;
  else out = 1'bz;

always @(select2 or in2)
  if (select2)
    out = in2;
  else out = 1'bz;
```

Example 14-15. Creation of two 3-state gates with independent controls.

The Verilog conditional statement may also be used to infer three states.

14.3.3 Registered or Latched Three-State

When a variable is registered (or latched) in the same always block in which it is three-stated, the enable pin of the three-state is also registered (or latched). Example 14-16 shows an example of this code.

```
module enable_ff (clock, condition, enable, in, out);
        input in, enable, condition, clock;
```

```
    output out;
    reg    out;

    always @ (posedge clock)
   begin
     if (enable)
        out = ( ~condition) ? in : out;
   else
        out= 1'bz;
    end
endmodule
```

Example 14-16. Three-state with registered enable.

14.3.4 Three-State with Registered Enable

In Example 14-16, the three-state gate has a register on its enable. To remove the register from the enable, use two always blocks to separate the register inference from the three-state gate inference, and add a register temp, as shown in Example 14-17.

```
module threestate_noreg_ff (clock, condition, enable, in, out);
        input in, enable, condition, clock;
        output out;
        reg out;
        reg temp;

        always @ (posedge clock)
         begin //flip-flop on input
            if (condition)
                temp = in;
         end

        always @ (enable or temp)
         begin
                 if (enable)          //three-state
                  out = temp;
                 else
                  out= 1'bz;
        end
endmodule
```

Example 14-17. Three-state without registered enable.

Synthesizer can infer three-state gates from the value (high impedance) in the Verilog language which maps to the target technology library. When a variable is assigned the value z, the output of the three-state gate is disabled.

14.4 Designs via Resource Sharing

14.4.1 Introduction

Resource sharing is a common term for synthesis techniques that applies to the assignment of same operation (for example, +) from different statements to a common library cell. Library cells are the resources—they are equivalent to built hardware units. Resource sharing or sharing of these units of hardware amongst different operations or statements in Verilog is an excellent optimization method. Without resource sharing, each Verilog operation is built with separate circuitry. For example, every + with non-computable operands causes a new adder to be built. This repetition of hardware increases the area of a design. In contrast, with resource sharing, several Verilog + operations can be implemented with a single adder, which reduces the amount of hardware needed. Also, different operations such as + and - can be assigned to a single adder/subtracter to further reduce area or number of gates in an implementation of a Verilog model.

The three sharing methods of resource sharing while using a synthesizer are:

- Automatic or Synthesizer driven,
- Manual or designer driven, and
- Automatic with designer's knowledge fed back into the system.

In automatic sharing, Synthesizer completely handles the job of resource-sharing without any knowledge on part of the user. In designer-driven sharing, directives are given to the synthesizer to assign operations to resources, and the operations shared are explicitly assigned to the same resource. In automatic sharing with designer's feedback, , you insert manual control statements in your Verilog source that assign operations to resources, and the undeclared resource-mappings are handled by the synthesizer. In general, manual controls modify the sharing configuration produced through automatic sharing to solve a specific problem, such as a violated timing constraint. When interacting with the synthesizer, several reports are available—like any software tool Use these to optimize and generate designs that meet all your criteria and needs.

14.4.2 Sharable Resources

Not all operations in your design can be shared. This section describes how to tell whether operations are candidates for sharing. Typically the datapath or the Arithmetic Logic Units can be shared. Control is inherently a non-sharable logic unit.

The following operators in Verilog can be shared:

```
*
+ -
> > = < < =
```

The shared resources above are those with same operation repeated elsewhere or similar operations like comparsions of various kinds and between add and subtract.

Operations can be shared only if they lie in the same always block. When units are in different blocks, their control is independent and thus, all the environment (or peripheral circuits) is not reproducible and thus is not a sharable resource. Shared resources must be exact matches in terms of not only their functionality but all their connections.

Example 14-18 shows several possible sharings.

```
always @(xl or yl or zl or w1 or conditiona)
begin
   case(conditiona)
     'b0:al=xl+yl;
     'bl : al =zl+wl;
end

always @(x2 or y2 or z2 or w2 or conditionb)
begin
   case (conditionb)
     'b0: z2 = a2 + b2;
     'bl :z2 = c2 + d2;
end
```

Example 14-18. Sharing of adder resources.

The table in Figure 14-1 summarizes the possible sharings in Example 14-18. A no indicates that sharing is not allowed because the operations lie in different always blocks. A yes means sharing is allowed.

	x1 + y1	z1 + w1	x2 + y2	z2 + w2
x1 + y11	yes	yes	no	no
z1 + w1	yes	yes	no	no
x2 + y2	no	no	yes	yes
z2 + w2	no	no	yes	yes

Figure 14-1. Sharings Adders Amongst Different Operations in Example 14-1

We can see that A1+B1 and C1+D1 occur in the same always block and can potentially share a resource.

In addition, their control path is mutually exclusive that is either one or the other may happen at any given time but not both.

14.5 Control Flow and Data Flow with Sharing

Two operations may be shared only if no execution path exists from the start of the block to the end of the block that reaches both operations. For example, if two operations lie in separate branches of an if or case statement, they are not on the same path (and can be shared). In Example 14-18, we shared resources amongst add operations on separate branches. In Example 14-19, the add of (a+b) and the adds for (c+d) lie on the same path and cannot share a resource. Since the sharing is done either in time or in space, i.e., in multiplexed connections to a resource, only one set of inputs will be active at a given time.

```
always
begin
    zl = a + b;
    if(cond_l)
      z2 = c + d;
    else
    begin
      z2 = e + f;
end
```

Example 14-19. Shared resources have independent paths.

```
always
begin
   x = a + b;
   case(op)
           2'h0:z2 = c+d;
           2'hl:z2 = e+f;
           2'h2: z2 = g+h;
           2'h3: z2 = i+j;
   endcase
end
```

Example 14-20. Case statement sharing.

In the Example 14-20, the four branches of case statements can share the adder resource. However, the first adder use before the case statement for assignment to x, needs a separate adder. Thus, total number of resources in this example is 2.

The '?' operator is an exact equivalent of an if statement and the resource-sharing rules that apply to if statement also apply to the assignments using this. However, some implementations (like Synopsys) do not treat these the same and consequently, resources are not shared for this operator.

14.5.1 Data Flow Conflicts

Operations may not be shared if doing so causes a combinational feedback loop. To understand how sharing can cause a feedback loop, consider Example 14-18 below:

```
always @(a or b or c or d or e or f or z or add_b)
begin
        if(add_b) begin
                temp_1 = a + b;
                z = temp_1 + c;
        end
        else begin
                temp_2 = d + e;
                z = temp_2 + f;
        end
end
```

Example 14-21. Sharing may result in feedback loop.

When the a+b addition is shared with the temp_2+f addition on an adder called r1, and the d+e addition is shared with the temp_1+c addition on an adder called r2, a feedback loop results. The variable temp_1 connects the output of r1 to the input of r2. the variable temp_2 connects the output of r2 to the input of r1, and a feedback loop is created.

The circuit generated has the multiplexing conditions never allow the entire path to be activated simultaneously. Still, Synthesizer's resource sharing mechanism does not allow combinational feedback paths to be created because most timing verifiers cannot handle them properly.

14.5.2 Resource Area

Resource sharing reduces the number of resources in your design, which reduces resource area. The area of a shared resource is a function of the types of operations that are shared on the resource, and their bit-widths. The shared resource is made large enough to handle the largest of the bit-widths and powerful enough to perform all the operations. Resource sharing usually adds multiplexers to a design to channel values from different sources into a common resource input. In some cases, resource sharing reduces the number of multiplexers in a design.

Multiplexer area is a function of both the number of multiplexed values.

14.5.3 Resource Sharing Methods

14.5.3.1 Synthesizer Driven (Automatic) Resource Sharing – Automatic resource sharing is the simplest way to share components and reduce design area. This method is ideal if you do not know how you want to map the operations in your design onto hardware resources. In automatic sharing, Synthesizer identifies the operations that can be shared.

14.5.3.2 Designer Driven and Mixed Designer and Synthesizer Driven – In automatic production of synthesized logic, the timing and area constraints can be met or not met. However, the real life designs involves tradeoffs between the two and

between other features (functionality) and design techniques. Thus, an interactive loop whereby one can automate tasks that one is satisfied with and manage the other parts in a iterative fashion to some extent is possible by interacting with the synthesizer. However, that involves intimate knowledge of workings of synthesis algorithms and techniques. Resource sharing is one such algorithm where this understanding is possible and interaction is designed into synthesizer tools. Thus, one can specify different operations mapping into resources and then specify which resources are shared between which units—in effect directing the design's creation of units like multiplexers [shared resources must be multiplexed], and the datapath elements. Mapping of the resources into the hardware library units is also done in this process. Several interaction commands involve setting of global parameters and then making exceptions to these at the local level.

15 SPECIFY BLOCKS — TIMING DESCRIPTIONS

15.1 Overview

The specify blocks describe module timing checks and pin to pin timing in Verilog. The timing checks include predefined systems tasks like $setup that support programmable checks on times of value changes module pins.

Two types of HDL constructs describe delays:

1. Distributed delays – These model the time it takes for events to propagate through gates, nets, udps, rtl and behavioral descriptions inside the module.
2. Module path delays – These describe the time it takes an event at a source [input port or inout port] to propagate to a destination [output port or inout port].

The first types of delays above are explained earlier in the relevant sections. This section describes how paths are specified in a module, how conditional paths are created in a model, how delays are assigned to these paths.

15.2 Example

```
module dff(q, clk, d);
        input clk, d;
        output q;

        specify
                specparam tRise_clk_q = 150, tFall_clk_q = 200;
                specparam tSetup = 70;

               (clk => q) = (tRise_clk_q, tFall_clk q);

                $setup (d, posedge clk, tSetup);
        endspecify
```

```
        dff_logic   dffi (q, d, clk);
endmodule

module dff_logic(q, data, clock);
    input clock, data;
    output q;

        always  @posedge clock
                q = data;..................
endmodule
```

Example 15-1. Timing checks and path delay specifications in specify blocks.

15.3 Specify Blocks – Syntax

specify_block
::= **specify** { specify_item} **endspecify**

specify_item
::= specparam_declaration
| path_declaration
| system_timing_check

path_declaration
::= simple_path_declaration
| edge_sensitive_path_declaration
| state_dependent_path_declaration<specify_block>

15.4 Timing Checks in Specify Blocks

Timing checks are done via pre-defined systems tasks as follows:

$setup, $hold, $width, $period, $skew, $recovery, $setuphold

The timing checks are done on the events on nets and regs with the goal of checking the time between two events on two signals or between two events on the same signal. More often than not, these checks are done on the inputs to flip-flops although these can be done on any ASIC cell. In the terminology used to describe these the following describe the arguments to the tasks above:

data_event - describes the net or the reg or an event on these that typically is indicative of a data
reference_event is the event with whose time the difference is obtained
limit is the positive constant expression or specparam
notifier - register that changes value on the timing violation

15.4.1 $Setup Timing Check

The setup timing check is typically performed between the data and the clock signal for a flip-flop and checks the time needed foe setting up the data input before a clocking edge arrives.

The arguments are:

```
$setup(data_event, reference_event, limit, notifier);
```

where

data_event is lower bound event
reference_event is upper bound event
limit is the positive constant expression or specparam
notifier - register that changes value on the timing violation

The timing check consists of:
(time of reference_event - time of data_event) < limit

EXAMPLE:

```
module m(. . . .);
    reg notif_reg;
    wire data, clock;

            specify
        $setup(posedge data, posedge clock, 10, notif_reg);
          endspecify

            always @ notif_reg
        $display("Setup violation in %m at time %t", $time);
endmodule
```

15.4.2 $Hold

This check typically checks for the duration for which data signal must hold its value to transfer the input value to the output of a flip-flop on the active clock edge. The arguments to $hold are described in the following line:

```
$hold(reference_event, data_event, limit, notifier);
```

The timing check in $hold is done as follows:

time of data_event - time of reference_event < limit

15.4.3 $Width

This check typically checks the pulse width of a signal to be longer than the given limit. $width has arguments as specified in the following line.

$width(reference_event, limit, threshold, notifier);

Here check is performed between two edges of same signal.

The reference event and its opposite event are checked for limit in $width timing check.

15.4.4 $Period

This typically checks for the time-period of a clock input to be as specified. The arguments to $period are as follows:

$period(reference_event, limit, notifier);

The check is performed as follows: data_event is internally generated from reference_event by using the next event same as reference_event.

Thus, period is checked for limit between two successive reference_events. Violation is reported if (time of data_event - time of reference_event < limit)

15.4.5 $Skew

The skew check is typically performed to see if the clock signal has shifted from its original timing due to the gating of other logic.

The arguments to $skew are as follows:

$skew(reference_event, data_event, limit, notifier);

The check is performed as follows:

(time of data_event - time of reference_event > limit)

If this is true, violation is reported.

15.4.6 $Recovery

The check for recovery time involves checking for a change in value to recover from a changed value.

The arguments to $recovery are as follows:

$recovery (reference_event, data_event, limit, notifier);

Violation is reported if (time of data_event - time of reference_event < limit).

15.4.7 $Setuphold

This check combines the setup and the hold checks as typically these two are done together.

This has the following format:

```
$setuphold(reference_event, data_event, setup_limit,
        hold_limit, notifier);
```

This system task is equivalent to two checks as follows:

```
$setup(reference_event, data_event, setup_limit, notifier);

$hold(reference_event, data_event, hold_limit, notifier);
```

15.4.8 Example of Timing Checks

```
primitive negdff(q, clock, data, preset, clear, notifier);
    output q; reg q;
    input clock, data, preset, clear, notifier;

table
    //clock data p c notifier state q
      f  0   1 1  ?: ? : 0;
              f   1    11?       :?:1;

              n   1    1?  1      :1:1;
              n  0    1 ?  ?     : 0 : 0;

              p  ?    ???:?:-;
              ?  ?   01  ?:?:-;

              ?  ?  * 1  ? : 1 : 1;
              ?   ?   10 ?      : ?: 0;

              ?  ?   10 ?   : ? : 0;
              ?   ?   ? ?  *; ? ; x;
    endtable
endprimitive

module dff(q, qBar, clk, d, p, c);
    output q, qBar;
    input clk, d, p, c;

    reg notifier;

    and (e, p, c);  //generate enable signal for dff
    not (qBar, udp_out);
    buf (q, udp_out);

    negdff n(udp_out, clk, d, p, c, notifier);

    specify
        // Define timing check specparam values
        specparam tsetup =10, thold =
            2, tclkwidth = 20, trecover = 5, t_pc_width;
```

```
        specparam tPLHc = 4:6:9, tPHLc = 5:8:11;
        specparam tPLHpc = 3:5:6, tPHLpc = 4:7:9;

        //Model module path delays
        (clk *> q, qBar) = (tPLHc, tPHLc);
        (p, c *> q, qBar) = (tPLHpc, tPHLpc);

        // Model the timing checks
        $setup(d, posedge clk && e, tsetup, notifier);
        $hold(d, negedge clk && e, thold, notifier);
        $period(negedge clk, twidth, notifier);

        $width(negedge c, t_pc_width, notifier);
        $width(negedge p, t_pc_width, notifier);

        $recovery(posedge c, posedge clk, (recover, notifier);
    endspecify
endmodule
```

Example 15-2. Specify block example – timing checks on module pins.

15.5 Module Path (Delay) Declarations

15.5.1 Introduction

In specify blocks, path, and delay declarations are made together in the same statement. These declarations fall into three categories for paths: simple, edge-sensitive, state-dependent. These are explained below.

15.5.2 Simple Paths

15.5.2.1 Examples

```
(A=>Q) = 10;
(B=>Q) = (12);
(C,D*> Q, QBAR) = 18;
```

Example 15-3. Specify blocks – simple paths from input to output.

This example contains three statements describing paths from inputs to outputs of a flip-flop. The last example implies all possible paths from inputs to outputs

15.5.2.2 Syntax

```
simple_path_declaration
    ::= parallel_path_description = path_delay_value;

parallel_path_description
```

::= (specify_input_terminal_descriptor[polarity_operator] **=>**
specify_output_terminal_descriptor)

specify_output_terminal_descriptor::=
output_identifier
| output_identifier[constant_expression]
| output_identifier[msb_constant_expression : lsb_constant_expression]
specify_input_terminal_descriptor
::=input_identifier
| input_identifier [constant_expression]
| input_identifier [constant_expression: constant_expression]

15.5.3 Edge-Sensitive Paths

```
(posedge clock => (out: in)) = (10, 8);
    (clock => (out: in)) = (10,8);
```

Example 15-4. Edge-sensitive paths with sensitivity to positive clock edges.

The edge sensitive path specifications are stated before the simple paths as for the clock edges in the above two examples.

15.5.3.1 Syntax

edge_sensitive_path_declaration ::=
parallel_edge_sensitive_path_declaration = 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_expresssion

full_edge_sensitive_path_description ::=
([edge_identifier]list_of_path_inputs*>
list_of_path_outputs [polarity_opertor]: data_source_expression

15.5.4 State-Dependent Paths

15.5.4.1 Examples – The following example uses state-dependent paths to describe the timing of an XOR gate.

```
module XORgate (a, b, out);

        input a, b:
        output out;
        xor xl (out, a, b);

        specify
```

```
                specparam noninvrise = 1, noninvfall = 2;
                specparam invertrise = 3, invertfall= 4;
                if (a) (b => out) = (invertrise, invertfall);
                if (b) (a => out) = (invertrise, invertfall);
                if (~a) (b => out) = (noninvrise, noninvfall);
                if ( ~ b ) (a => out) = (noninvrise, noninvfal 1 );
        endspecify
endmodule
```

Example 15-5. State dependent path delay specifications: example statements.

In this example, first two state-dependent paths describe a pair of output rise and fall delay times when the XOR gate (xl) inverts a changing input. The last two state-dependent paths describe another pair of output rise and fall delay times when the XOR gate buffers a changing input.

Another Example

```
module ALU (ol, il, i2, opcode);
        input [7:0] i1,i2;
        input [2:1] opcode;
        output [7 :0]o1;

        //functional description omitted
        specify
         // add operation
                if (opcode == 2'b00) (i1,i2 *> ol) = (25.0,25.0);
                //pass-through il operation
                if (opCOde == 2'b01) (i1 => ol) = (5.6,8.0);
                // pass-through i2 operation
                if (opcode == 2'b10) (i2 => ol) -=(5.6, 8.0);
                // delays on opcode changes
                        (opcode => ol) = (6.1,6.5);
        endspecify
endmodule
```

Example 15-6. State dependent path delay specifications – a full module.

15.5.4.2 Syntax

data_source_expression ::= expression
edge_identifier ::= **posedge** | **negedge**
state_dependent_path_declaration ::=
    if (conditional_expression) simple_path_declaration
    if (conditional_expression) edge_sensitive _path_declaration
    ifnone simple_path_declaration

15.5.5 Edge-Sensitive State-Dependent Paths

```
if (!reset && !clear)
(posedge clock => (out+ : in)) = (10, 8 );
```

Example 15-7. ***Edge sensitive state dependent path delays.***

In the Example 15-6, if the positive edge of clock occurs when reset and clear are low, a module path extends from clock to out using a rise delay of 10 and a fall delay of 8.

The following example shows three edge-sensitive path declarations. Note that each path has a unique edge or condition.

```
specify
    ( posedge clk => (q [ 0 ]: data)) = (10, 5 );
    ( negedge clk=>(q[0]:data))=(20,12);
    if (reset)
            ( posedge clk=>(q[0]:data))=(15,8);
endspecify
```

Example 15-8. ***Multiple edge and state delay specifications for the same simple path.***

The two state-dependent path declarations shown below are not legal because even though they have different conditions, the destinations are not specified in the same way—the first destination is a part-select, the second is a bit-select.

```
specify
    if (reset)
        (posedge clk => (q [3:0]) :data)) = (10,5);
    if (!reset)
        (posedge clk => (q[0] :data)) = (15, 8);
endspecify
```

Example 15-9. ***Illegal state dependent delay specification.***

15.5.6 State-Dependent Paths – ifnone Condition

State dependent paths enumerate conditions on a path and provide delay specifications for that path. For that path, 'ifnone' (default case) statement provides delays when none of the states explicitly specified are attained.

```
if(C1)(IN=>OUT)=(1,1);
ifnone ( IN=>OUT ) = ( 2,2 );
```

Example 15-10. ***ifnone statements in a state dependent specification.***

In the above example, the if statement is complemented by the following ifnone statement which indicates that when Cl is not 1 the second delay specification will be used.

```
// add operation
if (opcode ==2' b00) (il, i2 *> ol) = (25. 0, 25. 0);
// pass-through il operation
if (opcode == 2'b0l) (il => ol) = (5.6, 8.0);
// pass-through i2 operation
if (opcode == 2'bl0) (i2 => ol) = (5.6, 8.0);
// all other operations
ifnone (i2 => ol) = (15. 0,15. 0);
```

Example 15-11. ifnone statement – opcode dependent delays for execution unit.

15.6 Delay Specifications

The specify block enables specification of 1- to 12-delay elements in the delay value. Each of the delay value could be a triplet—(minimum, typical and maximum). In the following pages, we explain the delay specifications via different examples. The comments preceding the delay specifications explain the different kinds of delay specifications.

15.6.1.1 Examples of Delay Specification in Specify Blocks

```
// one expression specifies all transitions
(C=>Q) = 20;
(C=>Q)= 10:14:20;
```

Example 15-12. Delay specifications – one delay for all cases and a min-typ-max specification.

In the above example, delays are specified for the same path in two different ways. The example below contains specifications for a path using different levels of detail.

```
// two expressions specify rise and fall delays
specparam tPLHl = 12, tPHLl = 25;
specparam tPLH2 = 12:16:22, tPHL2 = 16:22:25;
(C => Q) = (tPLHl, tPHLI);

(C=>Q) = (tPLH2,tPHL2);

// three expressions specify rise, fall, and z transition delays
specparam tPLHl = 12, tPHLl = 22, tPzl = 34;
specparam tPLH2 = 12:14:30, tPHL2 = 16:22:40, tPz2 = 22:30:34;
(C => Q) = (tPLHl, tPHLl, tPzl);
(C => Q) = (tPLH2, tPHL2, tPz2);
```

```
// six expressions specify transitions to/from 0,1, and z
specparam t0l = 12, tl0 = 16, t0z = 13,
        tzl=10,tlz=14,tz0 = 34;
(C => Q) = (t01, tl0, tOz, tzl, tlz, tz0);
specparam TOl = 12:14:24, T10 = 16:18:20, TOz = 13:16:30;
specparam Tzl = 10:12:16, Tlz = 14:23:36, Tz0 = 15:19:34 ;
(C => Q) = (TOl, Tl0, TOZ, Tzl, Tlz, TZO);

// Twelve expressions specify all transition delays explicitly
specparam t01=10, tl0=12, tOz=14, tzl=15, tlz=29, tz0=36,
  tOx=14, txl=15, tlx=15, tx0=14, txz=20, tzx=30 ;
(c => Q) = (t01, tl0, tOz, tzl, tlz, tz0,
  tOx, txl, tlx, tx0, txz, tzx);
```

Example 15-13. ***Two, three, six, and twelve different delays for the same path.***

15.7 Mixing Distributed and Specified Delays

Verilog allows double specification of delays in a module. This can occur if the specify blocks and the functional blocks both contain delay specifications for the same paths. In other words, both path delays as well as distributed delays can be present on the same part of the design at the same time. In such case, distributed delays take effect as the events on the nets or regs take place -and the path delays are induced into the design when the output end of the path changes. When mixed, slower values take precedence -that means path delays will be effective if those values are larger than the delays inside the module.

15.8 Multi-Driver Nets

Declaring paths on these is not legal.

However, drivers going out from a module are allowed.

15.9 Pulse Specification

PATHPULSE$ construct

Example: PATHPULSEclkq = (2,9); // (reject-limit, error-limit)

15.10 Exercises

1. Provide the output from the following model that uses specify block.

```
module dff(q, d, clk, reset);
input clk, d, reset;
output q;

specify
    specparam tRise_clk_q = 150, tFall_clk_q = 200;
    specparam tSetup = 70;
```

```
        (clk => q) = (tRise_clk_q, tFall_clk_q);
        (reset =>q) = (20,10);

endspecify
dff_logic dffi (q, d, clk, reset);
endmodule

module dff_logic(q, data, clk, reset);
regq;
output q;
wire clk, reset, data;

always @posedge clk
        q = data;

always @reset
        if (reset ==1)
                deassign q;
        else
                assign q = 0;
endmodule
```

16 PROGRAMMING LANGUAGE INTERFACE

16.1 Overview and Examples

The Programming Language Interface (PLI) provides a mechanism to link non-Verilog code into Verilog simulation system. The non-Verilog code may be a model in another language like "C" or VHDL, a program to enhance the simulation system's capabilities in various ways or another simulator communicating with the Verilog simulation system. This aspect of Verilog HDL is explained in the sections below. In essence, the PLI provides ability to write programming language (e.g., "C") tasks that interact with a VERILOG model, and link it tightly. The commonly used versions of PLI are OVI 1.0, 2.0 and IEEE 1364. All the three versions have the same set of core utility and access routines. The VPI interface is added in 2.0 and 1364 recently. A program written in "C" must use the interface routines from PLI to interact with the Verilog engine.

This ability to extend the language by way of adding keywords is achieved through user-defined tasks and functions. For example, one can take his or her favorite waveform-processing package, and link it in place of $gr_waves. This can then be named $gr_waves_new() and invoked either in interactive debugging session or in simulation at a certain point in test-bench part of the Verilog system being built.

You may be currently using a system that displays output in certain format and you are switching to Verilog. Then, a C task that imitates current output will enable smooth transition. Whenever a value changes, this will print out the VERILOG values in a format similar to existing format. This way all diagnostics that happens post-simulation can be preserved mostly as is.

The three sets of call-backs in the PLI are:

1. Utility routines dealing with interaction to Verilog during simulation.

2. Access routines dealing with circuit and connectivity information about Verilog model

3. The Virtual Procedural Interface (VPI) Interface routines that provide simulation time as well access routines.

The common applications of the Programming Language Interface are:

- User interface independent of the simulator
- Linking in modules outside of Verilog modules (e.g., VHDL model of an fpga)
- Adding capabilities like delay calculators
- Adding simulation control capabilities like distributed simulation
- Interfacing to other tools
- Your creativity and imagination is the limit.

16.2 PLI Origin and Use

Programming Language Interface was developed during the development of the Verilog simulator to facilitate the development of system tasks and functions which are part of the Verilog HDL. This was a good software practice to develop different parts of the simulator itself. For example, the tasks of $display, $monitor, or $gr_waves are written and linked to the Verilog Simulation Engine using the same PLI mechanism as explained in this chapter.

16.3 PLI Function Types

A C function invoked from within Verilog Model could be invoked in several ways and is classified into such types. The main types are:

1. **calltf functions –** These are called during simulation when a task associated with the user function is called. An example of this is $display system task which displays the values as this task is called. A different type of invocation takes place for $monitor which is explained in the misctf type below.

2. **misctf functions –** These are called during simulation but are not called directly as a result of a call from the model. These are called due to value change on the inputs. A familiar application of this type is Smonitor system task which is called whenever one of its inputs changes values.

3. **checktf functions –** These are called at compilation time for the Verilog model and typically perform tasks of checking for legal parameters to the task.

4. **sizetf functions –** This is used to obtain number of bits in a return value of a system function but is not used for user-defined PLI functions.

5. **VCL or consumer functions –** The calltf functions can be modified to act like misctf functions dynamically. VCL stands for Value Change Link and is set in the calltf applications.

16.4 PLI Interface Classes

PLI Routines are classified into three classes:

1. **Simulation Time Routines with tf_ prefix –** These are used for interacting with the model at run-time. This is used for linking in external models and non-Verilog simulators into the Verilog simulation system amongst other applications. Most of the system tasks and functions are written using these conventions. Examples of these are $display and $monitor which clearly access the values of Verilog Model variables. Some examples of the routines here are tf_getp() and tf_putp() which get a value and write a value.

2. **Access Routines with acc_ prefix –** These are used for accessing the Verilog model data-structure and are typically used for applications such as delay calculators. These are typically called before simulation starts and after compilation is complete.

3. **New Procedural Interface routines with vpi_ prefix –** These routines provide the ability to access the data-structure as well as perform the simulation time functions through a new mechanism. In this method, a handle is obtained on an object like in access methods for the access part and a traversal mechanism to the data structure is provided using the iterate and vpi_get_next routines. This also provides mechanism to interact with different simulation and compilation phases.

16.5 Interface Definitions

Three header files are provided with your Verilog system:

veriuser.h, accuser.h and vpi_user.h.

Include in your "C" application.

The contents of these files are explained below. These provide the necessary "C" data-structure definitions and the interface routines that enable a user application to become part of the Verilog model.

16.5.1 veriuser.h

This is a header useful for interacting with Verilog via utility routines—the basic run-time interaction mechanism. This header file:

- Defines parameter passing conventions for integer, node, expression and string values.
- Defines types for node and expression data types as well as vector and strength value structs defines task to user routine mapping and synchronization convention.

The key to linking user-defined "C" routines to the Verilog Simulator lies in the following definitions:

```
typedef struct veriusertfs
      {     int type;
            int data;
            void (*calltf)();
            void (*checktf)();
            int (* sizetf)();
            void (*misctf)();
   } t_veriusertfs;
  _tfcell type - define 4 routines check, size, call, misc -
```

These coordinate "C" and Verilog activities. The callback reasons are returned in tf_reason call. The callbacks allows you to call your routine on conditions like value change or at the end of simulation. These also define callbacks supported for the IO between Verilog and your application.

16.5.2 acc_user.h

This is a header useful for interacting with Verilog via access routines—the basic compile-time interaction mechanism for accessing network data structure.

This header file defines:

- Type and configuration constant definitions

  e.g., accTri
- acc_configure() parameters

  e.g., accPathDelayCount
- Edge information used by acc_handle_tchk etc

  e.g., accEdge10
- Version defines

  e.g., accVersion16a4
- Delay modes

  e.g., accDelayModePath
- typedefs for time, delays, values(vector, strength and general), record,location
- Flags for VCL interface

  e.g., vcl_verilog_logi
- CallBack Routines interface
- Handle routines

  e.g., acc_handle_port()
- Next routines

  e.g., acc_next_port()
- Fetch routines

  e.g., acc_fetch_value()

- Modify routines
  e.g., acc_replace_delays()
- Pulse Err routines
  eg., acc_fetch_pulser()
- Utility routines
  e.g., acc_initialize()
- Value Change Link routines
  e.g., - acc_vcl_add()
- Line Callback Routines
  e.g., acc_mod_lcb_add()

16.5.3 vpi_user.h

This defines constants, data structure definitions, and routine interface definitions used in the VPI interface.

Examples of constant definitions are:

```
#define vpiEdge        36 /* edge type of module path: */
#define vpiNoEdge       0x00000000 /* no edge */
#define vpiEdge0l      0x00000001 /* 0-> 1*/
#define vpiEdgelO      0x00000002 /* 1-> 0*/
```

Some of the data structure definitions are as below :

```
/**************************************************************************/

typedef struct t_vpi_time   {
   int type;              /*  [vpiScaledRealTime,vpiSimTime,vpiSuppressTime]*/
   unsigned int high,low;     /* for vpiSimTime */
   double real;         /* for vpiScaledRealTime */
} s_vpi_time,*p_vpi_time;

/************************valuestructures ***************************/
/* vector value */

typedef struct t_vpi_vecval
{
 /* following fields are repeated enough times to contain vector */
 int aval, bval; /* bit encoding: ab: 00=0,10=1,11=X, O1=Z */
) s_vpi_vecval, *p_vpi_vecval;

extern vpi_handle( vpiObjectType otype, vpiHandle object)

extern vpi_handle_multi();

/*** register a system task/function ***/
```

```
extern vpi_register_systf();

/** Get information about a system task/function callback **/
extern vpi_get_systf_info();

/*** Obtain a handle by name ***/
extern vpi_handle_by_name(char *name, vpiHandle object);
***********************************************************************/
```

16.6 User Tasks and Functions

PLI allows one to define $xxx tasks and $yyy functions that associate with "C" functions written by you. These tasks can then be invoked from Verilog model just like system tasks and functions. In fact, most system functions and tasks in Verilog are written in "C" using the conventions of programming language interface.

```
Example:
%cat hello.c
s_tfcell veriusertfs[] =
{
      { usertask, 0,0,0, hello, 0, "$hello", 0},
      {0}
};

int helloQ
{
 io_printf("Hello World!\n");
}

%cc -c hello.c
% cc verilog.o hello.o -o myverilog
 %cat model.v
 module m;
initial
   $hello;
 endmodule
%myverilog model.v
Cl>$hello;
Hello World
C2>...
```

16.7 Steps Involved in Defining User Tasks/Functions

1. Define the task or function name and parameters.
2. Write a checking function for checking parameters.
3. Write a function to clean on simulation finishing.
4. If the task needs to be called asynchronously (e.g., on any value change in the simulator), define misctf parameters.

5. Define a struct of type t_tfcell for this set of functions.
6. Write the "C" routine, using the interface utility routines to obtain parameter values, and to interact with the Verilog model and its simulation profile, e.g. Use tf_getp() to get integer parameters in sequence.

16.8 C Interface Components

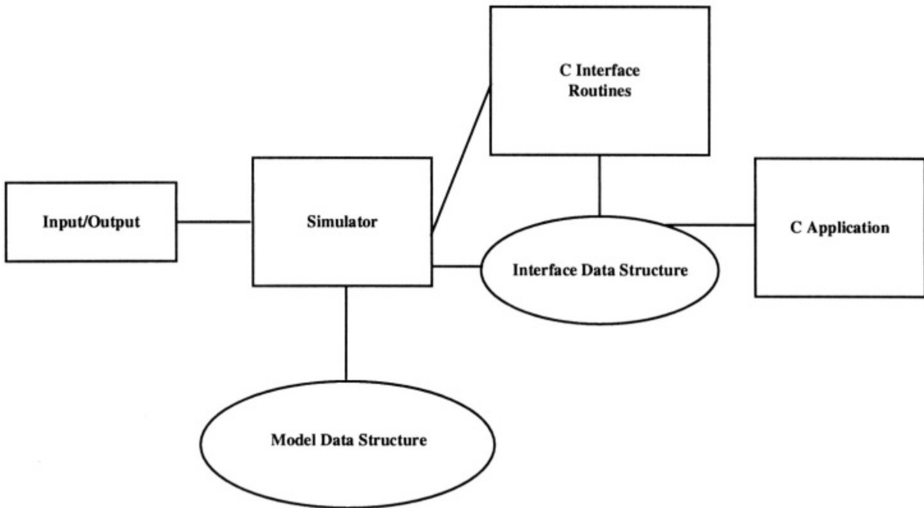


Figure 16-1. C Interface Components for Verilog HDL

16.9 Verilog Callbacks – Utility Routines

The Appendix C contains an annotated listing of the file veriuser.h that lists all the routines and their parameters.

16.10 Verilog Callbacks – Access Routines

The Appendix D contains an annotated listing of the file acc_user.h that lists all the routines and their parameters.

16.11 Verilog Callbacks – VPI Routines

The Appendix E contains an annotated listing of the file vpi_user.h that lists routines provided in this interface.

16.12 Exercises

1. Give the output of following two Verilog modules:

```
    module m1;
reg [7:0] in, out;
```

```
initial
      begin
            in = 8;
            $monitor($time,,in,, out);
            #100
            $log2(in, out);
      end
endmodule
```

Note: Use the $log2 task as defined using Programming Language Interface, in the book, for both the modules.

```
module m2;
reg [7:0] in, out;
initial
      begin
            in =16;
                  $monitor($time,,in,, out);
                  #100
            $log2(in, out, in);
      end
endmodule
```

2. Most of the Verilog system tasks and functions are written using PLI in simulators. $monitor is one such system task. Given the 'veriuser_tfs' type definition, and a set of "C" functions.

   tf_monitor_check() – Call this for compile time checks
   tf_monitor() – Call this on value change of one of its parameter

   Write the veriuser_tfs structure related to this task.

   Similarly do the same for $display, given:

   tf_display_check() – "C" function for compile time check
   tf_display() – Call this when $display task is called

17 STRENGTH MODELING WITH TRANSISTORS

17.1 Overview

This is the lowest level of modeling provided in Verilog. It also allows the greatest levels of details in terms of the circuit-implementation at the digital level. There are three main methods or levels in Verilog that describe transistors:

1. The first level includes unidirectional transistor models. At this level Verilog HDL provide nmos, pmos and cmos primitives. These have the switching behavior of the transistors including z values. The tables for these primitives have been discussed in Chapter 6.

2. At the next level, bidirectional transistors represent the transistors in real life as these do not have directionality. The primitives provided in this class of transistors includes:

   tran, tranif0, tranif1
   rtran, rtranif0, rtranif1.

   The terminals of a bidirectional transistor are of the inout type and the control terminal is of the input type. Steady state values are computed of the whole network, not of a single device when the circuit includes tran types. The primitive tran is always ON and conducts on either side. The primitive tranif0 is similar to tran but has ON/ OFF properties. It is on when the gate(or the control signal) is 0. The primitive tranif1 is ON when the gate is 1.

3. The next level is modeling with strength values. This allows removal of pessimism and allows transistor sizes and other electrical characteristics to be taken into account. This level defines an algebra of an extended value-set known as strength calculus. The value-set has now expanded from 0, 1, x, z to 128 different values. The extension occurs as each value has two components now—

the logic value (0,1,x,z) and the strength value. Strength values are like supply (the strongest), strong, weak, hiz (the weakest). Although eight levels of strengths and four logic values imply 256 combinations, half of these are not meaningful or non-distinct and this reduces the total number to 128 values. Examples of these are: with z as a logic value, no strengths are meaningful—implying that the 64 combinations are now reduced to 1. Similarly, for a logic value of 1, only combinations where 1 strength is higher than the 0 strength are meaningful. For logic value 1, the vice-versa is true.

17.2 Modeling with Unidirectional Switches – Example

```
//Dynamic MOS serial shift register circuit description
module shreg (out, in, phase1, phase2);
    /* IO port declarations, where 'out' is the inverse
    of 'in' controlled by the dual-phased clock */

        output      out;    //shift register output
        input       in,     //shift register input
        phase1,     //clocks
        phase2;

tri     wb1, wb2, out;                      //tri nets pulled up to VDD
pullup          //depletion mode pullup devices
        (wb1), (wb2), (out);

trireg      (medium) wa1, wa2, wa3      ;   //charge storage nodes

supply0         gnd;    //ground supply

    nmos #3  //pass devices and their interconnections
            a1(wa1,in,phase1), b1(wb1,gnd,wa1),
            a2(wa2,wb1 ,phase2), b2(wb2,gnd,wa2),
            a3(wa3,wb2,phase1),  gout(out,gnd,wa3);
endmodule

module testShReg;
wire    shiftout;  //net to receive circuit output value
reg     shiftin;        //register to drive value into circuit
reg     phase1,phase2;  //clock driving values

parameter           d = 100;            //define the waveform time step

    shreg       cct (shiftout, shiftin, phase1, phase2);

    initial
        begin :main
            shiftin = 0;                    //initialize waveform input stimulus
            phase1  = 0;
            phase2 = 0;
```

```
            setmon;                  // setup the monitoring information
            repeat(2)                //shift data in
                clockcct;
        end

task setmon;                 //display header and setup monitoring
    begin
        $display("           time  clks  in out  wa1-3  wb1-2");
        $monitor ($time,,,,phase1, phase2,,,,,,shiftin,,,, shiftout,,,,,
            cct.wa1, cct.wa2, cct.wa3,,,,,cct.wb1, cct.wb2);
    end
endtask

task clockcct;               //produce dual-phased clock pulse
    begin
        #d phase1 = 1;           //time step defined by parameter d
        #d phase1 = 0;
        #d phase2 = 1;
        #d phase2 = 0;
    end
endtask
endmodule
```

Example 17-1. Unidirectional transistors: a dynamic mos serial shift register.

17.3 Modeling with Bidirectionals and Strengths – Example

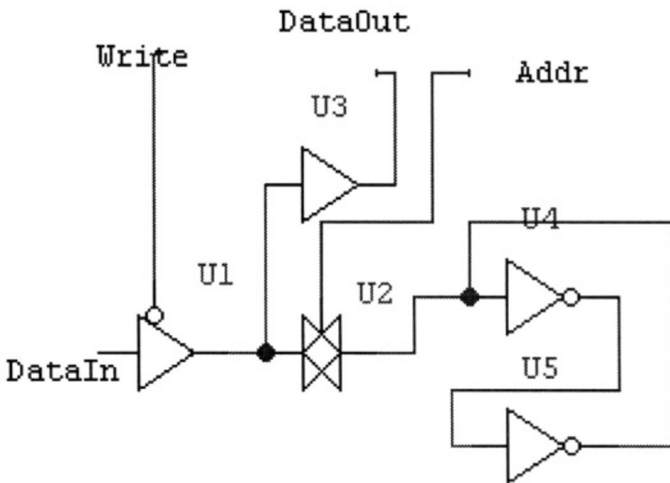


Figure 17-1. Schematics for a Static RAM Cell with Bidirectionals and Strengths

```
//description of a MOS static RAM cell
module sram(dataOut, address, dataIn, write);
        output      dataOut;
        input       address, dataIn, write;

        tri     w1, w3, w4, w43;

    bufif1
    g1(w1, dataIn, write);
    tranif1
    g2(w4, w1, address);
    not (pull0, pull1)
    g3(w3, w4), g4(w4, w3);
    buf
    g5(dataOut, w1);
endmodule

module wave_sram;
        wire dataOut;
        reg  address, dataIn, write;

    //make the sram a submodule and define the interconnections
sram cell(dataOut, address, dataIn, write);

    //define the waveform to drive the circuit
parameter d = 100;
```

Example (continued)

```
        initial
        begin
    #d dis;
    #d address = 1;
    #d dis;
    #d dataIn = 1;
    #d dis;
    #d write = 1;
    #d dis;
    #d write = 0;
    #d dis;
    #d write = 'bx;
    #d dis;
    #d address = 'bx;
    #d dis;
    #d address = 1;
    #d dis;
    #d write = 0;
    #d dis;
end

task dis;           //display the circuit state
```

```
        begin
    $display($time,,
    "addr=%v d_In=%v write=%v d_out=%v",
        address, dataIn, write, dataOut,
        " (134)=%b%b%b", cell.w1, cell.w3, cell.w4,
        "  w134=%v %v %v", cell.w1, cell.w3, cell.w4);
endtask
endmodule
```

Example 17-2. ***Bidirectional transistors and modeling with strengths – a static ram cell.***

17.4 Strength-Levels in Verilog

17.4.1 Overview

Verilog supports eight levels of strength with the following names:

- supply
- strong
- pull
- large
- weak
- medium
- small
- highz

Rule – When multiple drivers are connected, resulting value and strength is determined by combining these together using strength algebra. The higher strength signal wins over the lower level strengths. In the prior example, pull and strong signals are combined to resolve.

17.4.2 Examples of Strength Algebra

Pull0 + Strong1 = Strong1

Pull1 + Strong0 = Strong0

Weak1 + Weak0 = Weak X

WeakX + Pull0= Pull0

Strengths can also be represented by numbers.
7 through 0 for supply to highz.
X Signals may have differing components in them e.g., 35X

17.4.3 Strength Specifications On Gates

Strength specifications can be added onto gates, e.g., (pull0, highz1) buf (out, in);

This allows adding strength information at the gate level. This is also useful for switch-models intermixed with gates as is commonly used while modeling at the structural level.

17.5 Exercises

1. For the algebra of strengths, provide the results of connected drivers with the following values.
   i) Pu1 Pu 0
   ii) Pu1 St0
   iii) 53X St1
   iv) 66X St1
2. Give the output of the simulation of following model that uses switch models in example in section 17.2 with a different test module as below:

```
module NewTestShReg;
wire      shiftout;  //net to receive circuit output value
reg       shiftin;         //register to drive value into circuit
reg       phase1,phase2; //clock driving values

parameter            d = 100;  //define the waveform time step

      shreg          cct (shiftout, shiftin, phase1, phase2);

      initial
            begin :main
                  shiftin = 0;//initialize waveform input stimulus
                  phase1 = 0;
                  phase2 = 0;
                  $monitor("time=%d wa1=%d wb1=%d wa2=%d wb2 = %d wa3=%d out =
%d\n",
                        cct.wa1, cct.wb1, cct.wa2, cct.wb2,
                        cct.wa3, cctout);
                              // setup the monitoring information
                  repeat(2)                  //shift data in
begin
            #d phase1 = 0;                   //time step defined by parameter d
            #d phase1 = 1;
            #d phase1 = 0;
            #d phase2 = 0;
            #d phase2 = 1;
            #d phase2 = 0;
      end
            end

    endmodule
```

18 STANDARD DELAY FORMAT

18.1 Introduction

Standard Delay Format (SDF) is a file specification that assures consistent, accurate, and up-to-date data for timing. The EDA tools can use data created by other tools as input to their own processes via SDF. Sharing data in this way, layout tools can use design constraints identified during timing analysis, and simulation tools can use the post-layout delay data. The EDA tools create, read (to update their design), and write to SDF files.

SDF files support hierarchical timing annotation. A design hierarchy might include several different ASICs (and/or cells or blocks within ASICs), each with its own SDF file.

SDF contains constructs for the description of computed timing data for back-annotation and the specification of timing constraints for forward annotation. There is no restriction on using both sets of constructs in the same file, although these are typically different functions and are present in different files as such. Indeed, some design synthesis tools (such as floorplanning) may need access to computed timing data as well as the timing constraints they are intended to meet. In Figure 1-3, the step in which gate-level netlist with back annotated delays is generated after layout. This is commonly done via SDF files which are generated by the SDF generator that reads layout data as well as the library data, computes the delays and puts it out in the SDF form. The forward annotator can be seen in Figure 12-2, the time and area constraints used in the step in logic synthesis can be in SDF format. Similarly, the examples in Chapter 15 on specify blocks where there are pin to pin delay specifications as well as timing checks can be generated from an SDF file by SDF annotator (reader).

SDF includes constructs for describing the intended timing environment in which a design will operate. For example, you can specify the waveform to be applied at clock inputs and the arrival time of primary inputs.

A delay calculator tool is responsible for generating the delays in the SDF file. It will examine the specific design for which it has been instructed to calculate timing data. The delay calculator must locate, within the design, each region for which a timing model exists and calculate values for the parameters of that timing model. Strategies for doing this vary from technology to technology, but an example would be the location of each occurrence of a physical primitive from an ASIC library and the calculation of its timing properties at its boundary (pin-to-pin timing). Knowledge of the timing models may be obtained by accessing them directly (not shown) or may be built into the delay calculator and/or cell characterization data.

As the timing characteristics of ASICs are strongly influenced by interconnect effects, the figure shows the delay calculator using estimation rules (pre-layout) or actual interconnect data (post-layout). Thus, SDF is suitable for both pre-layout and post-layout application.

The SDF file is brought into the analysis tool through SDF annotator. The annotator matches data in the SDF file with the design description and the timing models. Each module in the design identified in the SDF file must be located and its timing model found. Data in the SDF file for this module must be applied to the appropriate parameters of the timing model.

An annotator may be a part of the tool whereby using access routines or VPI routines it will traverse the compiled Verilog design and match the SDF representation with the design and then generate delays and timing checks as a part of last phase of design compilation. Alternatively, the annotator may operate independently of the analysis tool and convert the data in the SDF file into a format suitable for the tool to read directly. The naming of design objects must be identical in the SDF file and design description. During annotation, inconsistencies between the SDF file and the design description are considered errors.

In addition to the back-annotation of timing data for analysis, SDF supports the forward-annotation of timing constraints and timing checks to design synthesis tools. (Here, we use the term "synthesis" in its broad sense of construction, thus including not only logic synthesis, but floorplanning, layout and routing.) Timing constraints are "requirements" for the design's overall timing properties, often modified and broken down by previous steps in the design process.

For example, the initial requirement might be that the primary clock should run at 50MHz. A static timing analysis of the design might identify the critical paths and the available "slack" time on these paths and pass constraints for these paths to the floorplanning, layout and routing (physical synthesis) tools so that the final design is not degraded beyond the requirement. Alternatively, if after layout and routing, the requirement cannot be met, constraints for the problem paths might be constructed and passed back to a logic synthesis tool so that it can "try again" and leave more slack for physical synthesis.

Constraints may also be originated by an analysis tool alone. Consider a synchronous system in which the clock distribution system is to be synthesized. A

static timing analysis may be able to determine the maximum permissible skew over the distribution network and provide this as a constraint to clock synthesis. In turn, this tool may break down the skew constraint into individual path constraints and forward this to physical synthesis.

Note: The term "timing constraint" is also in use to describe what in SDF are called timing checks. When viewed as statements of the form "this condition must be met or the circuit won't work", they are indeed the same. Perhaps the only distinction is that timing checks are applied to an analysis tool, which is only in a position to check to see if they are met and indicate a violation if they are not, whereas constraints are applied to a synthesis tool, which may adapt its operation to ensure that the specified condition is met.

Timing Models and Delays Supported by SDF

SDF supports both a pin-to-pin and a distributed delay modeling style.

A pin-to-pin modeling style is generally one in which each physical cell in an ASIC library has its timing properties described at its boundary, i.e. with direct reference only to the ports of the cell. The timing model is frequently distinct from the functional part of the model and has the appearance of a "shell", intercepting transitions entering and leaving the functional model and applying appropriate delays to output transitions. The SDF IOPATH construct is intended to apply delay data to input-to output path delays across cells described in this way. The COND construct allows any path delay to be made conditional, that is, its value applies only when the specified condition is true. This allows for state-dependency of path delays where the path appears more than once in the timing model with conditions to identify the circuit state when it applies.

A distributed modeling style is generally one in which the timing properties of the cell are embedded in the description of the cell as a network of modeling primitives. The primitives provided by analysis tools such as simulators and timing analyzers usually have simple timing capabilities built into them, such as the ability to delay an output signal transition. The delay properties of the cell are constructed by the careful arrangement of modeling primitives and their delays. The SDF DEVICE construct is intended to apply delay data to modeling primitives in distributed delay models.

SDF supports the specification of how short pulses propagate to the output of a cell described using a pin-to-pin delay model. A limit can be established for the shortest pulse that will affect the output and a larger limit can be established for the shortest pulse that will appear with its true logical value, rather than appearing as a "glitch" to the unknown state. The SDF PATHPULSE construct allows these limits to be specified as time values. The SDF PATHPULSEPERCENT construct allows these limits to be specified as percentages of the path delay.

Timing Checks in SDF

SDF supports setup, hold, recovery, removal, maximum skew, minimum pulse width, minimum period and no-change timing checks. Library models can specify timing

checks with respect to both external ports and internal signals. Negative values are permitted on timing checks where this is meaningful, although analysis tools that cannot use negative values may substitute a value of zero. The SDF COND construct allows conditional timing checks to be specified.

Interconnects

SDF supports two styles of interconnect delay modeling. The SDF INTERCONNECT construct allows interconnect delays to be specified on a point-to-point basis. This is the most general method of specifying interconnect delay.

The SDF PORT construct allows interconnect delays to be specified as equivalent delays occurring at cell input ports. This results in no loss of generality for wires/nets that have only one driver. However, for nets with more than one driver, it will not be possible to represent the exact delay over each driving-output-to-driven-input path using this construct. Note that for timing checks to operate correctly when interconnect is modeled in this way, the timing models must be constructed to apply the delay to the signal at input ports before they arrive at the timing checks.

SDF allows ports to be specified which are neither external connections of an ASIC library physical primitive nor connections between levels in the design hierarchy. Such "internal nodes" may have no corresponding terminal or net in the physical design but may instead be artifacts of the way the timing and/or functional model is constructed. For specific tools, the use of internal nodes can increase the flexibility and accuracy of the models. However, because the annotator must be able to match data in the SDF file to the models, SDF files referencing internal nodes will not be portable to tools that do not share the same concept of internal nodes or have models constructed without or with different internal nodes.

18.2 SDF Description

18.2.1 Introduction

This chapter describes the standard delay format. For each part of the format, the purpose is discussed, the syntax is specified and an example is presented. A complete, formal definition of the file syntax is contained in Appendix F. SDF files are ASCII text files. Every SDF file contains a header section followed by one or more cell entries. This is in essence an extension to the Specify blocks. Historically, specify blocks were implemented with a fast algorithm at the structure level and some limitations were put in the specification for this. As the timing specifications needed extensions from both design perspective and the tool perspective, a new format SDF was developed. It is possible to extend the Specify blocks to completely support SDF within a normal Verilog HDL.

18.2.2 Syntax

delay_file ::= (DELAYFILE sdf_header cell+)

18.2.3 Example

(DELAYFILE
(SDFVERSION "3.0") (DESIGN "BIGCHIP") (DATE "March 12, 1995 09:46") (VENDOR "Southwestern ASIC") (PROGRAM "Fast program") (VERSION "1.2a") (DIVIDER /) (VOLTAGE 5.5:5.0:4.5) (PROCESS "best:nom:worst") (TEMPERATURE -40:25:125) (TIMESCALE 100 ps) (CELL (CELLTYPE "BIGCHIP") (INSTANCE top) (DELAY (ABSOLUTE (INTERCONNECT mck b/c/clk (.6:.7:.9)) (INTERCONNECT d[0] b/c/d (.4:.5:.6))))) (CELL (CELLTYPE "AND2") (INSTANCE top/b/d) (DELAY (ABSOLUTE (IOPATH a y (1.5:2.5:3.4) (2.5:3.6:4.7)) (IOPATH b y (1.4:2.3:3.2) (2.3:3.4:4.3))))) (CELL (CELLTYPE "DFF") (INSTANCE top/b/c) (DELAY (ABSOLUTE (IOPATH (posedge clk) q (2:3:4) (5:6:7)) (PORT clr (2:3:4) (5:6:7)))) (TIMINGCHECK (SETUPHOLD d (posedge clk) (3:4:5) (-1:-1:-1)) (WIDTH clk (4.4:7.5:11.3)))) (CELL ...))

18.3 Header

18.3.1 Introduction

The header section of an SDF file contains information that relates to the file as a whole. Except for the SDF version, entries are optional, so that, in fact, it is possible to omit most of the header section. The syntax defines a strict order for header entries and those that are present must follow this order.

Most entries are for documentation purposes and do not affect the meaning of the data in the rest of the file. However, the SDF version, hierarchy divider and time scale entries will, if present, change the way in which the file is interpreted.

18.3.2 Syntax

sdf_header ::= sdf_version design_name?
date? vendor? program_name? program_version? hierarchy_divider? voltage?
process? temperature? time_scale?

18.4 Header SubParts

1. **VERSION**

   Syntax

   sdf-version ::= (SDFVERSION QSTRING)
   QSTRING must be one of "1.0", "2.0", "2.1" or "3.0".

   Example

   (SDFVERSION "OVI 3.0")

2. **DESIGN NAME**

   The design name entry allows you to record in the SDF file the name of the design to which the timing data in the file applies. It is for documentation purposes and does not affect the meaning of the data in the file.

Syntax

design_name ::= (DESIGN QSTRING)

QSTRING is a name that identifies the design. Although this entry is not used by the annotator, it is recommended that, if it is included, it should be the name given to the top level of the design description. This is analogous to the CELLTYPE entry, and, in fact, the same name would be used in a cell entry for the entire design (for example, to carry all interconnect delay data). It should not be the instance name of the design in a test-bench; this would rather be used as part of the cell instance path in the INSTANCE entries for all cells.

3. THE DATE ENTRY

Syntax

date ::= (DATE DateString)

Example

(DATE "Friday, September 17, 1993 - 7:30 p.m.")

4. THE VENDOR ENTRY

Syntax

vendor ::= (VENDOR VEndorName)

Example

(VENDOR "Acme Semiconductor")

5. THE PROGRAM NAME ENTRY

This allows you to record in the SDF file the name of the program that created the file. It is for documentation purposes and does not affect the meaning of the data in the file.

Syntax

program_name ::= (PROGRAM ProgramNAme)

Example

(PROGRAM "timcalc")

The program version entry allows you to record in the SDF file the version of the program that created the file. It is for documentation purposes and does not affect the meaning of the data in the file.

6. PROGRAM_VERSION

Syntax

program_version
::=(VERSION QSTRING)

QSTRING is a character string, in double quotes, containing the program version number used to generate the SDF file.

Example

(VERSION "version 1.3")

The hierarchy divider entry specifies which of the two permissible characters are used in the file to separate elements of a hierarchical path.

7. HIERARCHY DIVIDER

Syntax

hierarchy_divider
::= (DIVIDER HCHAR)
HCHAR is either a period (.), or a slash (/).
It should not be in quotes.

Example

(DIVIDER /) ... (INSTANCE
a/b/c) …

In this example, the hierarchy divider is specified to be the slash (/) character and hierarchical paths use / (rather than .) to separate elements.

If the SDF file does not contain a hierarchy divider entry then the default hierarchy divider is the period (.). See also the descriptions of IDENTIFIER and PATH in "Syntax Conventions."

8. THE VOLTAGE ENTRY

Syntax

voltage ::= (VOLTAGE rtriple) ||= (VOLTAGE RNUMBER)
rtriple or RNUMBER indicates the operating voltage (in volts) at which the design timing was calculated or the constraints are to apply.

Example

(VOLTAGE 5.5:5.0:4.5)

9. THE PROCESS ENTRY

The process entry allows you to record in the SDF file the process factor for which the data in the file was computed. It is for documentation purposes and does not affect the meaning of the data in the file.

Syntax

process ::= (PROCESS QSTRING)

QSTRING is a character string, in double quotes, which specifies the process operating envelope.

Example

(PROCESS "best=0.65:nom=l.0:worst=l.8")

10. **THE TEMPERATURE ENTRY**

The temperature entry allows you to record in the SDF file the operating temperature for which the data in the file was computed.

Syntax

temperature ::=(TEMPERATURE rtriple) ||=
(TEMPERATURE RNUMBER)

rtriple or RNUMBER indicates the operating
ambient temperature(s) of the design in degrees Celsius (centigrade).

Example

(TEMPERATURE -25.0:25.0:85.0)

11. **THE TIMESCALE ENTRY**

The timescale entry allows you to specify the units which you are using for all time values in the SDF file.

Syntax

time_scale ::= (TIMESCALE TSVALUE)

TSVALUE is a number followed by a unit. The number can be 1, 10, 100, 1.0, 10.0 or 100.0. The unit can be us, ns or ps representing microseconds, nanoseconds and picoseconds, respectively. A space may optionally separate the number and the unit. TSVALUE should not be in quotes.

Example

(TIMESCALE 100 ps) … (IOPATH (posedge clk) q (2:3:4) (5:6:7)) …

This example indicates that all time values in the file are to be multiplied by 100 picoseconds. Thus, the values supplied in the IOPATH entry are (0.2ns:0.3ns:0.4ns) and (0.5ns:0.6ns:0.7ns). If the SDF file does not contain a timescale entry then all time values in the file will be assumed to be in nanoseconds. This has the same effect as a timescale entry of 1ns.

18.5 Cell Entry

A cell entry identifies a particular "region" or "scope" within a design and contains timing data to be applied there. For example, a cell entry might identify an unique occurrence of an ASIC physical primitive, such as a 2- input NAND gate, in the design and provide values for its timing properties, such as the input-to-output path delays. As well as identifying such design-specific regions, cell entries can identify

all occurrences of a particular ASIC library physical primitive, such as a certain type of gate or flip-flop. Data is applied to all such library-specific regions in the design.

Syntax

cell ::= (CELL celltype cell_instance, timing_spec*)

The celltype and cell_instance fields identify a region of the design. The timing_spec field contains the timing data. These will be discussed in detail below.

Example

(CELL (CELLTYPE "DFF") (INSTANCE a/b/c) (DELAY (ABSOLUTE (IOPATH (posedge clk) q (2:3:4)(5:6:7)))))

An SDF file may contain any number of cell entries (other than zero) in any order. The order of the cell entries is significant only if they have overlapping effect, in other words, if data from two different cell entries applies to the same timing property in the design. In this situation, the cell entries are processed from the beginning of the file towards the end and the data they contain is applied in sequence to whatever region is appropriate to that cell entry. Where data is applied to a timing property previously referenced by the same SDF file, the new data will be applied over the old and the final value will be the cumulative effect, whether the data is applied as a replacement for the old value (absolute delays and timing checks) or is added to it (incremental delays).

18.5.1 The CELLTYPE entry.

This indicates the name of the cell.

Syntax

celltype ::= (CELLTYPE CellNAme)

Example

(CELLTYPE "flop")
(CELLTYPE "and")
(CELLTYPE "main")

In the first example, the cell entry identifies an occurrence of a cell which has the name "DFF" (perhaps a D-type flip-flop).
In this example, the cell entry identifies a "buf' modeling primitive in an analysis tool, perhaps a buf "gate" in a Verilog model.

18.5.2 The Cell Instance Entry

This identifies the region or scope of the design for which the cell entry contains timing data. The name by which this region is known in the design must be consistent with the CELLTYPE entry for the cell. If the annotator locates the region

and finds that its name does not match the CELLTYPE entry, it should indicate an error.

Syntax

cell_instance ::= (INSTANCE PATH?) ||= (INSTANCE WILDCARD)
WILDCARD ::= * // the asterisk character

The first form of the cell instance entry identifies an unique occurrence in the design of the region named in the cell type entry. If, for example, the cell is a physical primitive from an ASIC library, then a single occurrence of that cell on the chip will be identified. To do this, the cell instance entry Cell Entry provides a complete path through the design hierarchy to the cell or region of interest.

Example

(CELL (CELLTYPE "DFF") (INSTANCE al.bl.cl) . . .)

The timing data in the timing specifications of this cell entry apply only to the identified region of the design. If you do not specify PATH, i.e. you leave it blank, the default is the region (hierarchical level) in the design at which the annotator is instructed to apply the SDF file (see "The Annotator" page 3 in chapter 2). This can be useful for gathering all interconnect information into a top-level cell entry.

The second form of the cell instance entry can be used to associate timing data with all occurrences of the specified cell type. Instead of a hierarchical path, specify the wildcard character (*) after the INSTANCE keyword, as shown below.

Example

(CELL (CELLTYPE "DFF") (INSTANCE *))

18.6 Timing Specifications

There are three types of timing specifications that are identified by the DELAY, TIMINGCHECK, and TIMINGENV keywords.

Syntax

timing_spec ::= del_spec ||= tc_spec ||= te_spec
del_spec ::= (DELAY deltype+)
tc_spec ::= (TIMINGCHECK tchk_def+)
te_spec ::= (TIMINGENV te_def+)

Timing specifications that start with the DELAY keyword associate delay values with input-to-output paths, input ports, interconnects, and device outputs. They can also provide narrow-pulse propagation data for input to-output paths.

Syntax

del_spec::= (DELAY deltype+)

18.6.1 Delay Type – Absolute

The ABSOLUTE keyword introduces delay data that replaces existing delay values in the design during annotation.

Any number of deltype entries may appear in a del_spec entry. Each deltype will be a PATHPULSE or PATHPULSEPERCENT entry, specifying how pulses will propagate across paths in this cell, or ABSOLUTE or INCREMENT delay definition entries, containing delay values to be applied to the region identified by the cell.

Syntax

```
deltype::= ( ABSOLUTE del_def+ )
         ||= (INCREMENTdel_def+ )
         ||= (PATHPULSEinput_output_path?valuevalue? )
      ||= ( PATHPULSEPERCENT input_ouput_path? value value? )
```

Example

```
(CELL (CELLTYPE "DFF") (INSTANCE a.b.c) (DELAY (ABSOLUTE (IOPATH (posedge
clk) q (22:28:33) (25:30:37)) (PORT clr (32:39:49) (35:41:47)) ) ))
```

Negative delay values can be specified for absolute delays to accommodate certain styles of ASIC cell characterization. However, note that not all analysis tools will be able to make sense of negative delays and some may set them to zero.

18.6.2 The INCREMENT keyword

The INCREMENT keyword introduces delay data that is added to existing delay values in the design during annotation.

Syntax

```
(INCREMENT del_def+ )
```

The delay definition entries, del_def, contain the actual data and describe where it belongs in the design. The same delay definition constructs are used for increment and absolute delays.

Example

```
(CELL (CELLTYPE "DFF") (INSTANCE a.b.c) (DELAY (INCREMENT (IOPATH (posedge
clk) q (-4::2) (-7::5)) (PORT clr (2:3:4) (5:6:7))) ) )
```

Negative delay values can be specified for increment delays, in which case, of course, the value existing in the design will be reduced. For negative delays, note that not all analysis tools will be able to make sense of negative delays and may set them to zero.

Both absolute and increment delays are described by the same group of delay definition constructs.

18.6.3 The PATHPULSE Entry

The PATHPULSE entry represents narrow-pulse propagation limits associated with a legal path between an input port and an output port of a device. These limits determine whether a pulse of a certain width can pass through the device and appear at the output.

Syntax

```
( PATHPULSE input_output_path? value value? )
        input_output_path ::= port_instance port_instance
```

The first port_instance of input_output_path is an input or a bidirectional port.

The second port_instance of input_ouput_path is an output or a bidirectional port.

If input_output_path is omitted, then the data supplied refers to all input to-output paths in the region identified by the cell entry. The annotator must locate all paths that are able to model narrow-pulse propagation in the applicable timing model and apply the supplied data. The first value, in time units, is the pulse rejection limit. This limit defines the narrowest pulse that can appear at the output port of the specified path. Any narrower pulse does not appear at the output.

The second value, in time units, is the X limit. This limit defines the minimum pulse width necessary to drive the output of the specified path to a known state; a narrower pulse causes the output to enter the unknown (X) state or is rejected (if smaller than the pulse rejection limit). Note that the X limit must be greater than the pulse rejection limit to carry any significance.

If you specify only one value, both limits are set to that value. In all cases value can be either a single number or a triple, but must not be negative.

Example

```
(INSTANCE x) (DELAY (PATHPULSE il ol (13) (21)) )
```

In this example of a simple buffer cell, the pulse rejection limit is specified as 13 time units and the X limit is specified as 21 time units. It is assumed that the high-to-low and low-to-high delays from i1 to ol are the same. The first pulse, being shorter than 13, is rejected. The second pulse, being at least 13, but shorter than 21, appears at the output as an X. The third pulse, being at least 21, is passed to the output.

When narrow pulses arrive at an output due to changes at different inputs (rather than two changes at the same input, as in the above example), the two paths from the inputs to the output may have different limits. The assumption made in SDF is that the analysis tool will use the data for the path that terminated the pulse to control the pulse's appearance at the output.

Example

```
(INSTANCE x) (DELAY (ABSOLUTE
(IOPATH a y (45) (37)) (IOPATH b y (43) (35)) ) (PATHPULSE a y (13)
(24)) (PATHPULSE b y (15) (21)) )
```

Note that the order in which the inputs changed is of no consequence; pulse propagation is controlled by the data associated with the path through which the transition propagates that ends the output pulse.

If a path has not been given data for its pulse rejection or X limits, then the analysis tool assumes a pulse rejection limit and an X limit equal to the path delay. Thus, if this path terminates a narrow pulse, the pulse will be rejected if it is shorter than the path delay or otherwise passed.

18.6.4 The PATHPULSEPERCENT Entry

This is the same as PATHPULSE but the values are expressed as a percentage (%) of the cell path delay from the input to the output.

Syntax

(PATHPULSEPERCENT input_output_path? value value?)

Neither value should be greater than 100.0. To have any effect, the second value (X limit) must be greater than the first value (pulse rejection limit).

Example

(INSTANCE x) (DELAY (ABSOLUTE (IOPATH a y (45) (37))) (PATHPULSEPERCENT a y (25) (35)))

In this example, the pulse rejection limit is specified as 25% of the delay time from a to y and the X limit is specified as 35% of this delay. If more than one delval is specified in the delval-list of an IOPATH entry, the analysis tool selects that corresponding to the transition than ended the pulse. So, for a high-going output pulse, which ends with a high-to-low transition, the percentages are applied to the high-to-low delay of the path. In the above example, where the high-to-low delay is 37, the pulse rejection limit is 25% of 37 and the X limit is 35% of 37. The data used for pulse control comes from the path that caused the pulse to terminate (in the same way as for the PATHPULSE construct).

Note that if the analysis tool is able to model narrow-pulse propagation with different limits for each output transition, the tool can pre-compute the limit values from the percentages and path delay values. The annotator, however, cannot do this as new values for path delays may be supplied after the PATHPULSEPERCENT entry is processed.

18.7 Delay Definitions

Syntax

del_def ::= (IOPATH port_spec port_instance(RETAIN delval-list)* delval-list)
||= (COND QSTRING? conditional_port_expr (IOPATH port_spec port_instance (RETAIN
delval_list)* delval_list))

||= (CONDELSE (IOPATH port_spec port_instance (RETAIN delval_list)* delval_list))
||= (_PORT port_instance delval_list) ||= (INTERCONNECT
port_instance port_instance delval_list) ||= (DEVICE port_instance? delval-list)

In the syntax descriptions above, you will see that each construct uses delval-list to specify the operating values to be applied. The section "Data Values" on page 4-7 provides a formal definition of delval-list along with related syntax constructs. However, here we discuss delval_list in the context of specifying delay and pulse control data for the various delay constructs in SDF.

18.7.1 The Delay Data

This data in each delay definition entry is specified in a list of delvals.

Syntax

delval_list
::= delval ||= delval delval ||= delval delval delval ||= delval delval
delval delval delval? delval? ||= delval delval delval delval delval
delval delval delval? delval? delval? delval? delval?

The number of delvals in the delval_list can be one, two, three, six or twelve. Note, however, that the amount of data you include in a delay definition entry must be consistent with the analysis tool's ability to model that kind of delay. For example, if the modeling primitives of a particular tool can accept only three delay values, perhaps rising, falling and "Z" transitions, you should not attempt to annotate different values for 0*1 and Z*l transitions or for 1*Z and 0*Z transitions. It is recommended that in such situations annotators combine the information given in some documented manner and issue a warning.

The following paragraphs define the semantics of delval_lists of various lengths:

1. If twelve delvals are specified in delval_list, then each corresponds, in sequence, to the delay value applicable when the port (for IOPATH and INTERCONNECT, the output port) makes the following transitions:

   0*1,1*0,0*Z,Z*1,1*Z,Z*0,0*X,X*1,1*X,X*0,X*Z,Z*X
2. If fewer than twelve delvals are specified in delval_list, then the table below shows how the delays for each transition of the port are found from the values given.
3. If only two delvals are specified, the first ("rising") is denoted in the table by 01 and the second ("falling") by 10.
4. If three delvals are specified, the first and second are denoted as before and the third, the "Z" transition value, by -Z.
5. If six delvals are specified, they are denoted, in sequence, by 01, 10, 0Z, Zl, 1Z and Z0.

6. If a single delval is specified, it applies to all twelve possible transitions. This is not shown in the table.

In a delval_list, any delvals can be null, that is, the parentheses enclosing the RNUMBER or rtriple are empty (see "Data Values" on page 4-7). The meaning of this is the same as missing numbers in an rtriple: no data is supplied and values should not be changed by the annotator. Such null delvals act as "placeholders" to allow you to specify delvals further down the list.

Example

(IOPATH i3 ol () () (2:4:5) (4:5:6) (2:4:5) (4:5:6))

In this example, 0*1 and 1*0 delay values are not specified and might not even be present in the timing model. A delval-list consisting of nothing but null delvals is permitted by the syntax and has no effect.

0*1 1*0 0*Z
Z*l 1*Z Z*00*X X*1 1*X X*OX*Z Z*X

01 10-Z 01-Z10 min(01,-Z)
01 min(l0,-Z) 10-Zmin(01,10)

01 10 01 01 10 10 01 01 10 10 max(01,10)
min(01,10)

01 10 0ZZl 1ZZ0 min(01,0Z) max(01,Zl) min(10,lZ) max(10,Z0)
max(0Z,lZ) min(Z0,Zl)

Transition 2 3 6 Number of delvals in delval_list.

18.7.2 Delay Value

In delval_lists of length six and twelve, it is permissible to omit trailing null delvals. Thus, a list of four delvals, for example, provides data for the 0*1, 1*0, 0*Z and Z*l transitions, but not for the 1*Z, Z*0 transitions. Note that omitting three delvals is going too far as a mapping is defined above for an delval_list of three delvals onto all six transitions. Each delval is either an rvalue or a group of two or three rvalues enclosed in parentheses.

Syntax

delval ::= rvalue ||= (rvalue rvalue) ||=
(rvalue rvalue rvalue)
rvalue ::= (RNUMBER?) ||= (rtriple?)

When a single rvalue is used, it specifies the delay value. When two rvalues in parentheses are used, the first rvalue specifies the delay, as if a single rvalue were given. The second specifies both the pulse rejection limit, or "r-limit", associated with this delay, and the X-limit, or "e-limit". When three rvalues are used, the first

specifies the delay, the second specifies the pulse rejection limit, or "r-limit", and the third specifies the X-limit, or "e-limit". This allows pulse control data to be associated in a uniform way with all types of delays in SDR Note that since any rvalue can be an empty pair of parentheses, each type of delay data can be annotated or omitted as the need arises. Each rvalue is either a single RNUMBER or an rtriple, containing three RNUMBERs separated by colons, in parentheses.

The use of single RNUMBERs and rtriples should not be mixed in the same SDF file. All RNUMBERs can have negative, zero or positive values.

The use of triples in SDF allows you to carry three sets of data in the same file. Each number in the triple is an alternative value for the data and is typically selected from the triple by the annotator or analysis tool on an instruction from the user. The prevailing use of the three numbers is to represent minimum, typical and maximum values computed at three process/operating conditions for the entire design. Any one or any two (but not all three) of the numbers in a triple may be omitted if the separating colons are left in place. This indicates that no value has been computed for that data, and the annotator should not make any changes if that number is selected from the triple. For absolute delays, this is not the same as entering a value of 0.0.

The following sections describe delay definition entries.

18.7.3 The IOPATH Entry

The IOPATH entry represents the delays on a legal path from an input/ bidirectional port to an output/bidirectional port of a device. Each delay value is associated with a unique input port/output port pair.

Syntax

(IOPATH port_spec port_instance delval-list)
port_spec is an input or a bidirectional port and can have an edge identifier.
port_instance is an output or a bidirectional port. It cannot have an edge identifier.

Delay data for the different transitions at the path output port are conveyed by supplying an ordered list of values as described above. delval_list is the IOPATH delay data from port_spec to port_instance.

If the timing model includes conditions (state dependency) for the path delay between the two specified ports, the specified delval is still applied. If the model includes more than one delay path, each distinguished by its conditions, then the data applies to all of them. This has the same effect as specifying all paths (using the COND or CONDELSE keyword with IOPATH as described below) with the same IOPATH delay delval_list.

Example

```
(INSTANCE x.y.z) (DELAY (ABSOLUTE (IOPATH (posedge i1) ol (2:3:4) (4:5:6))
(IOPATH i2 ol (2:4:5) (5:6:7)) (IOPATH i3 ol () () (2:4:5) (4:5:6) (2:4:5) (4:5:6)) ) )
```

18.7.4 Conditionals

The COND keyword allows the specification of conditional (state dependent) input-to-output path delays.

Syntax

(COND QSTRING? conditional_port_expr (IOPATH port_spec port_instance delval_list))

QSTRING is an optional symbolic name that can stand in for the expression itself for annotators that operate by matching named placeholders in the model to SDF constructs. See "Condition Labels", below, for a full explanation.

conditional_port_expr is the description of the state dependency of the path delay. The syntax of conditional_port_expr is shown in "Conditions for Path Delays" on page 4-9. The perceptive reader will notice that this expression evaluates to a logical signal, rather than a boolean. The intent is that the analysis tool should treat a logical zero as FALSE and any other logical value (1, X or Z) as TRUE and that a particular conditional path delay in the timing model is used only if the condition is TRUE.

port_instance and delval_list have exactly the same meaning as in IOPATH entries without the COND keyword as described above, except that the annotator must locate a path delay with a condition matching the one specified and apply the data only to that. Other path delays from the same input port to the same output port but with different conditions in the timing model will not receive the data. Annotators may differ in their capabilities to match a condition in SDF to conditions in the timing model. Where the analysis tool uses the same syntax as SDF (derived from the Verilog language), the annotator may require an exact character-for character match in the string representations of the conditions.

Example

(INSTANCE x) (DELAY (ABSOLUTE (COND b (IOPATH a y (0.21) (0.54))) (COND ~b (IOPATH a y (0.27) (0.34))) (COND a (IOPATH b y (0.42) (0.44))) (COND ~a (IOPATH b y (0.37) (0.45)))))

The CONDELSE keyword allows the specification of default delays for conditional paths. The default delay is the delay that will be in force if, during the simulation or analysis, none of the conditions specified for the path in the model are TRUE but a signal must still be propagated over the path.

Syntax

(CONDELSE (IOPATH
port_spec port_instance delval_list))

This construct should be used only where the cell timing model includes an explicit mechanism for providing default delays. The annotator should match this SDF construct to such a mechanism in the model. It should not attempt to locate conditions for the path which have not been specified in COND constructs.

Annotators may operate by mapping constructs in the SDF file into symbolic names, locating placeholders with those names in the models and applying values from the SDF file to the variables associated with those placeholders. (An example of this is the annotator for VITAL models in a VHDL simulator.) To ease the problem of mapping a conditional_port_expr construct (or the timing_check_condition construct in timing checks, later) into symbolic names, these can optionally be preceded by a QSTRING.

Clearly, for a tool that uses a name mapping annotation scheme, models must be constructed so as to contain the correct placeholders. Therefore, the mapping algorithm of the tool's annotator must be clearly documented and available to users. The description of the mapping must include the way in which the QSTRING is used in constructing the name. For example, it may be appended to a name constructed from other information in the SDF file such as the type of construct, port names, etc. The description should also explain what will happen if the QSTRING is absent in a conditional construct and what will happen in certain timing checks where two QSTRINGs are possible.

The intent of SDF is that the QSTRING should stand in place of the conditional_port_expr or timing_check_condition in constructing unique placeholder names for each state or condition in which a timing property might have a different annotated value.

18.7.5 The RETAIN Entry

The RETAIN entry represents the time for which an output/bidirectional port will retain its previous logic value after a change at a related input/ bidirectional port. This is commonly used on paths from the address or select inputs to the data outputs of memory and register file circuits.

Syntax

(IOPATH port_spec port_instance (RETAIN delval_list)* delval-list) (COND QSTRING? conditional_port_expr (IOPATH port_spec port_instance (RETAIN delval_list)* delval_list)) (CONDELSE (IOPATH port_spec port_jnstance (RETAIN delval_list)* delval_list))

port_spec is an input or a bidirectional port and can have an edge identifier.

port-instance is an output or a bidirectional port. It cannot have an edge identifier. Delay data for the different transitions at the path output port are conveyed by supplying an ordered list of values as described above in "Specifying Delay Values" on page 16.

delval-list is the retain time data from port_spec to port_instance.

This construct should be used only where the cell timing model includes an explicit mechanism for providing retain times. The annotator should match this SDF construct to such a mechanism in the model.

The delays in delval_list for consecutive RETAIN statements must be strictly monotonically increasing.

Example

(IOPATH addr[13:0] do[7:0]

(RETAIN (4:5:7) (5:6:9))

In this example, the retain time of the bus do[7:0] with respect to changes on the bus addr[7:0] is described. It is assumed that the model for this cell contains path delays from addr to do and also a modeling construct to receive the retain times written so that after the retain time, do goes to the X state. The first delval, (4:5:7), is the "rising" time and will be used for do going from low to X. The second delval, (5:6:9), is the "falling" time and will be used for do going from high to X.

As with IOPATH entries, RETAIN entries can be made conditional or state dependent by the use of the COND and CONDELSE keywords.

18.7.6 The PORT Entry

The PORT entry is for the specification of interconnect delays (actual or estimated) that are modeled as delay at input ports. The start point for the delay path (the driving output port) is not specified.

Syntax

(PORT port_instance delval_list)
port_instance is an input or bidirectional port.
delval_list is
the PORT delay of the port_instance.

Example

(INSTANCE c) (DELAY
(ABSOLUTE (PORT rl.a (0.01:0.02:0.03)) (PORT r2.a (0.03:0.04:0.05))
))

Analysis tools must apply delay values obtained from SDF PORT entries before timing checks are applied. Thus, this construct models delay in the physical interconnect between the driver and the driven cell port.

18.7.7 The INTERCONNECT Entry

The INTERCONNECT entry is for the specification of interconnect delays (actual or estimated) that are modeled independently for each driver-to driven path. Both start and end points for the delay path are specified.

Syntax

(INTERCONNECT port_instance port_instance delval_list)
The first port-instance is an output or bidirectional port.
The second port_instance is an input or bidirectional port.

delval_list is the INTERCONNECT delay between the output and input ports.

Example

(INSTANCE top) (DELAY (ABSOLUTE (INTERCONNECT d1.y c.r1 .a (0.01:0.02:0.03)) (INTERCONNECT d1.y c.r2.a(0.03:0.04:0.05)) (INTERCONNECT d1.y r3.a (0.05:0.06:0.07))(INTERCONNECTb.d2.yc.r1.a(0.04:0.05:0.06))(INTERCONNECT b.d2.y c.r2.a (0.02:0.03:0.04)) (INTERCONNECT b.d2.y r3.a (0.02:0.03:0.04))))

Although INTERCONNECT entries are the most general way in which interconnect delays can be expressed, some analysis tools may not be able to model independent delay values over each driver-to-driven path on a net with more than one driver. Such tools may map INTERCONNECT entries into equivalent input port delays (such as would directly arise from PORT entries), sometimes losing information in the process. Even tools which can model independent delays over each path may do so less efficiently than input port delays. Writers of SDF files should bear this in mind when choosing whether to use PORT entries or INTERCONNECT entries or a combination of both to model interconnect delay.

18.7.8 The DEVICE Entry

The DEVICE entry represents the delay of all paths through a cell to the specified output port. This construct is intended primarily for use with distributed timing models where the cell to which it is applied is a modeling primitive. If it is used at a higher level in the hierarchy, then the effect is to apply the delay data to all input-to-output paths across the cell that terminate at the specified port.

Syntax

(DEVICE port_instance? delval_list)

port_instance is optional and, if present, specifies the output port to which the delay data is to be applied. If a cell has more than one output, you can therefore include several DEVICE entries in a single CELL entry, each indicating the desired output port using port_instance, and attach different delay data to each output. If port_instance is omitted, all paths to all output ports of the region identified in the cell entry receive the same delay data.

delval-list is the delay data. The number of triples in delval-list must correspond to the capabilities of the modeling primitives of the target analysis tool. For example, Verilog "gates" can accept one, two, or in some cases, three delay values, but never six or twelve.

Example

(CELL (CELLTYPE "buf") (INSTANCE rsl.nandl.bufa) (DELAY (ABSOLUTE (DEVICE (1:3:8) (4:5:7))))) (CELL (CELLTYPE "buf") (INSTANCE rsl.nandl.bufb) (DELAY (ABSOLUTE (DEVICE (2:4:9) (6:8:12)))))

In this example, a 2-input NAND gate model, nan2, is constructed in a distributed delay style from two buffer primitives, bufa and bufb, and a NAND gate primitive, nand. Two such NAND gates, nandl and nand2, are instantiated to create a design

for an RS latch. This is then instantiated in a higher level of the design as rs1. The SDF file demonstrates the annotation of delays to the a-to-y and b-to-y paths through the top NAND gate. The first of these defines the input-to-output path delay from sb to q of the RS latch; the second contributes to the rb to q delay. The delay on bufa also contributes to the sb-to-qb delay.

Example

(CELL (CELLTYPE "rslatch") (INSTANCE rs1) (DELAY (ABSOLUTE (DEVICE q (1:3:8) (4:5:7)) (DEVICE qb (2:4:9) (6:8:12)))))

In this example, the same RS latch is described in a pin-to-pin modeling style. Two nand gate primitives are connected to form the functional part of the model and all timing information is described separately in a timing model of whatever form the analysis tool requires. Typically, this timing model would specify input-to-output delay paths from sb to q, rb to to qb and rb to qb. The above excerpt from an SDF file annotates values for all paths to the q and qb outputs. It will have exactly the same effect as the following:

(CELL (CELLTYPE "rslatch") (INSTANCE rs1) (DELAY (ABSOLUTE (IOPATH sb q (1:3:8) (4:5:7)) (IOPATH rb q (1:3:8) (4:5:7)) (IOPATH sb qb (2:4:9) (6:8:12)) (IOPATH rb qb (2:4:9) (6:8:12)))))

18.8 Timing Check Entries

Timing specifications that start with the TIMINGCHECK keyword associate timing check limit values with specific cell instances.

Syntax

tc_spec ::= (TIMINGCHECK tchk_def+)

Any number of tchk_def entries may appear in a tc_spec entry. Each tchk_def will be a SETUP, HOLD, SETUPHOLD, RECOVERY, REMOVAL, RECREM, SKEW, WIDTH, PERIOD or NOCHANGE timing check entry, containing timing check limit values for this cell entry. Timing check entries specify limits in the way in which a signal can change or two signals can change in relation to each other for reliable circuit operation. EDA analysis tools use this information in different ways: Simulation tools issue warnings about signal transitions that violate timing checks. Timing analysis tools identify delay paths that might cause timing check violations and may determine the constraints for those paths.

Syntax

tchk_def ::= (SETUP port_tchk port_tchk value) ||= (HOLD port_tchk port_tchk value) ||= (SETUPHOLD port_tchk port_tchk rvalue rvalue) ||= (SETUPHOLD port_spec port_spec rvalue rvalue scond? ccond?) ||= (RECOVERY port_tchk port_tchk value) ||= (REMOVAL port_tchk port_tchk value) ||= (RECREM port_tchk port_tchk rvalue rvalue) ||= (RECREM port_spec port_spec rvalue rvalue scond? ccond?) ||= (SKEW port_tchk port_tchk rvalue) ||=

(WIDTH port_tchk value) ||= (PERIOD port_tchk value) ||= (NOCHANGE port_tchk port_tchk rvalue rvalue)

The COND keyword allows the specification of conditional timing checks. Its use is rather different from the specification of conditional input-output path delays described in "Conditional Path Delays" on page 19 in that the condition is associated with the specification of a port rather than the entry as a whole.

Syntax

port_tchk::= port_spec ||= (COND QSTRING? timing_check_condition port_spec)

timing_check_condition is the description of the state dependency of the timing check. The perceptive reader will notice that this expression evaluates to a logical signal, rather than a boolean. The intent is that the analysis tool should treat a logical zero as FALSE and any other logical value (1, X or Z) as TRUE and that a particular conditional timing check in the timing model is used only if the condition is TRUE.

The annotator must locate in the timing model a timing check with conditions matching those specified. Other timing checks of the same kind but with different conditions from the SDF entry will not receive the data. SDF timing check entries with no conditions match any timing check in the model of the same kind and between the ports specified in the SDF entry.

An alternative syntax is available for SETUPHOLD and RECREM timing checks. This associates the conditions with the "stamp" and "check" events in the analysis tool rather than the port_spec. Separate conditions can be supplied for the "stamp" and "check" events using the SCOND and CCOND keywords. Note, SCOND or CCOND or both SCOND and CCOND take precedence over COND.

Syntax

scond
::= (SCOND QSTRING? timing_check_condition) ccond ::= (CCOND QSTRING?
timing_check_condition)

For the setup phase of a setuphold timing check, the "stamp" condition applies to the data port and the "check" condition to the clock or gate port. For the hold phase, the "stamp" condition applies to the clock or gate port and the "check" condition to the data port.

These conditions restore flexibility in expressing conditions that is lost when SETUP and HOLD are combined into SETUPHOLD, or when RECOVERY and REMOVAL are combined into RECREM. For example, here are separate SETUP and HOLD statements for the same clock and data signals, but with the condition attached to the clock in one case, and to the data in the other:

```
(SETUP d (COND enb clk) (5))
(HOLD (COND enb d) clk (7))
```

These conditions cannot be combined into a single SETUPHOLD as shown here:

(SETUPHOLD (COND enb d) (COND enb clk) (5) (7))

This is because there is no way to specify that the condition should only apply to signal clk for SETUP checks, and only to signal d for HOLD checks. The SCOND and CCOND fields provide this capability. By definition, the CCOND field defines a condition for the check event (the 2nd event):

(SETUPHOLD d clk (5) (7) (CCOND enb))

Any port_spec can be qualified with an edge identifier as follows:

Syntax
port_spec
::= port_instance ||= port_edge

port_edge ::= (EDGE-IDENTIFIER port_instance)

This will be termed an "edge specification". When the annotator is locating a timing check at specified ports in the timing model, it must match the edge specification as well as the port names. A port without an edge specification in SDF matches any edge specification in the model.

Example

(CELL (CELLTYPE "DFF") (INSTANCE a.b.c) (TIMINGCHECK(SETUP din (posedge clk) (3:4:5.5)) (HOLD din (posedge clk) (4:5.5:7))))

This example shows a cell entry which provides values for setup and hold timing checks with respect to the rising edge of the clock signal.

In the syntax descriptions of the timing check constructs, you will see that either rvalue or value is used to specify the timing check limit to be applied. Although rvalue may be negative, value must be zero or positive.

Each rvalue or value may be a single value (RNUMBER or NUMBER, respectively) or three values separated by colons, (an rtriple or triple) representing three sets of data for minimum, typical and maximum delay conditions. However, the use of single RNUMBER/NUMBERs and rtriple/ triples should not be mixed in the same SDF file.

The use of triples in SDF allows you to carry three sets of data in the same file. Each number in the triple is an alternative value for the data and is typically selected from the triple by the annotator or analysis tool on an instruction from the user. The prevailing use of the three numbers is to represent minimum, typical and maximum values computed at three process/operating conditions for the entire design. Any one or any two (but not all three) of the numbers in a triple may be omitted if the separating colons are left in place. This indicates that no value has been computed

for that data, and the annotator should not make any changes if that number is selected from the triple.

SETUPHOLD, RECREM and NOCHANGE timing checks have two rvalues, the first for the setup limit and the second for the hold limit.

18.8.1 The SETUP Entry

The SETUP entry specifies limit values for a setup timing check.

Setup and hold timing checks are used to define a time interval during which a "data" signal must remain stable in order for a transition of a "clock" or "gate" signal to store the data successfully in a storage device (flip-flop or latch). The setup time limit defines the part of the interval before the clock transition; the hold time limit defines the part of the interval after the clock transition. Any change to the data signal within this interval results in a timing violation. To shift the interval with respect to the clock transition, either the setup time or the hold time can be negative; however, their sum must always be greater than zero.

Syntax

(SETUP port_tchk port_tchk value)

The first port_tchk identifies the data port. If it includes an edge specification, then the data is for a setup time check with respect only to the specified transition at the data port.

The second port_tchk identifies the clock/gate port and will normally include an edge specification to identify the active edge of the clock or the active-to-inactive transition of the gate. value is the SETUP time limit between the data and clock ports and must not be negative.

Example

(INSTANCE x.a) (TIMINGCHECK (SETUP din (posedge clk) (12)))

As with all port_tchks, the COND construct can be used to specify conditions associated with the setup timing check.

18.8.2 The HOLD Entry

The **HOLD entry** specifies limit values for a hold timing check.

Syntax

(HOLD port_tchk port_tchk value)
The first port_tchk identifies the data port.
The second port-tchk identifies the clock port.
value is the HOLD time between the data and clock events and must not be negative..

Example

(INSTANCE x.a) (TIMINGCHECK (HOLD din (posedge clk) (9.5))...)

As with all port_tchks, the COND construct can be used to specify conditions associated with the hold timing check.

18.8.3 The SETUPHOLD Entry

The SETUPHOLD entry specifies setup and hold limits in a single entry.

Syntax

(SETUPHOLD port_tchk port_tchk rvalue rvalue) (SETUPHOLD port_spec port_spec rvalue rvalue scond? ccond?)

The first port_tchk or port_spec identifies the data port. The second port_tchk or port_spec identifies the clock port. As with all port_tchks, the COND construct can be used in the first form of the setuphold timing check to specify conditions associated with the ports. The first rvalue is the setup time and the second rvalue is the hold time. Either can be negative, however their sum must be greater than zero. In the second syntax form, scond and ccond are the "stamp" and "check" conditions as described above in "Conditional Timing Checks".

Example

(INSTANCE x.a) (TIMINGCHECK (SETUPHOLD (COND ~reset din) (posedge clk) (12) (9.5)))

This SDF entry will match setup and hold timing checks in the model that are conditional on ~reset at the time the din port changes. At this time in the analysis tool, -reset must evaluate to TRUE, i.e., the reset signal must be in the zero, X or Z states, for the checks to be performed.

Example

(INSTANCE x.a) (TIMINGCHECK (SETUPHOLD din (posedge clk) (12) (9.5) (CCOND ~reset)))

This SDF entry, using the second syntax form, will match setup and hold timing checks in the model that are conditional on -reset at the time of the "check" event. For the setup phase of the check, this will be when the clk port undergoes a posedge transition. For the hold phase of the check, this will be when the din port undergoes any transition.

18.8.4 The RECOVERY Entry

The RECOVERY entry specifies limit values for recovery timing checks. A recovery timing check is a limit of the time between the release of an asynchronous control signal from the active state and the next active clock edge, for example between clearbar and the clock for a flip-flop. If the active edge of the clock occurs too soon after the release of the clearbar, the state of the flip-flop will become uncertain—it could be the value set by the clearbar, or it could be the value clocked into the flip-

flop from the data input. In other respects, a recovery check is similar to a setup check.

Syntax

(RECOVERY port_tchk port_tchk value)

The first port_tchk refers to the asynchronous control signal and will normally have an edge identifier associated with it to indicate which transition corresponds to the release from the active state.

The second port_tchk refers to the clock (flip-flops) or gate (latches). This will also normally have an edge identifier to indicate the active edge of the clock or the closing edge of the gate. value is the recovery limit value and must not be negative. It is the time it takes a device to recover after an extraordinary operation, such as set or reset, so that it can reliably return to normal operation, such as clocking in of new data.

Example

(INSTANCE x.b) (TIMINGCHECK (RECOVERY (posedge clearbar) (posedge clk) (11.5)))

As with all port_tchks, the COND construct can be used to specify conditions associated with the recovery timing check.

18.8.5 The REMOVAL Entry

The REMOVAL entry specifies limit values for removal timing checks. A removal timing check is a limit of the time between an active clock edge and the release of an asynchronous control signal from the active state, for example between the clock and the clearbar for a flip-flop. If the release of the clearbar occurs too soon after the active edge of the clock, the state of the flip-flop will become uncertain—it could be the value set by the clearbar, or it could be the value clocked into the flip-flop from the data input. In other respects, a removal check is similar to a hold check.

Syntax

(REMOVAL port_tchk port_tchk value)

The first port_tchk refers to the asynchronous control signal and will normally have an edge identifier associated with it to indicate which transition corresponds to the release from the active state. The second port_tchk refers to the clock (flip-flops) or gate (latches). This will also normally have an edge identifier to indicate the active edge of the clock or the closing edge of the gate. value is the removal limit value and must not be negative. It is the time for which an extraordinary operation, such as set or reset, must persist to insure that a device will ignore any normal operation, such as clocking in of new data.

Example

(INSTANCE x.b)

(TIMINGCHECK (REMOVAL (posedge clearbar) (posedge clk) (6.3)))

As with all port_tchks, the COND construct can be used to specify conditions associated with the recovery timing check.

18.8.6 The RECREM Construct

The RECREM construct specifies both recovery and removal limits in a single entry.

Syntax

(RECREM port_tchk port_tchk rvalue rvalue) (RECREM port_spec port_spec rvalue rvalue scond? ccond?)

The first port_tchk or port_spec identifies the asynchronous control port.

The second port_tchk or port_spec identifies the clock (for flip-flops) or gate (for latches) port.

As with all port_tchks, the COND construct can be used in the first form of the recovery/removal timing check to specify conditions associated with the ports.

The first rvalue is the recovery time and the second rvalue is the removal time. Either can be negative, however their sum must be greater than zero. In the second syntax form, scond and ccond are the "stamp" and "check" conditions as described above in "Conditional Timing Checks" on page 332.

Example

(INSTANCE x.b) (TIMINGCHECK (RECREM (posedge clearbar) (posedge clk) (1.5) (0.8)))

This example specifies a recovery time of 1.5 and a removal time of 0.8. The recovery time limit (1.5 time units) defines the part of the interval before the clock transition; the removal time limit (0.8 time units) defines the part of the interval after the clock transition. Any change to the clearbar signal within this interval results in a timing violation.

18.8.7 The SKEW Entry

The SKEW entry specifies limit values for signal skew timing checks. A signal skew limit is the maximum allowable delay between two signals, which if exceeded causes devices to behave unreliably.

Syntax

(SKEW port_tchk port_tchk rvalue) The first port_tchk is the stamp event and can include an edge specification. The second port_tchk is the check event and can include an edge specification. rvalue is the maximum skew limit.

Example

(INSTANCE x) (TIMINGCHECK (SKEW (posedge clk1) (posedge clk2) (6)))

As with all port_tchks, the COND construct can be used to specify conditions associated with the skew timing check.

18.8.8 The WIDTH Entry

The WIDTH entry specifies limits for a minimum pulse width timing check. The minimum pulse width timing check is the minimum allowable time for the positive (high) or negative (low) phase of each cycle.

Syntax

(WIDTH port_tchk value)

port_tchk refers to the port at which the minimum pulse width timing check is applied. If it includes an edge specification, then the data will apply to the width check for the phase of the signal beginning with this edge (see example below). If port_tchk does not include an edge specification, then the data applies to both high and low phases of the signal. value is the minimum pulse width limit and cannot be negative.

Example

(INSTANCE x.b) (TIMINGCHECK (WIDTH (posedge clk) (30)) (WIDTH (negedge clk) (16.5)))

In this example, the first minimum pulse width check is for the phase beginning with the positive clock edge, i.e., the high phase of the clock, and the second minimum pulse width check is for the phase beginning with the negative clock edge, i.e., the low phase. As with all port_tchks, the COND construct can be used to specify conditions associated with the minimum pulse width timing check.

18.8.9 The PERIOD Entry

The PERIOD entry specifies limit values for a minimum period timing check. The minimum period timing check is the minimum allowable time for one complete cycle of the signal.

Syntax

(PERIOD port_tchk value)

port_tchk refers to the port at which the minimum period timing check is applied. If it includes an edge specification, then the data will apply to the period check between consecutive edges of this direction (see example below). If port_tchk does not include an edge specification, then the data applies both to period checks between consecutive rising edges and between consecutive falling edges if they are present in the timing model. value is the minimum period limit and cannot be negative.

Example

(INSTANCE x.b) (TIMINGCHECK (PERIOD (posedge clk) (46.5))

In this example, the data applies to a minimum period check between consecutive rising edges. As with all port_tchks, the COND construct can be used to specify conditions associated with the minimum period timing check.

18.8.10 The NOCHANGE Entry

The NOCHANGE entry specifies limit values for a nochange timing check. The nochange timing check is a signal check relative to the width of a control pulse. A "setup" period is established before the start of the control pulse and a "hold" period after the pulse. The signal checked against the control signal must remain stable during the setup period, the entire width of the pulse and the hold period. A typical use of a nochange timing check is to model the timing of memory devices, when address lines must remain stable during a write pulse with margins both before and after.

Syntax

(NOCHANGE port_tchk port_tchk rvalue rvalue)

The first port_tchk refers to the control port, which is typically a write enable input to a memory or register file device. An edge specification must be included for the control port.

The second port_tchk refers to the port checked against the control port, which is typically an address or select input to a memory or register file device. An edge specification can be included.

The first rvalue is the minimum time that the data/address must be present (stable) before the specified edge of the control signal (setup). The second rvalue is the minimum time that the data/address must remain stable after the opposite edge of the control signal (hold).

Example

(INSTANCE x) (TIMINGCHECK (NOCHANGE (negedge write) addr (4.5) (3.5)))

This example defines a period beginning 4.5 time units before the falling edge of write and ending 3.5 time units after the subsequent rising edge of write. During this time period, the addr signal must not change.

As with all port_tchks, the COND construct can be used to specify conditions associated with the nochange timing check.

18.9 Timing Environment and Constraints

Timing specifications that start with the TIMINGENV keyword associate constraint values with critical paths in the design and provide information about the timing environment in which the circuit will operate. Constructs in this section are used in forward-annotation and not back-annotation.

Syntax

te_spec
::= (TIMINGENV te_def+)
te_def ||=
cns_def // constraint ::= tenv_def // timing environment

cns_def ::= (PATHCONSTRAINT name? port_instance port_instance+ rvalue rvalue) ||= (PERIODCONSTRAINT port_instance value exception?) ||= (SUM constraint_path constraint_path+ rvalue rvalue?) ||= (DIFF constraint_path constraint_path value value?) ||= (SKEWCONSTRAINT port_spec value)

Any number of te_def entries may appear in a te_spec entry. Each te_def will be a PATHCONSTRAINT, PERIODCONSTRAINT, SUM, DIFF or SKEWCONSTRAINT constraint entry, containing constraint values for the design or an ARRIVAL, DEPARTURE, SLACK or WAVEFORM timing environment entry, containing information about the timing environment in which the circuit will operate.

Constraint entries provide information about the timing properties that a design is required to have in order to meet certain design objectives. A tool that is synthesizing some aspect of the design (logic synthesis, layout, etc.) will adapt its strategy to try to ensure that the constraints are met and issue warning messages in the event that they cannot be met.

The following sections describe the SDF constraint constructs.

18.9.1 The PATHCONSTRAINT Entry

The PATHCONSTRAINT entry represents delay constraints for paths. Path constraints are the critical paths in a design identified during timing analysis. Layout tools can use these constraints to direct the physical design. The constraint specifies the maximum allowable delay for a path, which is typically identified by two ports, one at each end of the path. You can also specify intermediate ports to uniquely identify the path.

Syntax

(PATHCONSTRAINT name? port_instance port_instance+rvalue rvalue)
name ::= (NAME QSTRING)

name is optional and allows a symbolic name to be associated with the path. This name should be used by the tool to identify the path to the user when information about the path (problems, failures, etc.) is to be provided. The name is assumed to be more convenient for this purpose than the list of port instances.

The first port_instance is the start of the path. The last port_instance is the end of the path. You can specify intermediate points along the path by using additional port_instances in this entry. The first rvalue is the maximum rise delay between the start and end points of the path. The second rvalue is the maximum fall delay between the start and end points of the path.

Example

```
(INSTANCE x) (TIMINGENV
(PATHCONSTRAINT y.z.i3 y.z.o2 a.b.o1 (25.1) (15.6)) )
```

18.9.2 The PERIODCONSTRAINT Construct

The PERIODCONSTRAINT construct allows a path constraint value to be specified for groups of paths in a synchronous circuit. All paths in the group will be from the common clock input of some flip-flops to the data inputs of the flip-flops that share the common clock. This can be used to derive the frequency at which a circuit must operate as a constraint on how long signals can take after a clock edge to reach the register data inputs.

Syntax

```
( PERIODCONSTRAINT port_instance value exception?)
exception ::= ( EXCEPTION cell_instance+ )
```

port_instance identifies the common clock signal which is the start of all constrained paths. Whereas the start of a PATHCONSTRAINT entry is normally an input port, port_instance here is normally the output port of the device that drives the clock of the flip-flops. Only flip-flops directly connected to this output are in constrained paths. Paths that pass through other buffers before reaching a flip-flop clock are also considered in the group constrained by this entry.

Period Constraint value is the maximum allowable delay for each path in the group. Included in this delay is the clock-to-output delay of the flip-flop driven from port_instance, the setup time of the flip flop that ends the path, and the delay through any combinational logic before arrival at the data input of a flip-flop. Not included is the difference in the timing of the clock of that flip-flop that ends the path from the clock that starts the path. These two times will cause the value supplied in a PERIODCONSTRAINT entry to be different (typically smaller) than the intended clock period at which the circuit will operate. Since only one value can be supplied for all paths in this group, some data may be lost in combining many PATHCONSTRAINT entries into one PERIODCONSTRAINT entry. exception is optional and allows paths to be excluded from the group by the identification of a cell through which they pass. One or more cell instances can be listed after the EXCEPTION keyword. The hierarchical path to these cell instances is relative to the scope or design region identified by the cell entry. Therefore, the PERIODCONSTRAINT entry must appear at a hierarchical level that includes the cell instance that drives the common clock inputs of the flip-flops and any cell instances to be placed in the exception list.

Example

(INSTANCE x) (TIMINGENV
(PERIODCONSTRAINT bufa.y (10) (EXCEPTION (INSTANCE dff3))))

Clearly, any tool that makes use of PERIODCONSTRAINT entries in SDF must be able to traverse the design topology and recognize flip-flops and their clock and data inputs.

18.9.3 The SUM Entry

The SUM entry represents a constraint on the sum of the delay over two or more paths in a design.

Syntax

(SUM constraint_path constraint_path+ rvalue rvalue?)
constraint_path ::= (port_instance port_instance)

Each constraint_path specifies a path to be included in the sum. You must specify at least two paths, but can specify more.In each constraint_path the first port_instance is the beginning of the path and the second port_instance is the end of the path. rvalue is the constraint value. The total (sum) of the individual delays associated with each constraint_path must be less than rvalue. If two rvalues are supplied, the first applies to the rising transition at the end of the path and the second to the falling.

Example

(INSTANCE x) (TIMINGENV (SUM (m.n.o1 y.z.i1) (y.z.o2 a.b.i2) (67.3)))

This example constrains the sum of the delays along the two nets shown as heavy lines in the diagram to be less than 67.3 time units.

18.9.4 The DIFF Constraint

The DIFF entry represents a constraint on the difference in the delay over two paths in a design. Syntax (DIFF constraint_path constraint_path value value?) constraint_path specifies a path between two ports. You must specify exactly two paths. In each constraint_path the first port_instance is the beginning of the path and the second port_instance is the end of the path. value is the constraint value and must be a positive number or zero. The absolute value of the difference of the individual delays in the two circuit paths must be less than value. If two values are supplied, the first applies to the rising transition at the end of the path and the second to the falling.

Example

(INSTANCE x) (TIMINGENV (DIFF (m.n.o1 y.z.i1) (y.z.o2 a.b.i2) (8.3)))

18.9.5 The SKEWCONSTRAINT Entry

The SKEWCONSTRAINT entry represents a constraint on the spread of delays from a common driver to all driven inputs. Only the driving output port can be specified in this construct. All inputs connected to this output are implied end-points for constrained paths. Only paths over interconnect can be constrained as these implied paths cannot pass through any active devices.

Syntax

```
( SKEWCONSTRAINT port_spec value )
```

port_spec refers to the port driving the net. value is the constraint value and must be a positive number or zero (although zero clock skew might be a hard constraint for a layout tool to meet!). The delays from the output specified by port_spec to all inputs that it drives may not differ from each other by more than value. This does not place a constraint on the actual value of the delays, just their "spread".

Example

```
(CELL (CELLTYPE "buf") (INSTANCE top.clockbufs) (TIMINGENV
(SKEWCONSTRAINT (posedge y) (7.5))) )
```

In this example, a buffer cell of cell type buf is used to drive some clock inputs in a circuit. It is buried in the design hierarchy by being instantiated as bufb in a user block called clockbufs, which in turn is part of the block top. In the excerpt from an SDF file, this buffer is identified in a CELL entry and its output is specified in a SKEWCONSTRAINT entry. The effect is to request that the arrival of the positive edge of the clock should not deviate by more than 7.5 between all the inputs driven by the heavily drawn net in the diagram. Neither the inputs nor the net name need to be specified in the SDF file entry. Note that the driven inputs can be anywhere in the design, irrespective of the hierarchical organization.

18.10 Timing Environment – Information Entries

Timing environment entries provide information about the timing environment in which the circuit will operate. This can be used by analysis tools to determine whether or not a design will operate correctly given the back-annotation timing data given elsewhere in the file. It can also be used to compute constraints to be forward-annotated to subsequent stages in the design synthesis process.

Syntax

```
tenv_def ::= ( ARRIVAL port_edge? port_instance rvalue rvalue rvalue rvalue )
      ||= ( DEPARTURE
port_edge? port_instance rvalue rvalue rvalue rvalue ) ||= ( SLACK
port_instance rvalue rvalue rvalue rvalue NUMBER? ) ||= ( WAVEFORM
port_instance NUMBER edge_list )
```

The following sections describe the SDF timing environment constructs.

18.10.1 The ARRIVAL Construct

The ARRIVAL construct defines the time at which a primary input signal is to be applied during the intended circuit operation. Tools use this information to analyze the circuit for timing behavior and to compute constraints for logic synthesis and layout.

Syntax

(ARRIVAL port_edge? port_instance rvalue rvalue rvalue rvalue) bufa clockbufs

port_edge identifies a port and signal edge that form the time reference for the arrival time specification. The port must be an input port. The port_edge is required if the primary input signal is a fan-out from a sequential element, in which case, port_edge is usually referred to an active edge of a clock signal. Otherwise, the port_edge can be omitted. All ARRIVAL constructs that do not have the port_edge refer to the same implicit time reference point. This reference time should be treated as the time 0 of all WAVEFORM constructs. Note that, to fully specify a timing environment, a WAVEFORM statement is required for each clock signal. port_instance specifies the port at which the arrival time is to be defined. It must be an input or bidirectional port that is a primary (external) input of the top-level module.

Four rvalues carry the arrival-time data in this order: earliest rising, latest rising, earliest falling and latest falling arrival times. All values are relative to the time reference, either by a port_edge, or by the implicit reference point. The earliest arrival times must be less than the latest arrival times for the same transition.

Multiple ARRIVAL statements can be defined for the same input to represent signal paths of different reference port_edges.

Example

(INSTANCE top) (TIMINGENV (ARRIVAL (posedge MCLK) D[15:0] (10) (40) (12) (45)))

This example specifies that rising transitions at D[15:0] are to be applied no sooner than 10 and no later than 40 time units after the rising edge of the reference clock MCLK. Falling transitions are to be applied no sooner than 12 and no later than 45 time units after the edge.

18.10.2 The DEPARTURE Construct

The DEPARTURE construct defines the time at which a primary output signal is to occur during the intended circuit operation. Tools use this information to analyze the circuit for timing behavior and to compute constraints for logic synthesis and layout.

Syntax

(DEPARTURE port_edge? port_instance rvalue rvalue rvalue rvalue)

port_edge identifies a port and signal edge that form the time reference for the departure time specification. The port must be an input port. The port_edge is required if the primary output is a fanout from a sequential element, in which case, port_edge is usually referred to an active edge of a clock signal. Otherwise, the port_edge can be omitted. All DEPARTURE constructs that do not have the port_edge refer to the same implicit time reference point. This reference time should be treated as the time 0 of all WAVEFORM constructs. Note that, to fully specify a timing environment, a WAVEFORM statement is required for each clock signal. port_instance specifies the port at which the departure time is to be defined. It must be an output or bidirectional port that is a primary (external) output of the top-level module. Four rvalues carry the departure-time data in this order: earliest rising, latest rising, earliest falling and latest falling departure times. All values are relative to the time reference, either by a port_edge, or by the implicit reference point. The earliest departure times must be less than the latest departure times for the same transition. Multiple DEPARTURE statements can be defined for the same output to represent signal paths of different reference port_edges.

Example

```
(INSTANCE top) (TIMINGENV (DEPARTURE (posedge SCLK) A[15:0] (8) (20) (12) (34)))
```

The example specifies that rising transitions at primary output A[15:0] are to occur no sooner than 8 and no later than 20 time units after the rising edge of the reference clock SCLK. Falling transitions are to occur no sooner than 12 and no later than 34 time units after the edge.

18.10.3 The SLACK Construct

The SLACK construct is used to specify the available slack or margin in a delay path. This is a comparison of the calculated delay over a path to the delay constraints imposed upon that path. Positive slack indicates that the constraints are met with room to spare. Negative slack indicates a failure to construct the circuit according to the constraints. A layout or logic synthesis tool can use slack information to make trade-offs in cell placement and routing or re-synthesis of parts of the circuit. The objective should be to eliminate negative slack and achieve an even distribution of positive slack.

Syntax

```
( SLACK port_instance rvalue rvalue rvalue rvalue NUMBER? )
```

port_instance specifies the input port at which slack/margin information is given in this entry. Paths terminating at this port have at least the indicated slack/margin. It is not possible in this construct to specify individual paths. The values given must be the minimum of all paths that converge to the:

```
(CELL (CELLTYPE "cpu")
(INSTANCE top) (TIMINGENV (WAVEFORM clka 15 (posedge 0 2) (negedge 5 7)) ) )
```

0 5 10 15
Period = 15

This example shows the specification of a waveform of period 15 to be applied to port top.clka. Within each period, a rising edge occurs at somewhere between 0 and 2 and a falling edge somewhere between 5 and 7. Tools unable to deal with uncertainty in waveforms would place the rising edge and 1 and the falling edge at 6 and issue a warning.

Example

```
(CELL (CELLTYPE "cpu") (INSTANCE top) (TIMINGENV (WAVEFORM clkb 25 (negedge
0) (posedge 5) (negedge 10) (posedge 15) ) ) )
```

This example shows the specification of a waveform of period 25 to be applied to port top.clkb. Within each period, a falling edge occurs at 0, a rising edge at 5, a falling edge at 10 and a rising edge at 15.

Example

```
(CELL (CELLTYPE "cpu") (INSTANCE top) (TIMINGENV (WAVEFORM clkb 50 (negedge
-10) (posedge 20) ) ) )
```

This example shows that negative numbers can be used in defining a waveform.

0 5 10 15 20 25 30
Period = 25
-20 -10 0 10 20 30 40
Period = 50

The SLACK construct is used to specify the available slack or margin in a delay path. This is a comparison of the calculated delay over a path to the delay constraints imposed upon that path. Positive slack indicates that the constraints are met with room to spare. Negative slack indicates a failure to construct the circuit according to the constraints. A layout or logic synthesis tool can use slack information to make trade-offs in cell placement and routing or re-synthesis of parts of the circuit. The objective should be to eliminate negative slack and achieve an even distribution of positive slack.

Syntax

```
( SLACK port_instance rvalue rvalue rvalue rvalue NUMBER? )
```

port_instance specifies the input port at which slack/margin information is given in this entry. Paths terminating at this port have at least the indicated slack/margin. It is not possible in this construct to specify individual paths. The values given must be the minimum of all paths that converge to the specified port_instance. However, the slack/margin may be given at various places on the same path.

Four rvalues carry the slack/margin data. In order, they are the rising setup slack, the falling setup slack, the rising hold slack and the falling hold slack. "Rising" and "falling" indicate the direction of transitions at the specified port_instance to which data applies. The setup slack is the additional delay that could be tolerated in all paths ending at this port without causing design constraints to be violated. Similarly, the hold slack is the reduction of the delay that could be tolerated in all these paths. If rtriples are used in these rvalues, then each number belongs to the data set for that position in the triple. Since the prevailing use of these data sets is to carry data for minimum, typical and maximum delays, setup slack rtriples will have the unusual property of decreasing in value from left to right.

NUMBER is optional and, if present, represents the clock period on which the slack/margin values are based. The clock period refers to the one specified by a WAVEFORM construct.

Example

```
(CELL (CELLTYPE "cpu") (INSTANCE macro.AOI6) (TIMINGENV (SLACK B (3) (3) (7)
(7)) ) )
```

In this example, the delay of any or all data paths leading to port macro.AOI6.B could be increased by 3 time units without violating a setup requirement on a constrained device down the path traversed by this port. This SLACK entry indicates that the signal arrives at port macro.AOI6.B in time to meet the setup time requirement of a flip-flop down the path with 3 time units to spare. Thus, the signals could be delayed up the data path by an additional 3 time units with no ill consequences. The example also shows that the delay of any or all datapaths leading to port macro.AOI6.B could be decreased by 7 time units without violating a hold requirement on a constrained device down the path.

Multiple SLACK entries are allowable for the same port_instance and are distinct if NUMBER is different.

18.10.4 The WAVEFORM Construct

The WAVEFORM construct allows the specification of a periodic waveform that will be applied to a circuit during its intended operation. Typically, this will be used to define a clock signal. Tools can use this information in analyzing the circuit for timing behavior and to compute constraints for logic synthesis and layout.

Syntax

```
( WAVEFORM port_instance NUMBER edge_list )
edge_list ::= pos_pair+ ||= neg_pair+
pos_pair ::= ( posedge RNUMBER RNUMBER?) ( negedge RNUMBER RNUMBER?)
neg_pair ::= ( negedge RNUMBER RNUMBER?) ( posedge RNUMBER RNUMBER? )
```

port_instance identifies the port in the circuit at which the waveform will appear. It must be an input or bidirectional port. If the port is not a primary input of the circuit, i.e., if it is driven by the output of some other circuit element in the scope of

the analysis, then the signal driven in the circuit should be ignored and the specified waveform should replace it in the analysis. The hierarchical path to this port is relative to the scope or design region identified by the cell entry.

NUMBER specifies the period of the waveform. The waveform described repeats indefinitely at this interval.

edge_list describes a single period of the waveform. It consists of a list of edge pairs, which can be either a posedge entry followed by a negedge entry or a negedge entry followed by a posedge entry. Thus, the total number of edges in the list will be even and edges will alternate between posedge and negedge. In addition to the direction of the transition, each edge gives the time at which the transition takes place relative to the start of each period. Offsets must increase monotonically throughout the edge_list and must not exceed the period. If one RNUMBER is supplied, then this precisely defines the transition offset. If two RNUMBERs are supplied, then they define an uncertainty region in which the transition will take place. The first RNUMBER gives the beginning of the uncertainty region and the second RNUMBER gives its end. Tools using this construct with two RNUMBERs should assume that a single transition of the specified direction occurs somewhere in the uncertainty region, but should make no assumptions about exactly where. Tools unable to model this edge uncertainty should issue a warning message and use the mean of the two RNUMBERs to locate the transition.

Example

(CELL (CELLTYPE "cpu")(INSTANCE top) (TIMINGENV (WAVEFORM clka 15 (posedge 0 2) (negedge57))))
0 5 10 15
Period=15

This example shows the specification of a waveform of period 15 to be applied to port top.clka. Within each period, a rising edge occurs at somewhere between 0 and 2 and a falling edge somewhere between 5 and 7. Tools unable to deal with uncertainty in waveforms would place the rising edge and 1 and the falling edge at 6 and issue a warning.

Example

(CELL (CELLTYPE "cpu") (INSTANCE top) (TIMINGENV (WAVEFORM clkb 25 (negedge 0) (posedge 5) (negedge 10) (posedge 15))))

This example shows the specification of a waveform of period 25 to be applied to port top.clkb. Within each period, a falling edge occurs at 0, a rising edge at 5, a falling edge at 10 and a rising edge at 15.

Example

(CELL (CELLTYPE "cpu") (INSTANCE top) (TIMINGENV (WAVEFORM clkb 50 (negedge -10) (posedge 20))))
.0 5 10 15 20 25 30

Period = 25
-20 -10 0 10 20 30 40
Period = 50

18.11 SDF File Examples

SDF FILE EXAMPLE 1

```
(DELAYFILE
(SDFVERSION "1.0") (DESIGN "system") (DATE "Saturday September 30
08:30:33 PST 1990") (VENDOR "Yosemite Semiconductor") (PROGRAM "delay_calc")
(VERSION "1.5") (DIVIDER /) (VOLTAGE 5.5:5.0:4.5) (PROCESS "worst")
(TEMPERATURE 55:85:125) (TIMESCALE 1ns) (CELL (CELLTYPE "system")
(INSTANCE) (DELAY (ABSOLUTE (INTERCONNECT P1/z B1/C1/i (.145::.145)
(.125::.125)) (INTERCONNECT P1/z B1/C2/i2 (.135::.135) (.130::.130))
(INTERCONNECT B1/C1/z B1/C2/il (.095::.095) (.095::.095))
(INTERCONNECT
B1/C2/z B2/Cl/i (.145::.145) (.125::.125)) (INTERCONNECT B2/Cl/z
B2/C2/il (.075::.075) (.075::.075)) (INTERCONNECT B2/C2/z P2/i
 (.055::.055) (.075::.075)) (INTERCONNECT B2/C2/z D1/i (.255::.255)
(.275::.275)) (INTERCONNECT D1/z B2/C2/i2 (.155::.155) (.175::.175))
(INTERCONNECT D1/z P3/i (.155::.155) (.130::.130)) ) ) ) (CELL
(CELLTYPE "INV") (INSTANCE B1/C1) (DELAY (ABSOLUTE (IOPATH i z (.345::.345)
(.325::.325) ) ) ) ) (CELL (CELLTYPE "OR2") (INSTANCE B1/C2) (DELAY
(ABSOLUTE (IOPATH i1 z (.300::.300) (.325::.325) ) (IOPATH i2 z
(.300::.300) (.325::.325) ) ) ) ) (CELL (CELLTYPE "INV") (INSTANCE
B2/C1) (DELAY (ABSOLUTE (IOPATH i z (.345::.345) (.325::.325) ) )
) ) (CELL (CELLTYPE "AND2") (INSTANCE B2/C2) (DELAY (ABSOLUTE (IOPATH
i1 z (.300::.300) (.325::.325) ) (IOPATH i2 z (.300::.300) (.325::.325)
) ) ) ) (CELL (CELLTYPE "INV") (INSTANCE D1) (DELAY (ABSOLUTE (IOPATH
i z (.380::.380) (.380::.380) ) ) ) ) )
```

SDF FILE EXAMPLE 2

This example shows how you can use the COND construct with the IOPATH and TIMINGCHECK constructs.

```
(DELAYFILE (SDFVERSION "2.0") (DESIGN "top") (DATE "Feb 21, 1992 11:30:10")
(VENDOR "Cool New Tools") (PROGRAM "Delay Obfuscator") (VERSION "v1.0")
(DIVIDER .) (VOLTAGE :5:) (PROCESS "typical") (TEMPERATURE :25:) (TIMESCALE
1ns)(CELL (CELLTYPE "CDS_GEN_FD_P_SD_RB_SB_NO") (INSTANCE top.ff1)
(DELAY (ABSOLUTE (COND (TE == 0 && RB == 1 && SB == 1) (IOPATH (posedge
CP) Q (2:2:2) (3:3:3) ) ) ) (ABSOLUTE (COND (TE == 0 && RB == 1 &&
SB == 1) (IOPATH (posedge CP) QN (4:4:4) (5:5:5) ) ) ) (ABSOLUTE (COND
(TE == 1 && RB == 1 && SB == 1) (IOPATH (posedge CP) Q (6:6:6) (7:7:7)
) ) ) (ABSOLUTE (COND (TE == 1 && RB == 1 && SB == 1) (IOPATH (posedge
CP) QN (8:8:8) (9:9:9) ) ) ) (ABSOLUTE (IOPATH (negedge RB) Q (1:1:1)
(1:1:1) ) ) (ABSOLUTE (IOPATH (negedge RB) QN (1:1:1) (1:1:1) ) )
(ABSOLUTE (IOPATH (negedge SB) Q (1:1:1) (1:1:1) ) ) (ABSOLUTE (IOPATH
(negedge SB) QN (1:1:1) (1:1:1) ) ) ) (DELAY (ABSOLUTE (PORT D (0:0:0)
```

```
(0:0:0) (5:5:5) ) )
(ABSOLUTE
(PORT CP (0:0:0) (0:0:0) (0:0:0) ) ) (ABSOLUTE (PORT RB (0:0:0) (0:0:0)
(0:0:0) ) ) (ABSOLUTE (PORT SB (0:0:0) (0:0:0) (0:0:0) ) ) (ABSOLUTE
(PORT TI (0:0:0) (0:0:0) (0:0:0) ) ) (ABSOLUTE (PORT TE (0:0:0) (0:0:0)
(0:0:0) ) ) ) (TIMINGCHECK (SETUP D (COND D_ENABLE (posedge CP)) (1:1:1)
) (HOLD D (COND D_ENABLE (posedge CP)) (1:1:1) ) (SETUPHOLD TI (COND
TI_ENABLE (posedge CP)) (1:1:1) (1:1:1)) (WIDTH (COND ENABLE (posedge
CP)) (1:1:1) ) (WIDTH (COND ENABLE (negedge CP)) (1:1:1) ) (WIDTH
(negedge SB) (1:1:1) ) (WIDTH (negedge RB) (1:1:1) ) (RECOVERY (posedge
RB) (COND SB (negedge CP)) (1:1:1) ) (RECOVERY (posedge SB) (COND
RB (negedge CP)) (1:1:1) ) ) ) )
```

SDF FILE EXAMPLE 3

This example shows how State Dependent Path Delays can be annotated using COND and IOPATH constructs.

```
(DELAYFILE (SDFVERSION
"2.0") (DESIGN "top") (DATE "Nov 25, 1991 17:25:18") (VENDOR "Slick
Trick Systems") (PROGRAM "Viability Tester") (VERSION "v3.0") (DIVIDER
.) (VOLTAGE :5:) (PROCESS "typical") (TEMPERATURE :25:) (TIMESCALE
1ns) (CELL (CELLTYPE "XOR2") (INSTANCE top.xl) (DELAY (INCREMENT (COND
i1 (IOPATH i2 o1 (2:2:2) (2:2:2) ) ) ) (INCREMENT (COND i2 (IOPATH
i1 o1 (2:2:2) (2:2:2) ) ) ) (INCREMENT (COND ~i1 (IOPATH i2 o1 (3:3:3)
(3:3:3) ) ) ) (INCREMENT (COND ~i2 (IOPATH i1 o1 (3:3:3) (3:3:3) )
) ) ) ) )
```

SDF FILE EXAMPLE 4

This example shows how to forward annotate timing constraints. The key to specifying SDF constraints is to identify INSTANCE-PINS of library cells. In the example shown below I2 is an instance and H01 is a PIN (port) on that instance.

```
(DELAYFILE (SDFVERSION "3.0")
(DESIGN "testchip") (DATE "Dec 17, 1991 14:49:48") (VENDOR "Big Chips
Inc.") (PROGRAM "Chip Analyzer") (VERSION "1.3b") (DIVIDER .) (VOLTAGE
:3.8:) (PROCESS "worst") (TEMPERATURE : 37:) (TIMESCALE 10ps) (CELL
(CELLTYPE "XOR") (INSTANCE ) (TIMINGENV (PATHCONSTRAINT I2.H01 I1.N01
(989:1269:1269) (989:1269:1269) ) (PATHCONSTRAINT I2.H01 I3.N01 (904:1087:1087)
(904:1087:1087) ) ) ) )
```

18.12 Delay Model

18.12.1 Introduction

The delay model provides a guideline for using SDF in ASIC application tools. All constructs in SDF should be directly applicable to the delay model. ASIC timing is divided into forward annotation and back annotation. Although SDF supports both

timing concepts, this section concentrates on ASIC timing back-annotation model. A future release of SDF will provide an abstract model for forward annotation.

The following section defines the delay model and provides rules that should be adhered to to ensure proper interpretation and usage of SDF constructs.

18.12.2 The List of Delay Models

The delay model consists of the following kinds of delays:

1. Interconnect delay (INT), represented by the INTERCONNECT delay construct in SDF.
2. Path delay (PD), represented by IOPATH delay construct in SDF.
3. State-dependent path delay (SDPD), represented by COND keyword in SDF.
4. Port delay (IPD), represented by PORT delay construct in SDF.
5. Device delay (DEV), represented by DEVICE construct in SDF. Note when specified with a cell output port, this timing object is a degenerate path delay; when specified with a primitive instance, this timing object is its intrinsic delay.
6. Path pulse (PP), represented by PATHPULSE construct in SDF.
7. Timing checks (TC), represented with several keywords in SDF depending on the type of the timing checks.

18.12.3 Rules for the Delay Model

Summary of the Rules for the Delay Models in SDF

1. Path delay is described between any input (or bidirectional) port to any output (or bidirectional) port in the same cell.
2. Multiple path delays can be defined for any output (or bidirectional) port.
3. Multiple path delays can be defined between any pair of ports only by using state dependent delays.
4. Path delay can have up to twelve transition states with twelve different delay values.
5. Negative timing values for absolute input-output path, port, net, device and interconnect delays may default to zero in certain application tools.
6. Interconnect delay is described between any output (bidirectional) port of a cell to any input (bidirectional) port of any cell.
7. Multiple interconnect delays from different sources can be described for any input (bidirectional) port, destination port.
8. Depending on the type of the timing check, it can be applied to a single or a pair of ports.

9. Timing checks are allowed from an output port to another output port.
10. Timing checks are applied after the interconnect delays are applied.
11. Negative timing check limit values are allowed only for the SDF SETUPHOLD, RECREM and NOCHANGE constructs. Some application tools may use the negative values while others may compile them as zero values.
12. INTERCONNECT delay between a source and a destination signal cannot be used if PORT delay is specified for the same destination signal.
13. Similarly, PORT delay for a destination signal cannot be used if an INTERCONNECT delay is specified between a source and the same destination signal.
14. IOPATH delay cannot be used if a DEVICE delay is specified for the same output
15. Port within the same cell.
16. Similarly, DEVICE delay cannot be used if an IOPATH delay is specified between an input port and the same output port within the same cell.
17. All timing objects using the internal nodes may be ignored by application tools that have no concept of the internal nodes.
18. For the same timing object, delay annotation is executed in the sequential order as encountered in a single SDF file.

18.13 DCL – New Emerging Standard

DCL (Delay Calculation Language) is a new standard in the area of timing descriptions and is useful for deep submicron designs where interconnect delay descriptions are involved. The information on this is available from CFI. Originally developed at IBM where the deep submicron technology was implemented first, this is now accepted as CFI standard.

For interconnect effects of distributed RLC characteristics need to be modeled to characterize the transient function of each driven point on a net. Characterization of delay must consider state and switching dependencies between the controlling input to a gate and the other inputs. Signal slew rates, and degradation, or attenuation, as a result of crosstalk have become factors to be coped with in the delay characterization process.

DCL provides a file format and a programming interface that replaces the access routines typically used with SDF. DCL may be used for describing synthesis libraries and attempts have been made to make a single format description for synthesis as well simulation libraries for the ASIC. Currently, synthesis tools typically use a proprietary format and the simulation tools are using SDF files. Timing constraints are provided by SDF for synthesis and timing analysis tools. The working group on extensions of SDF, plans to add some of the new features into SDF 4.0 and the 3.0

specification already contains several features for describing interconnect delays as given earlier in this chapter.

18.14 OMI Standard

The OMI is a new standard proposed for interoperability of models across different languages and environments. This relies on a subset of PLI that is applicable to linking and running models like the one provided in veriuser.h files [[tf_] routines However, this requires an additional software component called the model manager which must work together with all the tools—thus making it a framework like or centralized interface as opposed to one-to-one like in the PLI. Similar interfaces have been proposed in the past [see the CFI web site for intertool communication standard accepted in early nineties—but have not really taken root. However, the IP and core methodology may rely on this new interface and attempts are being made in this direction.

19 VERILOG-A AND VERILOG-MS

19.1 Analog Module

19.1.1 Introduction

SPICE is a common method of modeling analog descriptions. Verilog-A and Verilog-MS allow Verilog HDL to be used for describing electrical circuit behaviorally like the digital system. The digital behavior is described behaviorally under the **initial** and **always** blocks. A new **analog** block is added to Verilog HDL for circuit behavior. Differential equations following Kirchoff's laws are described in the analog block using the symbol <+ for '=' commonly used in Mathematics. Analog nets are called nodes and have values that are n-tuples, typically voltage, current and few other characteristics. In this chapter, we describe the analog modules using the terminology and syntax that reuses those defined for digital Verilog whenever it is the same and defines the new syntax and semantic items. As in the rest of the book, each feature is explained in terms of semantic introduction, example and syntax.

19.1.2 Examples

An example of a resistor is:

```
module resistor(rl, r2);
    inout rl, r2; // Analog ports are almost always inout like in the bidirectionals
    electrical rl,r2;

    parameter real r=l;
    parameter real tc=1.5m;
```

```
        real reff;
        initial
        begin
            reff = r *( 1 + tc * $temp());
        end

        analog
        begin
            I(rl, r2) <+ V(rl, r2))/reff;
        end
    endmodule
```

Example 19-1. Analog resistor described in Verilog's analog extensions.

19.1.3 Syntax

analog_module ::=
module_keyword *module_*identifier [list_of_ports]
{analog_*module*_item}
endmodule

19.2 Analog Data Declarations

19.2.1 Introduction

The nets or nodes in analog designs have values that are real numbers and are typically measures such as voltage, current, temperature and are evaluated as differentials with respect to time. Such nets are declared within the analog modules as **electrical** types.

19.2.2 Examples

```
electrical rl, r2;
electrical [31:0] il, i2;
```

Example 19-2. Electrical type declarations in analog or mixed signal modules.

19.2.3 Syntax

analog_data_declaration::= **electrical** list_of_*electrical_* identifiers

19.3 Analog Behavioral Descriptions

19.3.1 Introduction

The analog behaviors are described in a new block that begins with keyword **analog.** One can express differential equations that will be solved using an iterative method of solving differential equations. The simulation algorithm consists of selecting a time-step for solving the set of equations generated from the parallel analog block. And then iteratively changing the time-step until solution is arrived. This is different from SPICE methodology as we do not have a predefined set of SPICE library elements. The level of abstraction is raised from structural to behavioral and thereby providing capability of describing analog systems or cells that could be larger in size and are more complex in design and could be synthesized using analog synthesis tools.

19.3.2 Examples

A resistor is described in Example 19-1. A transformer is described below in Example 19-3.

```
module transformer(inl, in2, outl, out2);
analog
begin
    V(node, outm) <+ leakL * dot(I(node, outl);
    V(out2, node) <+ ratio * V(in2, inl);
end
endmodule
```

Example 19-3. Example of analog block.

19.3.3 Syntax

```
analog_statement ::=
        analog analog_statement
            |initial_statement

        analog_statement ::=
            analog_assignment
            |sequential_assignment
            |sequential_block

        analog_assignment ::=
            electrical_identifier <+ analog_ expression
```

analog_expression is a combination of expressions in Verilog HDL and functions described before.

19.4 Expressions in Analog Assignments

As the analog equations are different from digital assignments, additional operations in expressions are provided in Verilog-A to handle functions such as integrals and differentials, sine, cosine amongst others. The key functions in this area are:

Trignometric Functions

sine cosine. tan asin acos atan sinh cosh tanh asinh acosh atanh

Calculus Functions

dot (Differential) and integ(Integral)

Simulation Environment Functions

$time $temp $vt[Thermal Voltage] $analysis (string)

Waveform Filter Functions

$transition $slew $tdelay $zdelay

Simulator Time Step Control Functions

$threshold $last_crossing $bound_step $break_point

Simulator Information Functions

$strobe $warning $error $fatal

19.5 Mixed Signal Designs in Verilog

The standard for mixed-signal designs based on Verilog HDL and Verilog-A analog description language is described here. It is expected that an interface module will be defined that instantiates an analog module and a digital module. This will be transparent to the language and one really has to develop a model underneath for the connections of electrical wires to digital net types.

```
module mixed-signal;
    electrical a1;
    wire w2, w3, w4;

    manalog mai(al, w2, w3);
    mdigital mdi(al, w4);

endmodule

module manalog(a1, a2, a3);
        inout a1, a2, a3;
    electrical a1, a2, a3;

        // Body of the module
endmodule

module mdigital(w1, w2, w3);
    output w3;
```

```
    input w1, w2;
    // Body of the module
endmodule
```

Example 19-4. Mixed signal design with Verilog MS.

In this example, analog to digital converters will be interfaced at the first port of instance mdi and digital to analog converters will be placed at the second and third ports of mai. This will be done implicitly by the compiler and the simulator. Explicit conversions of signals from analog to digital and vice versa will be supported in this language definition.

20 SIMULATION SPEEDUP TECHNIQUES

20.1 Cycle-Based Simulation

The simulation of a system can be a time-consuming part of a project. Optimizations to simulation algorithms can be performed on certain subsets of the HDL. One such technique commonly used is known as cycle-based simulation. As seen in the synthesis subset, the models written must follow a subset that describes a synchronous system using the model of a Sagdeo machine. [Figure 12-2]. In such cases, the activity in the simulator is only present on clock-edges. There are no other types waits involved or synchronizations or delays. Thus, one can lump all clock-based activity in one block of execution and perform an accelerated simulation by removing the events altogether. Thus, a network data structure similar to one in Figure 4-1 is created during compilation and one only has to perform evaluations based on this data structure. On each clock edge, stimulus changes are captured in a predetermined fashion and then applied to the circuit under simulation. On each net or reg change arising from this stimulus change, propagation takes place according to the compiled data structure as per connectivity of different nodes and evaluation blocks without creation of any events.

In general, event creation and processing takes up 90% of the simulation cpu time. With this method, when the input description is at RTL level and follows the basic synthesis subset model, one can achieve simulation speedup of an order of magnitude. However, the stimulus is still being generated at the higher level and the simulator must have the capacity to handle both kind so simulations together and interface the two algorithms correctly without losing the speedup achieved in the RTL sections.

The subset supported by cycle based simulators closely matches the subset of synthesis. Refer to Chapter 13 and Appendix C for details on modeling for performance using cycle-based simulations. Again, individual tools may have certain

limitations or additional features and refer to data-sheet of your simulator for any such features.

20.2 2-State Versus 4-State Simulations

Verilog regs and nets are 4-stated variables. However, in a fully debugged simulation, one does not expect x's or z's and if they appear then they are don't care conditions. For such a simulation, one can run the simulator in 2-state whereby only 0 and 1 values are needed for all evaluations of the system. This can result in some speedup when performing large-scale regression testing done extensively as the design is being completed and taped out. This method can be effectively applied for such cases. A good Verilog simulator should provide a compile time option to perform such type of optimized simulation.

20.3 Compiled, Native Code, and Interpretive Simulations

An interpretive simulator compiles the Verilog code into internal data structure that may be then used for step-by step simulation which can be debugged directly at the source level. This kind of simulation is ideally suited during module by module development of a design. As the design gets larger and the simulation initialization and test-sequence takes longer CPU times, one can optimize the simulation by compiling down the debugged modules into either binary code via C code translation and compilation or directly generating native code. The step of producing C code and then compiling this takes longer compilation times at the benefit of faster run-time. Direct generation of native code typically helps in speeded compilation and may even have some small advantage in simulation. However, C compilers have several optimizing cod generation algorithms which may benefit the run-time and the native code generator must match the quality of optimized code—especially when pipelining and other such processor dependent code-optimizations are involved.

20.4 Parallel Processors and Multi-Threaded Simulators

The parallel processing machines can speed up certain applications upto the number of processors and at times, more if the caches are local to each processor and the simulations get the advantages of this configuration. For multi-processing applications, a simulator must be rewritten with creation of threads such that all the processors get the computations in simulation done in parallel(generally with a shared memory). Some of the best results can be obtained again for code that has large number of processing done at the same time. The RTL code that is based on a single event in a cycle based simulation may not be vectorized so well although benefits in evaluating blocks in parallel may be seen. Events tend to be more naturally threaded and simulations with large number of events at the same time can run with speedups upto n times. Large gate-level or switch-level simulations can also run well in parallel in an algorithm whereby each thread creates its own events in a thread-based event queue. While running for the R1000 processor simulations on Silicon Graphics Challenge machines upto 16 processors, 10-fold speedups were obtained. Up to eight or less processors the speedup was more than the number of

processors since the local caches were now large enough to hold all the data and consequently the speed was multiplied more than just the number of processors. In general, as describes in the next section, management of cache and memory could be as important as managing the parallelism amongst threads in simulation runs.

20.5 Usage of Caches and Other Memory to Achieve Speedup

The typical simulations of a large design tends to run on certain parts of the design for certain item and then move onto other parts when the boundaries across higher level blocks get triggered. In such cases, the caches can hold the data on which the simulations run for longer time providing simulation speedup. Thus, managing the application of stimulus and locality of reference are good techniques of managing your simulation runs. The random pattern application really may test the circuit well but will not have the locality of reference that a well-developed systemic test-suite could provide.

Virtual memories typically tend to be too slow for simulations as the activity rate I every high, that is, this is a CPU and memory-access intensive application. Thus, providing real memory greater than the simulator would need for running the system is a must for simulation runs to complete in reasonable times. The needs of memory vary for different tools and should be looked at while evaluating the tool that will be used for a design project.

20.6 Distributed Simulations Over a Network of Workstations

Large designs and larger regression test-suites can be run on a distributed system by partitioning the tests in an easier fashion and partitioning the design in cases the design grows beyond the memory capacity of individual workstations. Communicating over a distributed network for events could be slow and thus partitioning of design must be done such that very few pieces of data are exchanged amongst a distributed system. All the tests must go through similar initialization of the circuit and then apply individual tests when test-data is partitioned. Tools are available to manage the distributed runs of a system which can schedule the runs, do the initial distribution of data, help in synchronization and in finally gathering the result together.

20.7 C Code Versus HDL Code

For hardware-software codesign, the full-detailed synthesizable HDL models of the system could be slow in terms of number of cycles needed for software to be run on the hardware to be built ahead of the time to speedup the final system delivery. In such cases, C code is widely used to model the high-level processor, cache and other system models. However, the modeling in Verilog using higher levels of abstractions is a viable and preferred alternative. As seen in the cache design examples of chapter 11, the behavioral models run order of magnitude faster than the RTL models as the details of synchronization are absent in these models. This level of abstraction can be further raised and cycle-based simulations can still be used to get performance comparable to C models and accuracy and model-development far better. The core or

IP developers face this question more than others and must address this in this direction. Higher level features are also being added to Verilog HDL in the 1364-98 proposed in IVC 97.

20.8 File Management in Simulation

Reading large set of patterns and writing large set of data can be quite time-consuming especially in a networked file system. Files must be locally present on a workstation in a distributed system for the speed issues. On a single workstation, reading and writing must be minimized especially within an event loop. This can be achieved by reading large chunks of data in a cached manner and similarly for writing, local caching must be done. The output dumpfiles in the format that Verilog provides could be long and several compression techniques are available either to be implemented as user-defined tasks or via tools such as Veritools waveform processor.

A FORMAL SYNTAX DEFINITION FOR VERILOG HDL

The following is reproduced from IEEE std 1364-1995 Verilog Hardware Description Language Reference Manual, Copyright 1995 by the Institute of Electrical and Electronics Engineers, Inc. The IEEE disclaims any responsibility or liability resulting from the placement and use of the publication. This information is reprinted with the permission of the IEEE.

The following table summarizes the format of the formal syntax descriptions.

Definition of Items in the Formal Syntax Specifications

Item	**Meaning**
White Space	May be used to separate lexical tokens.
name ::=	Starts off the definition of a syntax construct item. Sometimes name contains embedded underscores "_". Also, the "::=" may be found on the next line.
|	Introduces an alternative syntax definition, unless it appears bold. (See next item.)
name	Bold text is used to denote reserved keywords, operators, and punctuation marks required in the syntax.
[item]	Is an optional item that may appear zero or one time.
{item}	Is an optional item that may appear zero, one or more times. IF the braces are in bold, then they are part of the syntax.
Name1_name2	This 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.

This is fixed for some bugs from the orginal specification.

A.1 Source Text

source_text
    ::= {description}

description
    ::= module_declaration
    | UDP_declaration

module_declaration
    ::= module_keyword *module*_identifier [list_of_ports]
        {*module*_item}
        endmodule

module_keyword
    ::= module | macromodule

list_of_ports
    ::=(port {,port })

port
    ::= [port_expression]
    | .*port*_identifier ([port_expression])

port_expression
    ::= port_reference
    | { port_reference ,port_reference }

port_reference
    ::=*port*_identifier
    | *port*_identifier[constant_expression]
    | *port*_identifier [msb_constant_expression :lsb_constant_expression]

module_item
    ::= module_item_declaration
    | gate_instantiation
    | udp_instantiation
    | module_instantiation
    | parameter_override
    | continuous_assign
    | specify_block
    | initial_statement
    | always_statement

module_item_declaration
    ::= parameter_declaration
    | input_declaration
    | output_declaration
    | inout_declaration
    | net_declaration

| reg_declaration
| time_declaration
| realtime_declaration
| integer_declaration
| real_declaration
| event_declaration
| task_declaration
| function_declaration

A.2 Declarations

parameter_declaration
::= **parameter** list_of_param_assignments;

list_of_param_assignments
::=param_assignment {,param_assignment}

param_assignment
::=identifier = constant_expression

input_declaration
::= **input** [range] list_of_port_identifiers ;
output_declaration
::= **output** [range] list_of_ port_ identifiers;
inout_declaration
::= **inout** [range] list_of_ port_ identifiers;
net_declaration ::= net_type [expandrange] [delay] list_of_net_ identifiers;
| **trireg** [charge_strength] [expandrange] [delay]
list_of_ net_ identifiers; | NET_TYPE [drive_strength] [expandrange] [delay]
list_of_net_decl_assignments;
list_of_net_ identifiers ::= net_ identifier , { net_ identifier }
net_type ::=: **wire | tri | tri1 | supply0 | wand | triand | tri0 | supply1 | wor | trior | trireg**

expandrange
::= range
| **scalared** range
| **vectored** range

reg_declaration
::= **reg** [range] list_of_register_ identifiers;
list_of_ register_ identifiers ::= register _ identifier, {register _ identifier}
time_declaration
::= **time** list_of_register_ identifiers;

integer_declaration
::= **integer** list_of_register_ identifiers;

real_declaration
::= **real** list_of_ real_identifiers;

list_of_real_identifiers
::= real_ identifier , { real_ identifier }

event_declaration
::= **event** *event*_identifier{ *event*_identifier} ;

task _declaration
::= **task** *task*_identifier;
{task_item_declaration}
statement_or_null
endtask

task_item_declaration
::= block_item_declaration
| input_declaration
| output_declaration
| inout_declaration

function_declaration
::= **function** [range_or_type] *function*_identifier;
function_item_declaration
statement
endfunction

function_item_declaration
::= block_item_declaration
| input_declaration

range_or_type
::= range
| integer
| *function*_identifier
::= identifier

block_item_declaration
::= parameter_declaration
| reg_declaration
| time_declaration
| integer_declaration
| real_declaration
| realtime_declaration
| event_ declaration

continuous_assign
::= **assign** [drive_strength] [delay] list_of_assignments ;
| net_type [drive_strength] [expandrange] [delay] list_of_assignments ;

parameter_override
::= **defparam** list_of_param_assignments ;

list_of_register_variables
::=*register*_identifier { ,*register*_identifier }

register_variable
::=*register*_identifier
| *memory*_identifier [constant_expression : constant_expression] *register*_identifier
::=memory–identifier
::= *event*_identifier

charge_strength
::=**(small)**
| **(medium)**
| **(large)**

drive_strength
::= (strength0 , strength1)
| (strength1, strength0)

strength0 is one of the following keywords:
supply0 strong0 pull0 weak0 highz0

strength1 is one of the following keywords:
supply1 strong1 pull1 weak1 highz1

range
::= [msb_constant_expression : lsb_constant_expression]

list_of_net_decl_assignments
::= net_decl_assignment{ ,net_decl_assignment}

net_decl_assignment
::=*net*_identifier= expression

A.3 Primitive Instances

gate_instantiation
::=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};
| n_enable_gatetype [drive_strength] [delay3] enable_gate_instance
{ ,enable_gate_instance};
| mos_switchtype [delay3] mos_switch_instance{ , mos_switch _instance};
| pass_switchtype [delay3] pass_switch_instance{ , pass_switch_instance} ;
| pass_en_switchtype [delay3] pass_en_switch_instance{, pass_en_switch _instance} ;
| cmos_switchtype [delay3] cmos_switch_instance{, cmos_switch _instance};
| **pullup** [pullup_strength] pull-gate_instance{ , pull_gate_instance} ;
| **pulldown** [pulldown_strength] pull-gate_instance{ , pull_gate _instance} ;

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, input_terminal});
enable_gate_instance ::= [name_of_ gate_instance] (output_terminal, input_terminal{, input_terminal,enable_terminal});
mos_switch_instance ::= [name_of_ gate_instance] (output_terminal, input_terminal, enable_terminal});
pass_switch_instance ::= [name_of_ gate_instance] (inout_terminal, inout_terminal, enable_terminal});
pass_enable_switch_instance ::= [name_of_ gate_instance] (inout_terminal, inout_terminal, enable_terminal);
cmos_ switch_ instance ::= [name_of_ gate_instance] (output_terminal, input_terminal,ncontrol_terminal,pcontrol_terminal);
pull_ gate_ instance ::= [name_of_ gate_instance] (output_terminal)
name_of_ gate_instance ::= gate_instance_identifier[range]
pullup_strength ::= (strength0, strength1)
| (strength1, strength0)
| (strength0)
input_terminal ::= scalar_expression
enable_terminal ::= scalar_expression
ncontrol_terminal :: = scalar_expression
pcontrol_terminal :: = scalar_expression
output_terminal ::= terminal_identifier | terminal_identifier[constant_expression]
inout_terminal ::= scalar_expression

n_input_gatetype ::=
and | **nand** | **or** | **nor** | **xor** | **xnor**
n_output_gatetype::=**buf** | **not**
enable_gatetype ::=**bufif0** | **bufif1** | **notif0** | **notif1**
mos_switch_type ::=**nmos** | **rnmos** | **pmos** | **rpmos**
cmos_switch_type ::= **cmos** | **rcmos**
pass_switch_type ::=**tran** | **rtran**
pass_switch_type ::=**tranif0** | **rtranif0** | **tranif1** | **rtranif1**

delay3 ::= #delay_value | #(delay_value [,delay_value [,delay_value]])
delay2 ::= #delay_value | #(delay_value [,delay_value])
delay_value
::=unsigned_number
| parameter_identifier
| (mintypmax_expression [,mintypmax_expression] [,mintypmax_expression])

A.4 Module Instantiations

module_instantiation
::= *module*_identifier [parameter_value_assignment]
module_instance{ ,module_instance}

parameter_value_assignment
::= # (expression{ ,expression})

module_instance
 ::= name_ of_ instance ([list_of_module_connections])

list_of_module_connections
 ::= ordered_port_connection{ , ordered _port_connection}
 | named_port_connection{ ,named_port_connection}

ordered _port_connection
 ::= [expression]

named_port_connection
 ::= . *port*_identifier (expression)

A.5 UDP Declaration and Instantiation

udp_declaration
 ::= **primitive** *udp*_identifier (udp_port_list);
 udp_port_declaration {udp_port_declaration}
 udp_body
 endprimitive

udp_port_list ::= *output_port_identifier*, *input_port*_identifier {, *input_port*_identifier }

udp_port_declaration ::=
 output_declaration
 | reg_declaration
 | input_declaration

udp_body ::= combination_body | sequential_body

combination_body ::= **table** combinational_entry { combinational_entry } **endtable**

sequential_body ::= [UDP_initial_statement] **table** sequential_entry { sequential_entry } **endtable**

UDP_initial_statement
 ::= **initial** output_terminal_name = init_val;

init_val
 ::= **1'b0**
 | **1'b1**
 | **1'bx**
 | **1'bX**
 | **1'B0**
 | **1'B1**
 | **1'Bx**
 | **1'BX**
 | **1**
 | **0**

output_terminal_name
  ::= variable

combinational_entry
  ::= level_input_list: output_symbol;

sequential_entry
  ::= seq_input_list: current_state : next_state ;

seq_input_list
  ::= level_input_list
  | edge_input_Iist

level_input_list
  ::= level_symbol{level_symbol}

edge_input_list
  ::= {level_symbol}edge_indicator {level_symbol}

edge_indicator}
  ::= (level_symbol level_symbol)
  | edge_symbol

current_state
  ::=level_symbol

next_state
  ::=output_symbol
  | -

output_symbol is one of the following characters:
  0 1 x X

level_symbol is one of the following characters:
  0 1 x X ? b B

edge_symbol is one of the following characters:
  r R f F p P n N *

udp_instantiation
  ::=*udp_*identifier [drive_strength] [delay2]
  udp_instance{ ,udp_instance} ;

udp_instance
  ::= [name_of_udp_instance] (output_port_connection, input_port_connection
{output_port_connection ,input_port_connection})
  ::= identifier [range]
name_of_udp_instance::= *udp_instance_*identifier[range]

A.6 Behavioral Statements

initial_statement
    ::= **initial** statement

always_ statement
    ::= **always** statement

statement_or_null
    ::= statement
    |;

statement
    ::=blocking_assignment;
    | non-blocking_assignment;
    | procedural_continuous_assignment
    | procedural_timing_control_assignment
    | conditional_statement
    | case_statement
    | loop_statement
    | wait_statement
    | disable_statement
    | event_trigger
    | seq_block
    | par_block
    | task_enable
    | system_task_enable

procedural_timing_control_assignment ::=
    delay_or_event_control statement_or_null

procedural_continuous_assignment::=
    assign reg_assignment |
    deassign reg_1value
    force reg_assignment
    release reg_1value
    release net_lvalue

conditional_statement ::=
    if (expression) statement_or_null
    | **if** (expression) statement_or_null **else** statement_or_null

case_statement::=
    case (expression) case_item {case_item} **endcase**
    | **casez** (expression) case_item {case_item} **endcase**
    | **casex** (expression) case_item {case_item } **endcase**

loop_statement::=
    forever statement
    | **repeat** (expression) statement

| **while** (expression) statement
| **for** (assignment; expression ; assignment) statement

wait_statement::=
wait (expression) statement_or_null

event_trigger ::=
-> *event*_identifier;

blocking_assignment
::= reg_lvalue **=** [delay_or_event_control] expression ;

non-blocking_assignment
::= lvalue = expression
| lvalue = delay_or_event_control expression ;

delay_or_event_control
::= delay_control
| event_control
| **repeat** (expression) event_control

case_item
::=expression{ ,expression) : statement_or_null
| **default**: statement_or_null

seq_block
::= **begin{** statement} **end**
| **begin** : *block*_identifier{ block_item_declaration}{ statement} **end**

par_block
::= **fork{** statement} **join**
| **fork**: *block*_identifier{ block_item_declaration}{ statement} *block*_identifier
{ statement} **join**

block_declaration
::= parameter_declaration
| reg_declaration
| integer_declaration
| real_declaration
| time_declaration
| event_declaration

task_enable
::=*task*_identifier
| *task*_identifier (expression{ ,expression});

system_task_enable
::= system_task_name;
| system_task_name(expression{ ,expression});
system_task_name

::= $system_identifier (Note: the $ may not be followed by a space.)

A.7 Specify Section

specify_block
::= **specify{** specify_item} **endspecify**

specify_item
::= specparam_declaration
| path_declaration
| system_timing_check

path_declaration
::= simple_path_declaration
| edge_sensitive_path_declaration
| state_dependent_path_declaration

specparam_declaration
::= **specparam** list_of_specparam_assignments;

list_of_specparam_assignments
::=specparam_assignment{, specparam_assignment}

specparam_assignment
::= *specparam_*identifier=constant_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]);

limit_value::=constant_mintypmax_expression;
simple_path_declaration
::= parallel_path_description = path_delay_value ;

parallel_path_description
::= (specify_input_terminal_descriptor[polarity_operator] **=>**
specify _output_terminal_descriptor)

full_path_description::=
input_identifier |
output_identifier[constant_expression]
output_identifier[msb_expression: lsb_constant_expression]

specify_output_terminal_descriptor::=
output_identifier
| output_identifier[constant_expression]
| output_identifier[msb_constant_expression : lsb_constant_expression]

input_identifier ::= *input_port*_identifier | *inout_port*_identifier
polarity_operator ::= + | -
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

    | (list_of_path_inputs{ } 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}

specify_input_terminal_descriptor
    ::= input_identifier
    | input_identifier [constant_expression]
    | input_identifier [constant_expression : constant_expression]

specify_output_terminal_descriptor
    ::= output_identifier
    | output_identifier [constant_expression]
    | output_identifier [constant_expression : constant_expression]

input_identifier
    ::= the IDENTIFIER of a module input or inout terminal

output_identifier
    ::= the IDENTIFIER of a module output or inout terminal.

path_delay_value
    ::= 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)

path_delay_expression
::= constant_mintypmax_expression

system_timing_check
::=**$setup**(timing_check_event, timing_check_event,
timing_check_limit,[notify_register]);
| **$hold(** timing_check_event, timing_check_event, timing_check_limit [,notify_register]);
| **$period(** controlled_timing_check_event, timing_check_limit [,notify_register]);
| **$width(** controlled_timing_check_event, timing_check_limit [
,constant_expression,notify_register]);
| **$skew**(timing_check_event, timing_check_event, timing_check_limit, [,notify_register]);
| **$recovery(** controlled_timing_check_event,
timing_check_event,
timing_check_limit [,notify_register]);
| **$setuphold(** timing_check_event, timing_check_event,
timing_check_limit, timing_check_limit [,notify_register]);

edge_sensitive_path_declaration::=
parallel_edge_sensitive_path_declaration = 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_expresssion))

full_edge_sensitive_path_description ::=
([edge_identifier]list_of_path_inputs*>
list_of_path_outputs[polarity_opertor]: data_source_expression))

data_source_expression ::= expression
edge_identifier ::= **posedge | negedge**
state_dependent_path_declaration ::=
if (conditional_expression) simple_path_declaration
if (conditional_expression) edge_sensitive _path_declaration
ifnone simple_path_declaration

timing_check_event
::= [timing_check_event_control] specify_terminal_descriptor |
&&& [timing_check_condition]

specify_terminal_descriptor
::= specify_input_terminal_descriptor
|specify_output_terminal_descriptor

controlled_timing_check_event
::= timing_check_event_control specify_terminal_descriptor
&&& [timing_check_condition]

timing_check_event_control
    ::= posedge
    | negedge
    | edge_control_specifier

edge_control_specifier
    ::= **edge** [{ edge_descriptor,edge_descriptor}]

edge_descriptor
    ::=01
    | 10
    | 0x
    | x1
    | 1x
    | x0

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_expression
    A scalar expression is a one bit net or a bit-select of an expanded vector net.

timing_check_limit
    ::= expression

scalar_constant
    ::= 1'b0
    | 1'b1
    | 1'B0
    | 1'B1
    | 'b0
    | 'b1
    | 'B0
    | 'B1
    | 1
    | 0

notify_register
    ::= identifier

level_sensitive_path_declaration
 ::= if (conditional_port_expression)
 (specify_input_terminal_descriptor [polarity_operator] =
 specify_output_terminal_descriptor) = path_delay_value;
 | if (conditional_port_expression)
 (list_of_path_inputs [polarity_operator] { }
 list_of_path_outputs) = path_delay_value;
 (Note: The following two symbols are literal symbols, not syntax description conventions:)
 } =

conditional_port_expression
 ::= port_reference
 | unary_operatorport_reference
 | port_referencebinary_operatorport_reference

polarity_operator
 ::=+
 | -

edge_sensitive_path_declaration
 ::= if [(expression)] [(edge_identifier]
 specify _input_terminal_descriptor =
 (specify_output_terminal_descriptor [polarity_operator]
 : data_source_expression)) = path_delay_value;
 | if [(expression)] [(edge_identifier]
 specify_input_terminal_descriptor{ }
 (list_of_path_outputs [polarity_operator]
 : data_source_expression)) =path_delay_value;

data_source_expression
 Any expression, including constants and lists. Its width must be one bit or
 equal to the destination's width. If the destination is a list, the data
 source must be as wide as the sum of the bits of the members.

edge_identifier
 ::= posedge
 | negedge

sdpd
 ::=if(conditional_expression)path_description=path_delay_value;

sdpd_conditional_expresssion
 ::=expressionB INARY_OPERATOR expression
 |UNARY_OPERATOR expression

A.8 Expressions

net_lvalue
::= net_dentifier
| *net*_identifier [expression]
| *net*_identifier [constant_expression : constant_expression]
| *net*_concatenation

reg_lvalue ::=
*reg*_identifier
| *reg*_identifier[expression]
| *reg*_identifier[msb_constant_expression: lsb_constant_expression]
| *reg*_concatenation

constant_expression
::= constant_primary
| unary_operator constant_primary
| constant_expression binary_operator constant_expression
| constant_expression ? constant_expression : constant_expression
| string

constant_primary ::=
number
| *parameter*_identifier
| *constant*_concatenation
| *constant*_multiple_ concatenation

constant_mintypmax_expression ::=
constant_expression
| constant_expression : constant_expression : constant_expression

unary_operator ::=
+ | - | ! | ~ | | ~& | | | ~ | | ^ | ~^ | ^ ~

mintypmax_expression
::= expression
| expression : expression : expression

expression
::= primary
| unary_operator primary
| expression binary_operator expression
| expression question_mark expression : expression
| string

binary_operator ::=
+ - { } / % == != === !== && ‖ = = & | ^ ^~

QUESTION_MARK is ? (a literal question mark).

STRING is text enclosed in "" and contained on one line.

primary
    ::= number
    | identifier
    | identifier [expression]
    | identifier [msb_constant_expression : lsb_constant_expression]
    | concatenation
    | multiple_concatenation
    | function_call
    | (mintypmax_expression)

number
    ::= decimal_number
    | octal_number
    | binary_number
    | hex_number
    | real_number

real_number ::=
    [sign] unsigned_number.unsigned_number
    [sign] unsigned_number.[.unsigned_number]**e**[sign]unsigned_number
    [sign] unsigned_number.unsigned_number

decimal_number ::=
    [sign] unsigned_number |
    [size] decimal_base unsigned_number

binary_number ::= [size] binary_base binary_digit { _ | binary_digit}
octal_number ::= [size] octal_base octal_digit { _ | binary_digit}
hex_number ::= [size] hex_base hex_digit { _ | hex_digit}
sign ::= + | -
size ::= unsigned_number
unsigned_number ::= decimal_digit { _ | decimal_digit}
decimal_base ::= ‘d | ‘D
binary_base ::= ‘b | ‘B
octal_base ::= ‘o | ‘O
decimal_number
    ::= A number containing a set of any of the following characters, optionally preceded by + or-
        0|1|2|3|4|5|6|7|8|9_

hex_digit **::=0|1|2|3|4|5|6|7|8|9|a|b|c|d|e|f|A|B|C|D|E|F|x|X|z|Z**

BASE is one of the following tokens:
    'b 'B 'o 'O 'd 'D 'h 'H

concatenation
    ::= { expression{ ,expression} }

multiple_concatenation
 ::= { expression { expression{ ,expression} } }

function_call
 ::= *function*_identifier (expression{ , expression})
 | system_function_identifier (expression{ ,expression})
 | system_function

system__function_identifier
 ::=_$identifier

A.9 General

comment
 ::= short_comment
 | long_comment

short_comment
 ::= // comment_text END-OF-LINE

long_comment
 ::={ /* comment_text */

comment_text
 ::= { Any_ASCII_character }

identifier::=
 simple_identifier |
 escaped_identifier

simple_identifier ::= [a-zA-Z[a-zA-Z_$]
escaped_identifier ::= \{Any_ASCII_character_except_white_space} white_space
white_space ::= space | tab | newline

NOTE - The period in identifier may not be preceded or followed by a space.

{.identifier}
 (note: the period may not be preceded or followed by a space.)

identifier
 An identifier is any sequence of letters, digits, dollar signs ($), and
 underscore (_) symbol, except that the first must be a letter or the
 underscore; the first character may not be a digit or $. Upper and lower case
 letters are considered to be different. Identifiers may be up to 1024
 characters long. Some Verilog-based tools do not recognize {identifier}characters
 beyond the 1024th as a significant part of the identifier. Escaped
 identifiers start with the backslash character (\) and may include any
 printable ASCII character. An escaped identifier ends with white space. The
 leading backslash character is not considered to be part of{ the} identifier. }delay
 ::=# number

| # identifier
| # (mintypmax_expression [,mintypmax_expression]
[,mintypmax_expression])

delay_control
::=# number
| # identifier
|# (mintypmax_expression)

event_control
::= @ identifier
| @ (event_expression)

event_expression
::= expression
| posedge scalar_event_expression
| negedge scalar_event_expression
| event_expression or event_expression

scalar_event_expression
Scalar event expression is an expression that resolves to a one bit value.

B VERILOG SUBSET FOR LOGIC SYNTHESIS

B.1 Introduction

The following pages describe the Verilog language support for logic synthesis in a formal manner.

B.2 Syntax for Verilog for Logic Synthesis

The following pages describe the Verilog language support for logic synthesis in a formal manner.

The syntax definition uses the same notation used for describing the full language in the prior appendix.

We also define the keywords constructs that are not supported in logic synthesis tools in the later sections of this appendix.

Syntax

This section presents the syntax of the supported Verilog language.

BNF Syntax

```
source_text
  ::= {syn_description}

syn_description
     ::= syn_module_declaration

syn_module_declaration
     ::=module_keyword module_identifier [list_of_ports]
         {syn_module_item}
     endmodule
```

```
module_keyword
        ::= module | macromodule

name_of_module
   ::=  IDENTIFIER

list_of_ports
   ::=( port  {,port})
   |( )

port
   ::=  [port_expression]
   | .port_identifier ( [port_expression] )

port_expression
   ::=  port_reference
   | { port_reference {, port_reference} }

port_reference
   ::=port_identifier  |  port_identifier [ expression ]
   |port_identifier [ expression : expression]

syn_module_item
   ::=  parameter_declaration
   | input_declaration
   | output_declaration
   | inout_declaration
   | net_declaration
   | reg_declaration
   | integer_declaration
   | syn_gate_instantiation
   | module_instantiation
   | continuous_assign
   | function_declaration
   | syn_always_statement

function_declaration
   ::= function [range]function_identifier;
            {function_item_declaration}
            syn_statement
      endfunction

function_item_declaration
   ::=  parameter_declaration
   | input_declaration
   | reg_declaration
   | integer_declaration
```

```
syn_always_statement
  ::= always @ ( identifier or identifier )
  | always @ ( posedge identifier )
  | always @ ( negedge identifier )
  | always @ ( egde or edge or ... )

edge
  ::= posedge identifier
  | negedge identifier

parameter_declaration
  ::= parameter [range] list_of_assignments ;

input_declaration
  ::= input [range] list_of_variables ;

output_declaration
  ::= output [range] list_of_variables ;

inout_declaration
  ::= inout [range] list_of_variables;

syn_net_declaration
  ::= syn_NET_TYPE [charge_strength] [expandrange] [delay]

list_of_variables;
  | NET_TYPE [drive_strength] [expandrange] [delay]

list_of_assignments;

syn_NET_TYPE
  ::=wire
  | wor
  | wand
  | tri

expandrange
  ::= range

  | scalared range
  | vectored range

reg_declaration
  ::= reg [range] list_of_register_variables ;

integer_declaration
  ::= integer list_of_integer_variables;

continuous_assign
  ::= assign [drive_strength] [delay]
          list_of_assignments;
```

```
list_of_variables
  ::=variable_identifier   {,variable}

list_of_register_variables
  ::= register_variable  {, register_variable)

register_variable
  ::= IDENTIFIER

list_of_integer_variables
  ::= integer_variable {, integer_variable}

integer_variable
  ::=  IDENTIFIER

charge_strength
  ::= ( small )
  | ( medium )
  | ( large )

drive_strength
  ::= ( STRENGTH0, STRENGTH1 )
  | ( STRENGHT1 , STRENGTH0 )

STRENGTH0
  ::=  supply0
  | strong0
  | pull0
  | weak0
  | highz0

STRENGTH1
  ::= supply1
  | strong 1
  | pull1
  | weak1
  | highz1

range
  ::= [ expression : expression ]

list_of_assignments
  ::= assignment {, assignment}

syn_gate_instantiation
  ::= syn_GATETYPE [drive_strength] [delay]
        gate_instance {, gate_instance};
```

```
syn_GATETYPE
  ::=and
  | nand
  | or
  | nor
  | xor
  | xnor
  | buf
  | not

gate_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};

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, input_terminal});

output_terminal
  ::= identifier
  | expression

input_terminal
  ::= identifier
  | expression

module_instantiation
  ::=module_identifier [parameter_value_assignment]
     module_instance {, module_instance}

parameter_value_assignment
  ::= #( expression {,expression})

module_instance
  ::= module_instance_identifier ( [list_of_module_connections] )

list_of_module_connections
     ::= ordered_port_connection{, ordered _port_connection}
     | named_port_connection{ ,named_port_connection}

ordered _port_connection
     ::= [expression]

named_port_connection
     ::= . port_identifier (expression)
```

```
syn_statement
   ::=  assignment
   | if ( expression )
          statement
   | if ( expression )
          statement
       else
          statement
   | case ( expression )
          case_item+
       endcase
   | casex (expression )
          case_item+
       endcase
   | casez ( expression )
          case_item+
       endcase
   | for ( assignment; expression ; assignment )
          statement
   | seq_block
   | disable IDENTIFIER;
   | forever statement
   | while ( expression ) statement

statement
   ::=  statement
   | ;

assignment
   ::= 1value = expression

case_item
   ::= expression {,expression} : statement
   | default: statement
   | default statement

seq_block
   ::=  begin
           {statement}
        end
   | begin :block_identifier          {block_declaration}
           {statement}
     end


block_declaration
   ::= parameter_declaration
   | reg_declaration
   | integer_declaration
```

```
1value
  ::= IDENTIFIER
  | IDENTIFIER [ expression ]
  | concatenation

expression
  ::= primary
  |UNARY_OPERATOR primary
  | expression BINARY_OPERATOR
  | expression [expression: expression]

UNARY_OPERATOR
  ::= !
  | ~
  | &
  | ~&
  | |
  | ~|
  | ^
  | ~^
  | -
  | +
  | -
  | }
  | /
  | %
  | ==
  | !=
  | &&
  | ||
  |
  | =
  |
  | =
  | &
  ||
  |
  |

primary
  ::= number
  I identifier
  I identifier [ expression ]
  | identifier [ expression: expression ]
  | concatenation
  | multiple_concatenation
  | function_call
  | (expression)
```

number
::= NUMBER
| BASE NUMBER
| SIZE BASE NUMBER

Number

A number can have any of the following characters:
0123456789abcdefxzABCDEFXZ

Size

::='b
|'B
|'o
|'O
|'d
|'D
|'h
|'M

Size

Any number of the following digits: 0123456789

concatenation
::= { expression {,expression} }

multiple_concatenation
::= { expression { expression {,expression}

identifier
An identifier is any sequence of letters, digits, and the under- score character (_), where the first character is a letter or underscore. Uppercase and lowercase letters are treated as different characters. Identifiers can be any size and all characters are significant. Escaped identifiers start with the backslash character (\) and end with a space. The leading backslash character (\) is not part of the identifier. Use escaped identifi- ers to include any printable ASCII {characters}in}an }identifier.

delay
::= # NUMBER
| # identifier
| # (expression {,expression})

Compiler Directives

The following compiler directives are supported:
`include
`define

B.3 Ignored Constructs for Logic Synthesis

B.3.1 Compiler Directives

The following compiler directives are ignored (only useful for simulation):

`accelerate
`celldefine
` default_net_type
`endcelldefine
`endprotect
` expand_vectornets
`noaccelerate
`noexpand_vectornets`noremove_netnames
`nounconnected_drive`protect
` remove_netnames
`resetall
`timescale
`unconnected_drive

B.3.2 Verilog System Functions

Verilog system function and task names start with a dollar sign ($).
These functions and tasks are parsed and ignored by HDL Compiler.
Examples of these are:
$monitor
$display
$dumpfile, etc.

B.4 Unsupported Verilog Language Constructs

Some Verilog constructs are not supported in the synthesis subset.
These constructs are described in this section.

B.4.1 Unsupported Definitions and Declarations

primitive definition
time declaration
event declaration
triand, trior, tri1, tri0, and trireg net types
Ranges and arrays for integers

Unsupported Statements

defparam statement

initial statement

repeat statement

delay control

event control

wait statement

fork statement

deassign statement

force statement

release statement

Assignment statement with a variable used as a bit-select on the left side of the equal sign

Unsupported Operators

Case equality and inequality operators (=== and !==)

Division and modulus operators for variables

Unsupported Gate-Level Constructs

nmos, pmos, cmos, rnmos, rpmos, rcmos, pullup, pulldown, tranif0, tranif1, rtran, rtranif0, and rtranif1 gate types

Unsupported Miscellaneous Constructs

Hierarchical names within a module

`ifdef, `endif and `else compiler directives

B.5 Verilog Keywords Set for Logic Synthesis

always	and	assign	begin
buf	bufif0	bufif1	case
casex	casez	cmos	deassign
default	defparam	disable	else
end	case	endunction	endmodule
endprimitive	endtable	endtask	event
for	force	forever	fork
function	highz0	highz1	if
initial	inout	input	integer
join	large	medium	module
nand	negedge	nmos	nor
not	notif0	notif1	or
output	parameter	pmos	posedge
primitive	pulldown	pullup	pull0
pull1	rcmos	reg	release
repeat	rnmos	rpmos	rtran
rtranif0	rtranif1	scalared	small

strong0	strong1	supply0	supply1
supply1	table	task	time
tran	tranif0	tranif1	tri
triand	trior	trireg	tri0
tri1	vectored	wait	wand
weak0	weak1	while	wire
wor	xnor	xor	

C PROGRAMMING LANGUAGE INTERFACE (PLI)
Header File - veriuser.h

```
/*************************************************************************
 * veriuser.h
 *
 * IEEE 1364 1995 Verilog HDL Programming Language Interface (PLI).
 *
 * This file contains the constant definitions, structure definitions, and
 * routine declarations used by the Verilog PLI procedural interface TF
 * task/function routines.
 *
 * The file should be included with all C routines that use the PLI TF
 * routines.
This file is annotated  and contains explanations f the routines , data structures and the
parameters */
#ifndef VERIUSER_H
#define VERIUSER_H


/*-
- ----- definitions   [of constants] -----
*/
#define true 1
#define TRUE 1
#define false 0
#define FALSE 0
#define bool int

/*------ - --- -------- defines for error interception -------*/
#define ERR_MESSAGE 1
```

```
#define ERR_WARNING 2
#define ERR_ERROR 3
#define ERR_INTERNAL 4
#define ERR_SYSTEM 5


/*----- - --- - values for reason parameter to misctf routines */
/* (misctf routines are typically called for value changes, but they can also be called for
various other reasons; most of these reasons
      involve a coordinated activity on part of the user-defined C tasks and functions that is
consistent with the actions of the simulator.
The parts of simulation cycle or different activities involved in simulation include:
      compilation
      task or function call
      save and restart of simulation state
      disabling of a task
      parameter value change
      synchronizing the times of the C function and the simulation engine
      completion if simulation
      completion of compilation
      enter interactive mode
      force a value
      release a value
)*/


#define reason_checktf 1
#define REASON_CHECKTF reason_checktf
#define reason_sizetf 2
#define REASON_SIZETF reason_sizetf
#define reason_calltf 3
#define REASON_CALLTF reason_calltf
#define reason_save 4
#define REASON_SAVE reason_save
#define reason_restart 5
#define REASON_RESTART       reason_restart
#define reason_disable       6
#define REASON_DISABLE       reason_disable
#define reason_paramvc       7
#define REASON_PARAMVC       reason_paramvc
#define reason_synch         8
#define REASON_SYNCH         reason_synch
#define reason_finish        9
#define REASON_FINISH        reason_finish
#define reason_reactivate    10
#define REASON REACTIVATE    reason_reactivate
#define reason_rosynch       11
#define REASON_ROSYNCH       reason_rosynch
#define reason_paramdrc      15
#define REASON_PARAMDRC      reason_paramdrc
```

```
#define reason_endofcompile   16
#define REASON_ENDQFCOMPILE   reason_endofcompile
#define reason_scope          17
#define REASON_SCOPE          reason_scope
#define reason_interactive    18
#define REASON_INTERACTIVE    reason_interactive
#define reason_reset          19
#define REASON_RESET          reason_reset
#define reason_endofreset     20
#define REASON_ENDOFRESET     reason_endofreset
#define reason_force          21
#define REASON_FORCE          reason_force
#define reason_release        22
#define REASON_RELEASE        reason_release
#define reason_startofsave    27
#define reason_startofrestart 28
#define REASON_MAX            28


/*-- types used by tf_typep() and expr type field in tf_exprinfo structure --*/
#define tf_nullparam      0
#define TF_NULLPARAM      tf_nullparam
#define tf_string         1
#define TF_STRING         tf_string
#define tf_readonly       10
#define TF_READONLY       tf_readonly
#define tf_readwrite      11
#define TF_READWRITE      tf_readwrite
#define tf_rwbitselect    12
#define TF_RWBITSELECT    tf_rwbitselect
#define tf_rwpartselect   13
#define TF_RWPARTSELECT   tf_rwpartselect
#define tf_rwmemselect    14
#define TF_RWMEMSELECT    tf_rwmemselect
#define tf_readonlyreal   15
#define TF_READONLYREAL  tf_readonlyreal
#define tf_readwritereal  16
#define TF_READWRITEREAL  tf_readwritereal


/*--------- types used by node_type field in tf_nodeinfo structure - -------*/
#define tf_null_node        100
#define TF_NULL_NODE        tf_null_node
#define tf_reg node         101
#define TF_REG_NODE         tf_reg_node
#define tf integer_node     102
#define TF_INTEGER_NODE     tf_integer_node
#define tf_time_node        103
#define TF_TIME_NODE        tf_time_node
#define tf_netvector_node   104
```

```
#define TF_NETVECTOR_NODE     tf_netvector_node
#define tf_netscalar_node      105
#define TF_NETSCALAR_NODE   tf_netscalar_node
#define tf_memory_node       106
#define TF_MEMORY_NODE       tf_memory_node
#define tf_real_node          107
#define TF_REAL_NODE          tf_real_node
/*--                        -*/
/* -------------- ---------- structure definitions --------- - -- -----------*/
/********** here we define the data structures used for exchanging information between the
simulator and the C application
*/


/*--                        -*/
/*--- structure used with tf_exprinfo() to get expression information - - -*/
typedef struct t_tfexprinfo
{
      short expr_type;
      short padding;
      struct t_vecval *expr_value_p;
      double real_value;
      char *expr_string;
      int expr_ngroups;
      int expr_vec_size;
      int expr_sign;
      int expr_lhs_select;
      int expr_rhs_select;
}s_tfexprinfo,*p_tfexprinfo;


/*------- structure for use with tf nodeinfo() to get node information ------ */
typedef struct t_tfnodeinfo
{
      short node_type;
      short padding;
      union
      {
            struct t_vecval *vecval_p;
            struct t_strengthval  *strengthval_p;
            char *memoryval_p;
            double  *real_val_p;
      } node_value;
      char  *node_symbol;
      int  node_ngroups;
      int  node_vec_size;
      int  node_sign;
      int  node_ms_index;
      int  node_ls_index;
      int  node_mem_size;
```

```
    int node_lhs_element;
    int node_rhs_element;
    int  *node_handle;
 }
 s_tfnodeinfo, *p_tfnodeinfo;


/*------- -- --- -- -- data structure of vector values --- -- - - ---- --- /
  -*/
typedef  struct  t_vecval
{
    int avalbits;
    int bvalbits;
 } s_vecval, *p_vecval;


/*------------- data structure of scalar net strength values ------  -- -*/


typedef struct t_strengthval
{
    int strength0;
    int strengthl;
 } s_strengthval,*p_strengthval;
/*‾‾--          */
/*- - - - routine definitions - - -- --- -- ---*/
/*--____ _____ _ ____                              */
/*                                            -*/
#if defined(__STDC__) || defined(__cplusplus)


#ifndef PROTO_PARAMS
#define PROTO_PARAMS(params) params
#define DEFINED_PROTO_PARAMS
#endif
#ifndef EXTERN
#define EXTERN
#define DEFINED_EXTERN
#endif


#else


#ifndef PROTO_PARAMS
#define PROTO_PARAMS(params) (/* nothing */)
#define DEFINED_PROTO_PARAMS
#endif
#ifndef EXTERN
#define EXTERN extern
```

```
#define DEFINED_EXTERN
#endif


#endif /* STDC */


/**** Following is a list of routines provided to the user for interacting with the simulator
using the io tasks and functions */
/** Explanation for each routine is provided prior to its prototype **/


/* Following are the input-output routines */
/* The following will display a value to multi-channel descriptor */
EXTERN void    io_mcdprintf PROTO_PARAMS((int mcd,char *format,..));


/* The following will display to the standard output */
EXTERN void   io_printf PROTO_PARAMS((char *format,..));


/** The following task will allow creating a plus argument to the Verilog Simulator to be
used in User application **/
EXTERN char   *mc_scan_plusargs PROTO_PARAMS((char *plusarg));


/** Add 64 bit time variables */
EXTERN int   tf_add_long PROTO_PARAMS((int *aof_lowtimel,
int *aof_hightimel,int lowtime2,int hightime2));


/*** Set the asynchronous (or misctf) calls to off for this C function**/
EXTERN int   tf_asynchoff PROTO_PARAMS((void));


/* Set the asynchronous (or misctf) calls to on to this C function **/
EXTERN int  tf_asynchon   PROTO_PARAMS((void));


/* Clear all delays to be able to do zero delay simulation */
EXTERN int     tf_clearalldelays PROTO_PARAMS((void));


/**** Copy the flags **/
EXTERN int     tf_copypvc_flag PROTO_PARAMS((int nparam));


/* Divide 2 64-bit time values **/
EXTERN void     tf_divide_long PROTO_PARAMS((int *aof_lowl,
int *aof_highl,int low2,int high2));
```

```
/*** Perform the $finish on the simulator from C program **/
EXTERN int      tf_dofinish PROTO_PARAMS((void));


/* Perform the $stop from the C program on the simulator **/
EXTERN int      tf_dostop PROTO_PARAMS((void));


/******** Set an error message on **/
EXTERN int      tf_error PROTO_PARAMS((char *fmt,...));


/************ Evaluate an expression present in the parameter *************/
EXTERN int      tf_evaluatep PROTO_PARAMS((int pnum));


EXTERN p_tfexprinfo tf_exprinfo PROTO_PARAMS((int pnum,p_tfexprinfo pinfo));


/********* Get the string  parameter **/
EXTERN char     *tf_getcstringp PROTO_PARAMS((int nparam));


/* get instance id for a particular module which calls the C task or fucntion **/
EXTERN char     *tf_getinstance PROTO_PARAMS((void));



/********Get a pointer to 64 bit parameter value **/
EXTERN int      tf_getlongp PROTO_PARAMS((int *aof_highvalue,int pnum));


/********* Get a pointer to high order bytes of 64 bit time **/
EXTERN int      tf_getlongtime PROTO_PARAMS((int *aof_hightime));


/************ Get next parameter's time value **/
EXTERN int      tf_getnextlongtime PROTO_PARAMS((int *aof_lowtime, int
*aof_hightime));


/*** Obtain value of a parameter - number pnum **/
EXTERN int      tf_getp PROTO_PARAMS((int pnum));


/* obtain the value-change flag of the parameter **/
EXTERN int      tf_getpChange PROTO_PARAMS((int nparam));
```

```
/****************** get real parameter value *************************/
EXTERN double tf_getrealp PROTO_PARAMS((int pnum));

/*************** get real-time ********************/
EXTERN double tf_getrealtime PROTO_PARAMS((void));

/********************** get current simulation time *******************/
EXTERN int tf_gettime PROTO_PARAMS((void));

/* Get the current timescale information - the precision part of it **/
EXTERN int tf_gettimeprecisian PROTO_PARAMS((void));

/* Get the current timescale information - the unit part of it **/
EXTERN int tf_gettimeunit PROTO_PARAMS((void));

/* Obtain memory via Verilog's memory -allocation scheme **/
EXTERN char *tf_getworkarea PROTO_PARAMS((void));

/************ The following set of routines is similar to other set of routines - but is
instance-based ******/

/**** Instance-based misctf calls - asynchronous calls off ***/
EXTERN int tf_iasynchoff PROTO_PARAMS((char *inst));

/**** Instance-based misctf calls for the current "C" function - asynchronous calls on ***/
EXTERN int tf_iasynchon PROTO_PARAMS((char *inst));

EXTERN int tf_iclearalldelays PROTO_PARAMS((char *inst));
EXTERN int tf_icopypvc_flag PROTO_PARAMS((int nparam,char *inst));
EXTERN int tf_ievaluatep PROTO_PARAMS((int pnum,char *inst));

/********** Get expression-node (parameter) information *****/
EXTERN p_tfexprinfo tf_iexprinfo PROTO_PARAMS((int pnum,p_tfexprinfo pinfo,
char *inst));

/***************** Get String for a parameter for a particular instance ****/
EXTERN char *tf_igetcstringp PROTO_PARAMS((int nparam, char *inst));
EXTERN int tf_igetlongp PROTO_PARAMS((int *aof_highvalue,int pnum,
```

```
char  *inst));
EXTERN int      tf_igetlongtime PROTO_PARAMS((int *aof_hightime,
char *inst));
EXTERN int      tf_igetp PROTO_PARAMS((int pnum,char *inst));
EXTERN int      tf_igetpchange PROTO_PARAMS((int nparam,char *inst));
EXTERN double   tf_igetrealp PROTO_PARAMS((int pnum,char *inst));
EXTERN double   tf_igetrealtime  PROTO_PARAMS((char *inst));
EXTERN int     tf_igettime PROTO_PARAMS((char *inst));
EXTERN int     tf_igettimeprecision PROTO_PARAMS((char *inst));
EXTERN int     tf_igettimeunit PROTO_PARAMS((char *inst));
EXTERN char    *tf_igetworkarea PROTO_PARAMS((char *inst));
EXTERN char    *tf_imipname PROTO_PARAMS((char *cell));
EXTERN int     tf_imovepvc_flag PROTO_PARAMS((int nparam, char *inst));
EXTERN p_tfnodeinfo tf_inodeinfo PROTO_PARAMS((int pnum,p_tfnodeinfo pinfo,
char *inst));
EXTERN int     tf_inump PROTO_PARAMS((char *inst));
EXTERN int     tf_ipropagatep PROTO_PARAMS((int pnum,char *inst));
EXTERN int     tf_iputlongp PROTO_PARAMS((int pnum,int lowvalue,
int highvalue,char *inst));
EXTERN int     tf_iputp PROTO_PARAMS ((int pnum,int value,char *inst));
EXTERN int     tf_iputrealp PROTO_PARAMS((int pnum,double value, char *inst));


EXTERN int   tf_irosynchronize PROTO_PARAMS((char *inst));
EXTERN int   tf_isetdelay PROTO_PARAMS((int delay,char *inst));
EXTERN int   tf_isetlongdelay PROTO_PARAMS((int lowdelay,
int highdelay,char *inst));
EXTERN int   tf_isetrealdelay PROTO_PARAMS((double realdelay,
char *inst));
EXTERN int   tf_isetworkarea PROTO_PARAMS((char *workare,
char *inst));
EXTERN int   tf_isizep PROTO_PARAMS((int pnum,char *inst));
EXTERN char  *tf_ispname PROTO_PARAMS((char *cell));
EXTERN int   tf_istrdelputp PROTO_PARAMS ((int nparam,int bitlength,          int
format_char, char *value_p,   int
delay,int delaytype,char *inst));

EXTERN char   *tf_istrgetp PROTO_PARAMS((int pnum,int format_char,char *inst));
EXTERN int    tf_istrlongdelputp PROTO_PARAMS((int nparam,       int bitlenqth,int
format_char,char *value_p,
int lowdelay,int highdelay,int delaytype,            char *inst));


EXTERN int    tf_istrrealdelputp PROTO_PARAMS((int nparam,            int bitlength,int
format_char,char *value_p,
double realdelay,int delaytype,char *inst));
EXTERN int    tf_isynchronize PROTO_PARAMS ((char *inst));
EXTERN int    tf_itestpvc_flag PROTO_PARAMS((int nparam,char *inst));
EXTERN int    tf_itypep PROTO_PARAMS((int pnum,char *inst));
EXTERN void   tf_long_to_real PROTO_PARAMS((int lo,int hi,          double *aof_real));
```

```
/*************************************************************************/
/* Convert a 64-bit quantity to a string value **/
EXTERN char   *tf_longtime_tostr PROTO_PARAMS((int lowtime,
int  hightime));
/**** Display a message on the screen **/
EXTERN int    tf_message PROTO_PARAMS((int level,char *facility,             char
*messno,char *message,..));

/********** ..................... **************************/
EXTERN char   *tf_mipname PROTO_PARAMS((void));
EXTERN int     tf_movepvc_flag PROTO_PARAMS((int nparam));
EXTERN void    tf_multiply_long  PROTO_PARAMS((int *aof_lowl,            int
*aof_highl,int low2,int high2));


/*************** Obtain information about the node  **************************/
EXTERN p_tfnodeinfo tf_nodeinfo PROTO_PARAMS((int pnum, p_tfnodeinfo pinfo));


/* Obtain number of parameters to this C function ***/
EXTERN int  tf_nump PROTO_PARAMS((void));
EXTERN int  tf_propagatep  PROTO_PARAMS((int pnum));
EXTERN int  tf_putlongp PROTO_PARAMS((int pnum,int lowvalue,             int
highvalue));


/* write back a value of the parameter ***/
EXTERN int   tf_putp PROTO_PARAMS((int pnum,int value));
EXTERN int   tf_putrealp  PROTO_PARAMS((int  pnum,double  value));


EXTERN int   tf_read_restart PROTO_PARAMS((char *blockptr,             int blocklen));


/** Convert a real number to a 64 bit long number **/


EXTERN void   tf_real_to_long PROTO_PARAMS((double real,    int *aof_int_lo,int
*aof_int_hi));
EXTERN int    tf_rosynchronize  PROTO_PARAMS((void));
/* Following set of routines help get the delays adjusted ***/
/* Scale delays by a factor **/
EXTERN void   tf_scale_longdelay PROTO_PARAMS((char *cell, int delay_lo,int
delay_hi,int *aof_delay_lo,     int *aof delay_hi));
EXTERN void   tf_scale_realdelay  PROTO_PARAMS((char *cell,  double realdelay,double
*aof_realdelay));
EXTERN int     tf_setdelay  PROTO_PARAMS((int delay));
EXTERN int     tf_setlongdelay PROTO_PARAMS((int lowdelay,  int highdelay));
EXTERN int     tf_setrealdelay PROTO_PARAMS((double realdelay));
```

```
/** Set the work area to a pointer previously obtained using tf_getarea ***/
EXTERN int  tf_setworkarea PROTO_PARAMS((char *workarea));
EXTERN int  tf_sizep PROTO_PARAMS ((int pnum));
EXTERN char  *tf_spname PROTO_PARAMS((void));
/******************** Write  a delay  value  in  the  form  of  a  string ******************/
EXTERN int  tf_strdelputp PROTO_PARAMS((int nparam,int bitlength,            int
format_char, char *value_p, int delay,
int  delaytype));

/*****  Obtain a parameter value in the form of a string ****/
EXTERN char   *tf_strgetp PROTO_PARAMS((int pnum,int format_char));
/** Obtain a time parameter value in the form of a string **/
EXTERN char   *tf_strgettime PROTO_PARAMS((void));
/******* Put a delay value into a parameter ***/
EXTERN int    tf_strlongdelputp PROTO_PARAMS((int nparam,  int bitlength, int
format_char, char *value_p,      int lowdelay,
int highdelay,int delaytype));
/************       Put a parameter that is a delay in the real format **/
EXTERN int   tf_strrealdelputp PROTO_PARAMS((int nparam,    int bitlength,int
format_char,char *value_p,            double
realdelay.int delaytype));

/**  Subtract 2 64-bit quantities ***/
EXTERN int   tf_subtract_long PROTO_PARAMS((int *aof_lowtimel,            int
*aof_hightimel,  int lowtime2,  int hightime2));

EXTERN int   tf_synchronize PROTO_PARAMS((void));
EXTERN int   tf_testpvc_flag PROTO_PARAMS((int nparam));
EXTERN int   tf_text PROTO_PARAMS((Char *fmt,...));
EXTERN int   tf_typep PROTO_PARAMS((int pnum));
EXTERN void  tf_unscale_longdelay PROTO_PARAMS((char *csll, int delay_lo,int
delay _hi,int *aof_delay_lo, int *aof_delay_hi));
EXTERN void  tf_unscale_realdelay PROTO_PARAMS((char *cell,   double
realdelay,double *aof_realdelay));

/*** Issue a warning out on the screen ****/
EXTERN int    tf_warning PROTO_PARAMS ((char *fmt,...));
EXTERN int    tf_write_save PROTO_PARAMS((char *blockptr,             int blocklen));
#ifdef DEFINED_PROTO_PARAMS
#undef DEFINED_PROTO_PARAMS
#undef  PROTO_PARAMS
#endif

#ifdef DEFINED_EXTERN
#undef DEFINED_EXTERN
#undef EXTERN
#endif

#endif/*VERIUSER H*/
```

D PROGRAMMING LANGUAGE INTERFACE (PLI)

Header File - acc_user.h

```
/* Appendix D  Programming Language Interface -  Header File Listing acc_user.h*/
/**************************************************************************************
 * acc_user.h
 *
 * IEEE 1364 1995 Verilog HDL Programming Language Interface (PLI).
 *
 * This file contains the constant definitions, structure definitions, and
 * routine declarations used by the Verilog PLI procedural interface ACC
 * access routines.
 *
 * The file should be included with all C routines that use the PLI ACC
 * routines.
 ***********************************************************************/
#ifndef ACC_USER_H
#define ACC_USER_H
/* - - - -----       - definitions            ------ -          */
/*-- ---- - ---- -- ----- -   general defines - ----- - -------- ----*/
typedef int *HANDLE;
typedef int *handle;
#define bool int
#define true 1
#define TRUE 1
#define false 0
#define FALSE 0
#define global extern
#define exfunc
#define localstatic
#define accNet 25
#define accReg 30
```

```
#define accRegister accReg
#define accPort 35
#define accTerminal 45
#define accInputTerminal 46
#define accOutputTerminal 47

#define accInoutTerminal 48
#define accCombPrim 140
#define accSeqPrim 142
#define accAndGate 144
#define accNandGate 146
#define accNorGate 148
#define accOrGate 150
#define accXorGate 152
#define accXnorGate 154
#define accBufGate 156
#define accNotGate 158
#define accBufifOGate 160
#define accBufiflGate 162
#define accNotifOGate 164
#define accNotiflGate 166
#define accNmosGate 168
#define accPmosGate 170
#define accCmosGate 172
#define accRnmosGate 174
#define accRpmosGate 176
#define accRcmosGate 178
#define accRtranGate 180
#define accRtranifOGate 182
#define accRtraniflGate 184
#define accTranGate 186
#define accTranifOGate 18B
#define accTraniflGate 190
#define accPullupGate 192
#define accPulldownGate 194
#define accIntegerParam 200
#define accIntParam acc1ntegerParam
#define accRealParam 202
#define accStringParam 204
#define accPath 206
#define accTchk 208
#define accPrimitive 210
#define accPortBit 214
#define accNetBit 216
#define accRegBit 21B
#define accParameter 220
#define accSpecparam 222
#define accTopModule 224
#define accModuleInstance 226
#define accCellInstance 228
```

```
#define accModPath 230
#define accWirePath 234
#define accInterModPath 236
#define accScalarPort 250
#define accPartSelectPort 254
#define accVectorPort 256
#define accConcatPort 258
#define accWire 260
#define accWand 261
#define accWor 262
#define accTri 263
#define accTriand 264
#define accTrior 265
#define accTriO 266
#define accTril 267
#define accTrireg 268
#define accSupply0 269
#define accSupplyl 270
#define accNamedEvent 280
#define accEventVar accNamedEvent
#define accIntegerVar 281
#define accIntVar 281
#define accRealVar 282
#define accTimeVar 283
#define accScalar 300
#define accVector 302
#define accCollapsedNet 304
#define accExpandedVector 306
#define accUnExpandedVector 307
#define accSetup 366
#define accHold 367
#define accWidth 368
#define accPeriod 369
#define accRecovery 370
#define accSkew 371
#define accNochange 376
#define accNoChange accNochange
#define accSetuphold 377
#define accInput 402
#define acc0utput 404
#define accInout 406
#define accMixedIo 407
#define accPositive 408
#define accNegative 410
#define accUnknown 412
#define accPathTerminal 420
#define accPathInput 422
#define accPathOutput 424
#define accDataPath 426
#define accTchkTerminal 428
```

```
#define accBitSelect 500
#define accPartSelect 502
#define accTask 504
#define accFunction 506
#define accStatement 508
#define accSystemTask 514
#define accSystemFunction 516
#define accSystemRealFunction 518
#define accUserTask 520
#define accUserFunction 522
#define accUserRealFunction 524
#define accNamedBeginStat 560
#define accNamedForkStat 564
#define accConstant 600
#define accConcat 610
#define accOperator 620
#define accMinTypMax 696
#define accModPathHasIfnone 715


/*---------------- parameter values for acc configure() -- --------------*/
#define accPathnelayCount 1
#define accPathDelimStr 2
#define accDisplayErrors 3
#define accDefaultAttrO 4
#define accToHiZDelay 5
#define accEnableArqs 6
#define accDisplayWarnings 8
#define accDevelopmentVersion 11
#define accMapToMipd 17
#define accMinTypMaxDelays 19


/*--- ------- edge information used by acc_handle tchk(),etc. - --- - - --*/
#define accNoedge 0
#define accNoEdge 0
#define accEdge0l 1
#define accEdgelO 2
#define accEdgeOx 4
#define accEdgexl 8
#define accEdgelx 16
#define accEdgexO 32
#define accPosedge 13
#define accPosEdge accPosedge
#define accNegedge 50
#define accNegEdge accNegedge

/*----- ---- --------- - ------ delay models ------ ---------- -- - ----*/
#define accDelayModeNone 0
```

```
#define  accDelayModePath     1
#define  accDelayModeDistrib  2
#define  accDelayModeUnit     3
#define  accDelayModeZero     4
#define  accDelayModeMTM      5


/* ---------- values for type field in t setval delay structure --- --- - --*/
#define  accNoDelay            0
#define  accInertialDelay      1
#define  accTransportDelay     2
#define  accPureTransportDelay 3
#define  accForceFlag          4
#define  accReleaseFlag        5
#define  accAssignFlag         6
#deflne. accDeassignFlag       7


/* ------ --- values for type field in t setval_value structure --- ------ -*/
#define  accBinStrVal   1
#define  accOctStrVal   2
#define  accDecStrVal   3
#define  accHexStrVal   4
#define  accScalarVal   5
#define  accIntVal      6
#define  accRealVal     7
#define  accStringVal   8
#define  accVectorVal   10


/* ---------------   - --- ---- scalar values ---- --- - ---- --- ---- --- -*/
#define  acc0     0
#define  accl     1
#define  accX     2
#define  accZ     3


/*------------------ - - - ----VCL sCalar values --- --- ----- - ---- - -- --*/
#define      vcl0      acc0
#define      vcll      accl
#define vclX           accX
#define  vclx          vclX
#define  vclZ          accZ
#define vclz           vclZ


/*----- ----values  for vc_reason field in t_vc_record   structure -- --------*/
#define logic_value_change   1
#define strength_value change2
#define real_value_change   3
```

```
#define vector_value_change  4
#define event_value_chanqe   5
#define integer_value_change 6
#define time_value_change   7
#define  sregister_value change  8
#define vregister value change  9


/*-- -- ---------- - --- ---VCL strength values - --- -- ----- - ---- - -*/
#define  vclSupply          7
#define vclStrong           6
#define  vclPull              5
#define  vclLarge            4
#define  vclWeak              3
#define  vclMedium           2
#define vclSmall            1
#define  vclHighZ            0


/*--                 flags used with acc_vcl_add - - --- -- -- - - ----*/
#define vcl_verilog_logic    2
#define  VCL_VERILOG_LOGIC  vcl_verilog_logic
#define vcl_verilog_strength 3
#define VCL_VERILOG_STRENGTH vcl_verilog_strength


/*--                 flags used with acc_vcl_delete -- --- - - --- --- -*/
#define vcl_verilog         vcl_verilog_logic
#define     VCL_VERILOG     vcl_verilog


/*---------- values for the type field in the t acc time structure - ---- --*/
#define  accTime           1
#define accSimTime          2
#define  accRealTime                         3


/*--                        - product types ---- - - - - ----- - --- - -*/
#define accSimulator        1
#define  accTimingAnalyzer     2
#define  accFaultSimulator     3
#define accOther            4


/* -- */


/* --- --------------- global variable definitions */
/*--*/
extern  bool  acc_error_flag;
```

```
typedef int (*consumer_function)();
/*


/*-- structure definitions- */
/*--*/


/*-- ------------- data structure used with acc_set_value()-  -------*/
typedef struct t_acc_time
{
      int type;
      int low,
      high;
      double real;
} s_acc_time, *p_acc_time;


/* ------------- data structure used with acc_set_value()-*/
typedef struct t_setval_delay
{
      s_acc_time  time;
      int model;
} s_setval_delay, *p_setval_delay;


/*                    data structure of vector values-*/
typedef struct t_acc_vecval
{
      int aval;
      int bval;
}  s_acc_vecval,  *p_acc_vecval;

typedef struct t_setval_value
{
      int format;
      union
            {
           char *str;
           int  scalar;
           int integer;
           double  real;
           p_acc_vecval vector;
      } value;
}  s_setval_value,  *p_setval_value,  s_acc_value,  *p_acc_value;


/*--- --------------------structure for VCL strengths ----- - --- - --------- */
typedef struct t_strengths
{
```

```
      unsigned  char  logic_value;
      unsigned char  strength1;
      unsigned char strength2;
} s_strengths, *p_strengths;


/* ----- ------ - structure passed to callback routine for VCL -------------*/
typedef struct  t_vc_record
{
      int vc_reason;
      int vc_hightime;
      int  vc_lowtime;
      char  *user_data;
      union
      {
            unsigned char logic_value;
            double real_value;
            handle   vector_handle;
            s_strengths  strengths_s;
      } out_value;
}
s_vc_record,  *p_vc_record;


/* ------------ structure used with acc_fetch_location() routine -----------*/
typedef  struct  t_location
{
      int line_no;
      char *filename;
}  s_location,  *p_location;


/* -- - - -- structure used with acc_fetch_timescale_info() routine --------*/
typedef struct t_timescale_info
{
      short  unit;
      short  precision;
} s_timescale_info, *p_timescale_info;


/*-- -*/
/*------------ ------------ routine declarations--- ----- --------------*/
/* -- -*/
#if defined(_STDC_) || defined( _cplusplus)


#ifndef PR020_PARAMS
#define PROTO_PARAMS(params) params
#define  DEFINED_PROTO_PARAMS
#endif
```

```
#ifndef EXTERN
#define EXTERN
#define  DEFINED_EXTERN
#endif
#else
#ifndef PROTO_PARAMS
define PROTO_PARAMS(params) (/* nothing */)
#define DEFINED_PROTO_PARAMS
#endif
#ifndefEXTERN
#define EXTERN extern
#define  DEFINED_EXTERN
#endif
#endif/*STDC*/


/************* Descriptions of routines in the access routines interface
************************************/


/**** append delays into an object ****/
EXTERN bool     acc_append_delays PROTO_PARAMS((handle object,...));


/************ append  pulse  values  into  the  objects ************************/
EXTERN bool     acc_append_pulsere PROTO_PARAMS((handle object,double vallr,
double  vallx,...));
EXTERN void      *acc_close PROTO_PARAMS  (void)

EXTERN handle   acc_collect (handle (*p_next_routine,   handle scope_object, int
*aof_count));
EXTERN bool      acc_compare_handles PROTO_PARAMS((handle  h1,handle  h2));
EXTERN bool     acc_configure PROTO_PARAMS((int item,char *value));

/***********************************************/
EXTERN int       acc_count PROTO_PARAMS((handle (*next_func)(), handle object_
handle));
EXTERN int      acc_fetch_argc PROTO_PARAMS((void));
EXTERN char      **acc_fetch_argv PROTO_PARAMS((void));
EXTERN double   acc_fetch_attribute PROTO_PARAMS((handle object,...));
EXTERN int      acc_fetch_attribute_int PROTO_PARAMS((handle object,...));
EXTERN char      *acc_fetch_attribute_str PROTO_PARAMS((handle object,...));
EXTERN char      *acc_fetch_defname PROTO_PARAMS((handle object_handle));
EXTERN int     acc_fetch_delay_mode PROTO_PARAMS((handle object_p));
EXTERN bool    acc_fetch_delays PROTO_PARAMS((handle object,...));
EXTERN int       acc_fetch_direction  PROTO_PARAMS((handle  object_handle));
EXTERN int     acc_fetch_edge PROTO_PARAMS((handle acc_obj));
EXTERN char   *acc_fetch_fullname PROTO_PARAMS((handle object_handle));
;
EXTERN int    acc_fetch_fulltype PROTO_PARAMS((handle object_h));
```

```
EXTERN int      acc_fetch_index PROTO_PARAMS((handle object_handle));
EXTERN double   acc_fetch_itfarg PROTO_PARAMS((int n,handle tfinst));
EXTERN int      acc_fetch_itfarg_int PROTO_PARAMS((int n,handle tfinst));
EXTERN char     *acc_fetch_itfarg_str PROTO_PARAMS((int n,handle tfinst));
EXTERN int      acc_fetch_location PROTO_PARAMS((p_location location_p, handle
object));
EXTERN char     *acc_fetch_name PROTO_PARAMS ((handle object_handle));
EXTERN int      acc_fetch_paramtype PROTO_PARAMS((handle param_p));
EXTERN double   acc_fetch_paramval PROTO_PARAMS((handle param));
EXTERN int      acc_fetch_polarity PROTO_PARAMS((handle path));
EXTERN int      acc_fetCh_precision PROTO_PARAMS((void));
EXTERN bool     acc_fetch_pulsere PROTO_PARAMS((handle path_p,double *vallr,double
*valle,...));
EXTERN int      acc_fetch_range PROTO_PARAMS((handle node,int *msb,int *lsb));
EXTERN int      acc_fetch_size PROTO_PARAMS((handle obj_h));
EXTERN double   acc_fetch_tfarg PROTO_PARAMS((int n));
EXTERN int      acc_fetch_tfarg_int PROTO_PARAMS((int n));
EXTERN char     *acc_fetch_tfarg_str PROTO_PARAMS((int n));
EXTERN void     acc_fetch_timescale_info PROTO_PARAMS((handle obj,p_timescale_info
aof_timescale_info));
EXTERN int      acc_fetch_type PROTO_PARAMS((handle object_handle));
EXTERN char     *acc_fetch_type_str PROTO_PARAMS((int type));
EXTERN char     *acc_fetch_value PROTO_PARAMS((handle object_handle,char
*format_str,p_acc_value acc_value_p));
EXTERN void     acc_free PROTO_PARAMS((handle *array_ptr));
EXTERN handle   acc_handle_by_name PROTO_PARAMS((char *inst_name,handle
scope_p));
EXTERN handle   acc_handle_condition PROTO_PARAMS((handle obj));
EXTERN handle   acc_handle_conn PROTO_PARAMS((handle term_p));
EXTERN handle   acc_handle_datapath PROTO_PARAMS((handle path));
EXTERN handle   acc_handle_hiconn PROTO_PARAMS((handle port_ref));
EXTERN handle   acc_handle_interactive_scope PROTO_PARAMS((void));
EXTERN handle   acc_handle_itfarg PROTO_PARAMS((int n,void *suena_inst));
EXTERN handle acc_handle_loconn PROTO_PARAMS((handle port_ref));
EXTERN handle acc_handle_modpath PROTO_PARAMS((handle mod_p,char
*pathin_name, char *pathout_name, ...));
EXTERN handle acc_handle_notifier PROTO_PARAMS((handle tchk));
EXTERN handle acc_handle_object PROTO_PARAMS((char *inst_name, ...));
EXTERN handle acc_handle_parent PROTO_PARAMS((handle object_p));
EXTERN handle acc_handle_path PROTO_PARAMS((handle source,   handle destination));
EXTERN handle acc_handle_pathin PROTO_PARAMS((handle path_p));
EXTERN handle acc_handle_pathout PROTO_PARAMS((handle path_p));
EXTERN handle acc_handle_port PROTO_PARAMS((handle mod_handle,int port_num,
...));
EXTERN handle acc_handle_scope PROTO_PARAMS((handle object));
EXTERN handle acc_handle_simulated_net PROTO_PARAMS((handle net_h));
EXTERN handle acc_handle_tchk PROTO_PARAMS ((handle mod_p, int tchk_type,char
*argl_conn_name, int argl_edgetype,
..));
EXTERN handle acc_handle_tchkargl PROTO_PARAMS((handle tchk));
```

EXTERN handle acc_handle_tchkarg2 PROTO_PARAMS((handle tchk));
EXTERN handle acc_handle_terminal PROTO_PARAMS ((handle gate_handle, int terminal_index));
EXTERN handle acc_handle_tfarg PROTO_PARAMS((int n));
EXTERN handle acc_handle_tfinst PROTO_PARAMS((void));
EXTERN bool acc_initialize PROTO_PARAMS((void));
EXTERN handle acc_next PROTO_PARAMS((int *type list, handle h_scope, handle h_object));
EXTERN handle acc_next_bit PROTO_PARAMS ((handle vector, handle bit));
EXTERN handle acc_next_cell PROTO_PARAMS((handle scope, handle cell));
EXTERN handle acc_next_cell_load PROTO_PARAMS((handle net_handle, handle load));
EXTERN handle acc_next_child PROTO_PARAMS((handle mod_handle, handle child));
EXTERN handle acc_next_driver PROTO_PARAMS((handle net,handle driver));
EXTERN handle acc_next_hiConn PROTO_PARAMS((handle port,handle hiconn));
EXTERN handle acc_next_input PROTO_PARAMS((handle path,handle pathin));
EXTERN handle acc_next_load PROTO_PARAMS((handle net,handle load));
EXTERN handle acc_next_loconn PROTO_PARAMS((handle port,handle loconn));
EXTERN handle acc_next_modpath PROTO_PARAMS((handle mod_p, handle path));
EXTERN handle acc_next_net PROTO_PARAMS((handle mod_handle,handle net));
EXTERN handle acc_next_output PROTO_PARAMS((handle path,handle pathout));
EXTERN handle acc_next_parameter PROTO_PARAMS((handle module_p, handle param));
EXTERN handle acc_next_port PROTO_PARAMS((handle ref_obj_p,handle port));
EXTERN handle acc_next_portout PROTO_PARAMS((handle mod_p,handle port));
EXTERN handle acc_nextprimitive PROTO_PARAMS((handle mod_handle, handle prim));
EXTERN handle acc_next_scope PROTO_PARAMS((handle ref_scope_p, handle scope));
EXTERN handle acc_next_specparam PROTO_PARAMS((handle module_p, handle sparam));
EXTERN handle acc_next_tohk PROTO_PARAMS((handle mod_p,handle tchk));
EXTERN handle acc_next_terminal PROTO_PARAMS((handle gate_handle, handle term));
EXTERN handle acc_next_topmod PROTO_PARAMS((handle topmod));
EXTERN bool acc_object_of_type PROTO_PARAMS((handle object,int type));
EXTERN bool acc_object_in_typelist PROTO_PARAMS((handle object, int *type_list));
EXTERN int acc_product_type PROTO_PARAMS((void));
EXTERN char *acc_product_version PROTO_PARAMS((void));
EXTERN int acc_release_object PROTO_PARAMS((handle obj));
EXTERN bool acc_replace_delays PROTO_PARAMS((handle object,..));
EXTERN bool acc_replace_pulsere PROTO_PARAMS((handle object,double vallr, double vallx,....));
EXTERN void acc_reset_buffer PROTO_PARAMS((void));
EXTERN bool acc_set_interactive_scope PROTO_PARAMS((handle scope, int callback_flag));
EXTERN bool acc_set_pulsere PROTO_PARAMS((handle path,double vallr, double valle));
EXTERN char *acc_set_scope PROTO_PARAMS((handle object,...));
EXTERN int acc_set_value PROTO_PARAMS((handle obj, p_setval_value setvall, p_setval_delay delay));
EXTERN void acc_vcl_add PROTO_PARAMS((handle object_p, int (*consumer)(), char *user_data,int vcl_flags));
EXTERN void acc_vcl_delete PROTO_PARAMS((handle object, int (*consumer)(),char *user_data,int vcl_flags));

```
EXTERN char   *acc_version PROTO_PARAMS((void));
#ifdef DEFINED_PROTO_PARAMS
#undef DEFINED_PROTO_PARAMS
#undef PROTO_PARAMS
#endif


#ifdef DEFINED_EXTERN
#undef  DEFINED_EXTERN
#undef EXTERN
#endif

#define    acc_handle_calling_mod_m    acc_handle_parent((handle)tf_getinstance())


#endif /*ACC_USER H */
```

E PROGRAMMING LANGUAGE INTERFACE (PLI)

Header File – vpi_user.h file

```
/*Appendix E  Programming Language Interface - vpi_user.h file*/
/***********************************************************************
 * vpi_user.h


*


 * IEEE 1364 1995 Verilog HDL Programming Language Interface (PLI).
 *
 * This file contains the constant definitions, structure definitions,
 * and routine declarations used by the Verilog PLI VPI procedural
 * interface.
 *
 * The file should be included with all C routines that use the VPI
 *  routines.
 **********************************************************************/
#ifndef VPI_USER_H
#defineVPI_USER_H


/* basic typedefs */


typedef unsigned long  *vpiHandle;


/*Following are the constant definitions. They are divided into three
  major areas:
```

```
 1) object types
 2) access methods
 3) properties

 Note that most of the object types can also be used as access
 methods, and that some methods can also be used a properties.
*/

/*********** OBJECT TYPES **********/
#define vpiAlways      1     /* always block */
#define vpiAssign5tmt 2     /* quasi-continuous assignment */
#define vpiAssignment 3     /* procedural assignment */
#define vpiBegin       4     /* block statement */
#define vpiCase        5     /* case statement */
#define vpiCaseItem    6     /* case statement item */
#define vpiConstant    7     /* numerical constant or literal string */
#define vpiContAssign  8      /* continuous assignment */
#define vpiDeassign    9     /* deassignment statement */
#define vpiDefParam   10     /* defparam */
#define vpiDelayControl       11      /* delay statement (e.g. #10) */
#define vpiDisable   12     /* named block disable statement */
#define vpiWait      69     /* wait statement */
#define vpiWhile     70     /* while statement */
/*********** METHODS ***********/
/*********** methods used to traverse 1 to 1 relationships ***********/
#define vpiCondition 71      /* condition expression */
#define vpiDelay      72     /* net or gate delay */
#define vpiElseStmt   73     /* else statement */
#define vpiForIncStmt 74    /* increment statement in for loop */
#define vpiForInitStmt 75   /* initialization statement in for loop */
#define vpiHighConn   76      /* higher connection to port */
#define vpiLhs        77     /* left-hand side of assignment */
#define vpiIndex      78     /* index of var select, bit select, etc. */
#define vpiLeftRange  79      /* left range of vector or part select */
#define vpiLowConn    80     /* lower connection to port */
#define vpiParent     81     /* parent object */
#define vpiRhs        82   /* right-hand side of assignment */
#define vpiRightRange 83    /* right range of vector or part select */
#define vpiScope      84    !* containing scope object */
#define vpiSysTfCall 85   /* task function call */
#define vpiTchkDataTerm 86 /* timing check data term */
#define vpiTchkNotifier 87 /* timing check notifier */
#define vpiTchkRefTerm 88 /* ti.ming check reference term */


/*********** methods used to traverse 1 to many relationships ***********/
```

```
#define vpiArgument 89        /* argument to (system) task or function */
#define vpiBit 90             /* bit of vector net or port */
#define vpiDriver 91          /* driver for a net */
#define vpiInternalScope 92   /* internal scope in module */
#define vpiLoad 93            /* load on net or register */
#define vpiModDataPathIn 94   /* data terminal of a module path *,/
#define vpiModPathIn 95       /* Input terminal of a module path */
#define vpiModPathOut 96      /* output terminal of a module path */
#define vpiOperand 97         /* operand of expression */
#define vpiPortInst 98        /* connected port instance */
#define vpiProcess 99         /* process in module */
#define vpiVariables 100      /* variables in module */
#define vpiUse 101            /* usage */


/****** methods which can traverse 1-to-l, or 1-to-many relationships ******/
#define vpiExpr 102 /* connected expression */
#define vpiPrimitive 103 /* primitive (gate, switch, UDP) */
#define vpiStmt 104 /* statement in process or task */


/*********** PROPERTIES ***********/
/*********** generic object properties ***********/
#define vpiundefined -1 /* undefined property */
#define vpiType 1 /* type of object */
#define vpiName 2 /* local name of object */
#define vpiFullName 3 /* full hierarchical name */
#define vpiSize 4 /* size of gate, net, port, etc. */
#define vpiFile 5 /* File name in which the object is used */
#define vpiLineNo 6 /* File line numbez' where object is used */


/*********** modules properties **********/
#define vpiTopModule module time precision */
#define vpiDefNetType 13 /* default net type */
#define vpiUnconnDrive 14 /* unconnected port drive strength */
#define vpiHighZ 1 /* No default drive given */
#define vpiPulll 2 /* default pulll drive */
#define vpiPullO 3 /* default pul l0drive */
#define vpiDefFile 15 /* File name where the module is defined */
#define vpiDefZineNo 16 /* File line number where module is defined*/
#define vpiDefDelayMode 17 /* Delay mode of the module */
#define vpiDelayModeNone 1 /* No delay mode specified */
#define vpiDelayModePath 2 /* Path delay mode */
#define vpiDelayModeDistrib 3 /* Distributed delay mode */
#define vpiDelayModeUnit 4 /* Unit delay mode */
#define vpiDelayModeZero 5 /* Zero delay mode */
#define vpiDelayModeMTM 6 /* min:typ:max delay mode */
#define vpiDefDecayTime 18 /* Decay time for trireg net */
```

```
/*********** port and net properties ***********/
#define vpiScalar        17 /* scalar (boolean) */
#define vpiVector      18/*      vector (boolean) */
#define vpiExplicitName   19 /* port is explicitly named */
#define vpiDirection      20 /* direction of port: */
#define vpiInput     1 /* input */
#define vpi0utput    2 /* output */
#define vpiInout     3 /* inout */
#define vpiWor       3    /* wire-or net */
#define vpiTri       4    /* tri-state net */
#define vpiTriO      5    /* pull-down net */
#define vpiTril      6    /* pull-up net */
#define vpiTriReg    7    /* tri state reg net */
#define vpiTriAnd    8    /* tri-state wire-and net */
#define vpiTriOr     9    /* tri-state wire-or net */
#define vpiSupplyl   10  /* supply 1net */
#define  vpiSupply0  11   /* supply zero net */


#define  vpiExplicitScalared 23    /* explicitly scalared (boolean) */
#define  vpiExplicitVectored 24    /* explicitly vectored (boolean) */
#define vpiExpanded          25   /* expanded vector net (boolean) */
#define vpiImplicitDecl   26   /* implicitly declared net (boolean) */
#define vpiChargeStrength   27   /* charge decay strength of net */
#define vpiArray            28   /* variable array (boolean) */
#define vpiPortIndex        29   /* Port index */


/*********** gate and terminal properties ***********/
#define vpiTermIndex 30 /* Index of a primitive terminal */
#define  vpiStrength0 31 /* 0-strength of net or gate */
#define vpiStrenqthl 32 /*  1-strength of net or gate */
#define vpiPrimType 33 /* prmitive subtypes: */
#define vpiAndPrim 1 /* and gate */
#define vpiNandPrim 2 /* nand gate */
#define vpiNorPrim 3 /* nor gate */
#define vpiOrPrim 4 /* or gate */
#define vpiXorPrim 5 /* xor gate */
#define vpiXnorPrim 6 /* xnor gate *!
#define vpiBufPril /* zero-enabled not gate */
#define vpiNotiflPrim 12 /* one-enabled not gate */
#define vpiNmosPrim 13 /* nmos switch */
#define vpiPmosPrim 14 /* pmos switch */
#define vpiCmosPrim 15 /* cmos switch */
#define vpiRnmosPrim 16/* resistive nmos switch */
#define vpiRpmosPrim 17 /* resistive pmos switch *!
#define vpiRcmosPrim 18 /* resistive cmos switch */
#define vpiRtranPrim 19 /* resistive bidirectional */
#define vpiRtranifOPrim 20 /* zero-enable resistive bidirectional */
```

```
#define vpiRtraniflPrim 21 /* one-enable resistive bidirectional */
#define vpiTranPrim 22 /* bidirectional */
#define vpiTranifOPrim 23 /* zero-enabled bidirectional *J
#define vpiTraniflPrim 24 /* one-enabled bidirectional */
#define vpiPullupPrim 25 /* pullup */
#define vpiPulldownPrim 26 /* pulldown */
#define vpiSeqPrim 27 /* sequential UDP */
#define vpiCombPrim 28 /* combinational UDP */


#define vpiPolarity 34
#define vpiDataPolarity 35
#define vpiPositive 1
#define vpiNegative 2
#define vpiUnknown 3


#define vpiEdge          36 /* edge type of module path: */
#define vpiNoEdge        0x00000000 /* no edge */
#define vpiEdge0l        0x00000001 /* 0-> 1 */
#define vpiEdgelO        0x00000002 /* l-> 0*/
#define vpiEdgeOx        0x00000004 /* 0-> x */
#define vpiEdgexl        0x00000008 /* x -> 1 */
#define vpiEdgelx        0x00000010 /* l-> x */
#define vpiEdgexO        0x00000020 /* x osedge      (vpiEdgexl | vpiEdge0l | vpiEdgeOx)
#define vpiNegedge       (vpiEdgexO | vpiEdgelO | vpiEdgelx)
#define vpiAnyEdge       (vpiPosedge | vpiNegedge)
#define vpiPathType      37 /* path delay connection subtypes: */
#define vpiPathFull   1 /* (a *> b) */
#define vpiPathParallel   2 /* (a => b) */


/* timing check properties **********/
/* polarity of module path... */
/* or data path: */
/* positive */
/* negative */
/* unknown (unspecified) */


#define vpiModPathHasIfnone 38  /* state-dependent module path has ifnone
condition specified */
#define vpiTchkType      39   /* timing check subtypes: */
#define vpiSetup         1    J* Ssetup */
#define vpiHold          2    /* $hold */
#define vpiPeriod        3    /* $period */
#define vpiWidth         4    /* $width */
#define vpiSkew          5    /* $skew */
#define vpiRecovery      6    /* $recovery */
#define vpiNoChange      7    /* $nochange */
```

```
#define vpiSetupHold      8    /* $setuphold */
/********** expression propert **********/
#define vpiOpType         40   /* operation subtypes: */
#define vpiMinusOp        1    /* unary minus */
#define vpiPlusOp         2    /* unary plus */
#define vpiNotOp          3    /* unary not */
#define vpiBitNegOp       4    /* bitwise negation */
#define vpiUnaryAndOp     5    /* bitwise reduction and */
#define vpiUnaryNandOp    6    /* bitwise reduction nand */
#define vpiUnaryOrOp      7    /* bitwise reduction or */
#define vpiUnaryNorOp     8    /* bitwise reduction nor */
#define vpiUnaryXorOp     9    /* bitwise reduction xor */
#define vpillnaryXNorOp   10   /* bitwise reduction xnor */
#define vpiSubOp          11   /* binary subtraction */
#define vpiDivOp          12   /* binary division */
#define vpiModOp          13   /* binary modulus */
#define vpiEqOp           14   /* binary equality */
#define vpiNeqOp          15   /* binary inequality */
#define vpiCaseEqOp       16   /* case (x and z) equality */
#define vpiCaseNeqOp      17   /* case inequality */
#define vpiGtOp           18   /* binary greater than */
#define vpiGeOp           19   /* binary greater than or equal */
#define vpiLtOp           20   /* binary less than */
#define vpiLeOp           21   /* binary less than or equal */
#define vpiLShiftOp       22   /* binary left shift */
#define vpiRShiftOp       23   /* binary right shift */
#define vpiAddOp          24   /* binary addition */
#define vpiMultOp         25   /* binary multiplication */
#define vpiLogAndOp       26   /* binary logical and */
#define vpiLogOrOp        27   /* binary logical or */
#define vpiBitAndOp       28   /* binary bitwise and */
#define vpiBitOrOp        29   /* binary bitwise or */
#define vpiBitXorOp       30   /* binary bitwise xor */
#define vpiBitXNorOp      31   /* binary bitwise xnor */
#define vpiConditionOp    32   /* ternary conditional */
#define vpiConcatOp       33   /* n-ary concatenation */
#define vpiMultiConcatOp  34   /* repeated concatenation */
#define vpiEventOrOp      35   /* event or */
#define vpiNullOp         36   /* null operation */
#define vpiListOp         37   /* list of expressions */
#define vpiMinTypMaxOp    38   /* min:typ:max: delay expression */
#define vpiPosedgeOp      39   /* posedge */
#define vpiNegedgeOp      40   /* negedge */


#define vpiConst2ype      41   /* constant subtypes: */
#define vpiDecConst       1    /* decimal integer */
#define vpiRealConst      2    /* real */
#define vpiBinaryConst    3    /* binary integer */
#define vpiOctConst       4    /* octal integer */
```

```
#define vpifiexConst       5   /* hexadecimal integer */
#define vpiStringConst      6   /* string literal */


#define vpiBlocking       42  /* blocking assignment (boolean) */
#define  vprties   ******************/
#define vpiSysFuncType   45  /* system function type */
#define vpiSysFuncInt    1   /* returns integer */
#define vpiSysFuncReal   2   /* returns real */
#define vpiSysFuncTime    3   /* returns time */
#define vpiSysFuncSized  4   /* returns sized */
#define vpiUserDefn     46  /* user defined system tf_[boolean) */


#define vpiScheduled  47 /* is object vpiSchedEvent still scheduled */
/************ I/0 related defines ***************************/
#define  VPI_MCD_STDOUT  0x00000001
#define  VPI_MCD_STDERR 0x00000002
#define  VPI_MCD_LOG 0x00000004


/*********************** STRUCTURE DEFINITIONS *************************/
/************************** time structure ********************************/
typedef struct t_vpi_time   {
      int type;             /* [vpiScaledRealTime, vpiSimTime,vpiSuppressTime]*/
      unsigned int high,low;   /* for vpiSimTime */
      double real;          /* for vpiScaledRealTime */
} s_vpi_time,*p_vpi_time;


/* time types */
#define vpiScaledRealTime 1
#define vpiSimTime 2
#define vpiSuppressTime 3


/************************* delay structures *****************************/
typedef struct t_vpi_delay
{
      struct t_vpi_time *da; /* ptr to user allocated array of delay values */
      int no_of_delays; /* number of delays */
      int  time_type;  /*   [vpiScaledRealTime,vpiSimTime,vpiSuppressTime]*/
      int mtm_flag; /* true for mtm values */
      int append_flag; /* true for append */
      int pulsere_flag; /* true for pulsere values */
} s_vpi_delay, *p_vpi_delay;
```

```
/************************* value structures ***************************/
/* vector value */
typedef struct t_vpi_vecval
{
    /* following fields are repeated enough times to contain vector */
    int aval, bval; /* bit encoding: ab: 00=0,10=1,11=X, O1=Z */
} s_vpi_vecval, *p_vpi_vecval;


/* strength (scalar) value */
typedef struct t_vpi_strengthval
{
    int logic; /* vpi [0,1, X, Z] */
    int s0, si; /* refer to strength coding below */
} s_vpi_strengthval, *p_vpi_strengthval;


/* strength values */
#define vpiSupplyDrive   0x80
#define vpiStrongDrive   0x40
#define vpiPullDrive     0x20
#define vpiWeakDrive     0x08
#define vpiLargeCharge   0x10
#define vpiMediumCharge  0x04
#define vpiSmallCharge   0x02
#define vpiHiZ           0x01


/* generic value */
typedef struct t_vpi_value
{
    int format; /* vpi[[Bin,Oct,Dec,HexJStr,Scalar,Int,Real,String,Vector,
                   Strength,Suppress,Time,ObjType]Val*/
    union
    {
        char *str;
        int scalar;
        int integer;
        double real;
        struct t_vpi_time *time;
        struct t_vpi_vecval *vector;
        struct t_vpi_strengthval *strength;
        char *misc;
    } value;
}
s_vpi_value, *p_vpi_value;


/* value formats */
#define vpiBinStrVal     1
```

```
#define vpiOctStrVal     2
#define vpiDecStrVal     3
#define vpiHexStrVal     4
#define vpiScalarVal     5
#define vpiIntVal        6
#define vpiRealVal       7
#define vpiStringVal     8
#define vpiVectorVal     9
#define vpiStrengthVal   10
#define vpiTimeVal       11


/* string value */
/* vpi[0,l,X,Z) */
/* integer value */
/* real value */
/* time value */
/* vector value */
/* strength value */
/*...other */


#define vpiObjTypeVal 12
#define vpiSuppressVal 13


/* delay modes */
#define vpiNoDelay 1
#define vpiInertialDelay 2
#define vpiTransportDelay 3
#define vpiPureTransportDelay 4
/* force and release flags */
#define vpiForceFlag 5
#define vpiReleaseFlag 6
/* scheduled event cancle flag */
#define vpiCancelEvent 7


/* bit mask for the flags argument to vpi_put_value() */
#define vpiReturnEvent 0x1000


/* scalar values */
#define vpi0
#define vpil
#define vpiZ
#define vpiX
#define vpiH
#define vpiL
#define vpiDontCare
```

```
/*
#define vpiNoChange
here.
*/


/********************  system  taskfunc  structure  ***********************/
typedef struct t_vpi_systf_data
{
      int type; /* vpiSysTask, vpiSysFunc */
      int subtype; /* vpiSys(Task, Func(Int,Real,Time,Sized]] */
      char *tfname; /* first character must be `$' */
      int (*calltf) ();
      int (*compiletf) ();
      int (*sizetf)(); /* for vpiSysFuncSized callbacks only */
      char *user_data;
} s_vpi_systf_data, *p_vpi_systf_data;


#define vpiSysTask 1
#define vpiSysFunc 2
/* the subtypes are defined under the vpiSysFuncType property */
/****************   Verilog execution information structure ***************/
typedef struct t_vpi_vlog_info
{
      int argc;
      char **argv;
      char *product;
      char *version;
} s_vpi_vlog_info, *p_vpi_vlog_info;


/****************  PLI  error  information structure ****************/
typedef struct t_vpi_error_info
{
      int state;         /* vpi[Compile,PLI,Run] */
      int level;          /* vpi[Notice,Warning,Error,System,Internal]*/
      char *message;
      char *product;
      char *code;
      char *file;
      int line;
} s_vpi_error_info,*p_vpi_error_info;
/* error types */
#define vpiCompile  1
#define vpiPLI      2
#define vpiRun      3


#define vpiNotice   1
```

```
#define vpiWarning   2
#define vpiError     3
#define vpiSystem     4
#define  vpiInternal  5


/*********************** callback  structures  ***************************/
/* normal callback structure */
typedef struct t_cb_data
{
      int reason;               /* callback reason */
      int (*cb_rtn)();          /* call routine */
      vpiHandle obj;            /* trigger object */
      p_vpi_time *time;         /* callback time */
      p_vpi_value *value;       /* trigger object value */
      int index;               /* index of the memory word or var select
                                  which changed value */
      char *user_data;
 } s_cb_data, *p_cb_data;


/* Callback Reasons */
/* Simulation-related */
#define cbValueChange   1
#define cbStmt          2
#define cbForce         3
#define cbRelease        4
/* Time-related */
#define cbAtStartOfSimTime  5
#define cbReadWriteSynch    6
#define cbReadOnlySynch      7
#define cbNextSimTime      8
#define cbAfterDelay        9
/*  Action-related */
#define cbEndOfCompile       10
#define cbStartOfSimulation   11
#define cbEndOfSimulation     12
#define cbError            13
#define cbTchkViolation     14
#define cbStartOfSave        15
#define  cbEndOfSave         16
#define cbStartOfRestart      17
#define cbEndOfRestart       18
#define  cbStartOfReset       19
#define cbEndOfReset         20
#define  cbEnterInteractive  21
#define  cbExitInteractive  22
#define cbInteractiveScopeChange 23
#define cbUnresolvedSystf   24
```

```
#if defined(_STDC_) || defined( _cplusplus)

#ifndef  PROTO_PARAMS
#define  PROTO_PARAMS(params) params
#define  DEFINED_PROTO_PARAMS
#endif
#ifndef EXTERN
#define EXTERN
#define DEFINED_EXTERN
#endif

#else
#ifndef PROTO_PARAMS
#define PROTO_PARAMS(params) (/* nothing */)
#define DEFINED_PROTO_PARAMS
#endif
#ifndef EXTERN
#define EXTERN extern
#define DEFINED_EXTERN
#endif

/** Following is a list of routines from the VPI procedural interface **/

extern vpi_handle( vpiObjectType otype,  vpiHandle object);

extern vpi_handle_multi();

/*** register a system task/function ***/
extern vpi_register_systf();

/** Get information about a system task/function callback **/
extern vpi_get_systf_info();

/*** Obtain a handle by name ***/
extern vpi_handle_by_name(char *name, vpiHandle object);

extern vpi_handle_by_index();
```

```
extern vpi_register_cb();
extern vpi_remove_cb();
extern vpi_get_cb_info();


/*****************************************************************/
/*** obtain a logic or strength value of an object ***/
p_vpi_value vpi_get_value(vpiHandle object);
vpi_put_value(vpiHandle object, p_vpi_value value);


/************* write delays or timing limits to an object *******/
vpi_put_delays();


/********** obtain a handle to an object with one to many relationship and then iterate ***/
vpiHandle vpi_iterate(vpiObjectType otype, vpiHandle object);
vpiHandle vpi_scan (vpiHandle object);


/******* miscellaneous vpi routines *************************/
/*** find the current simulation time or the scheduled time for future events **/
vpi_get_time();


/******* write to stdout and the current log file ******/
vpi_printf();


/********** open a file for writing ***************/
vpi_mcd_open();


/********* close one or more files ***************/
vpi_mcd_printf();


/************* retrieve the name of the open file *****/
vpi_mcd_name();


/************* retrieve data about product invocation options ***/
vpi_get_vlog_info();


/********** see if two handles refer to the same object ******/
vpi_compare_objects();

vpi_chk_error();
```

```
/**** obtain error status and error information about the previous calls to a VPI routine ****/
vpi_free_object();


/*** Get the values of objects of types integer or boolean **/
int vpi_get();


/************ get the values of objects of string type ****/
char *vpi_get_str();


#endif /* STDC */


/* utility routines */
EXTERN int vpi_compare_objects PROTO_PARAMS((vpiHandle objectl, vpiHandle
object2));
EXTERN int vpi_chk_error PROTO_PARAMS((p_vpi_error_info error_info_p));
EXTERN int vpi_free_object PROTO_PARAMS((vpiHandle object));
EXTERN int vpi_get_vlog_info PROTO_PARAMS((p_vpi_vlog_info vlog_info_p));


#ifdef DEFINED_PROTO_PARAMS
#undef DEFINED_PROTO_PARAMS
#undef PROTO_PARAMS
#endif


#ifdef DEFINED_EXTERN
#undef DEFINED_EXTERN
#undef EXTERN
#endif


/*********************** GLOBAL VARIABLES ********************************/
extern void (*vlog_startup_routines[])(); /* array of function pointers, */
/* last pointer should be null */
#endif /*VPI_USER_H */
```

F FORMAL SYNTAX DEFINITION OF SDF

delay_file ::= (DELAYFILE sdf_header cell {, cell})

sdf_header ::= sdf_version [design_name][date] [vendor] [program_name] [program_version] [hierarchy_divider] [voltage] [process] [temperature] [time_scale]

sdf_version ::= (SDFVERSION QSTRING)

design_name ::= (DESIGN QSTRING)

date ::= (DATE QSTRING)

vendor ::= (VENDOR QSTRING)

program_name ::= (PROGRAM QSTRING)

program_version ::= (VERSION QSTRING)

hierarchy_divider ::= (DIVIDER HCHAR)

HCHAR ::= .// a period character | / // a slash character

voltage ::= (VOLTAGE rtriple) | (VOLTAGE RNUMBER)

process ::= (PROCESS QSTRING)

temperature ::= (TEMPERATURE rtriple) | (TEMPERATURE RNUMBER)

time_scale ::= (TIMESCALE TSVALUE)

cell ::= (CELL celltype cell_instance {timing_spec})

celltype ::= (CELLTYPE QSTRING)

cell_instance ::= (INSTANCE [PATH])
| (INSTANCE WILDCARD)

WILDCARD ::= * // the asterisk character

timing_spec ::= del_spec
| tc_sp del_defec | te_spec

del_spec ::= (DELAY deltype {, deltype})

tc_spec ::= (TIMINGCHECK tchk_def {, tchk_def})

te_spec ::= (TIMINGENV te_def {, te_def})

deltype ::= (PATHPULSE [input_output_path] value [value])
| (PATHPULSEPERCENT [input_output_path] value [value])
| (ABSOLUTE deLdef {,del_def})
| (INCREMENT del_def {, del_def})

input_output_path ::= {port_instance}

del_def ::= (IOPATH port_spec port_instance (RETAIN { delval_list})
|(COND [QSTRING] conditional_port_expr(IOPATH port_spec port_instance(RETAIN { delvaljist}))
|(CONDELSE (IOPATH port_spec portjnstance (RETAIN {delvaljist}))
| (PORT port_instance delvaljist)
| (INTERCONNECT port_instance portjnstance delvaljist)
| (DEVICE [port_instance] delvaljist)

tchk_def ::= (SETUP port_tchk port_tchk value)
| (HOLD port_tchk port_tchk value) I (SETUPHOLD port_tchk port_tchk rvalue rvalue)
| (SETUPHOLD port_spec port_spec rvalue [rvalue scond] [ccond])
| (RECOVERY port_tchk port_tchk value)
| (REMOVAL port_tchk port_tchk value)
| (RECREM port_tchk port_tchk rvalue rvalue)
| (RECREM port_spec port_spec rvalue rvalue [scond] [ccond])
| (SKEW port_tchk port_tchk rvalue) I (WIDTH port_tchk value)
| (PERIOD port_tchk value) I (NOCHANGE port_tchk port_tchk rvalue rvalue)

port_tchk ::= port_spec I (COND [QSTRING] timing_check_condition port_spec)

scond ::= (SCOND [QSTRING] timing_check_condition)

ccond ::= (CCOND [QSTRING] timing_check_condition)

name ::= (NAME [QSTRING])

exception::= (EXCEPTION cell_instance {, cell_instance})

te_def ::= cns_def | tenv_def

cns_def::= (PATHCONSTRAINT [name] port_instance {,port_instance} rvalue rvalue)
| (PERIODCONSTRAINT port_instance value [exception])
| (SUM constraint_path constraint_path {, constraint_path} rvalue [rvalue]) | (DIFF constraint_path
constraint_path value [value])
| (SKEWCONSTRAINT port_spec value)

tenv_def ::= (ARRIVAL [port_edge] port_instance rvalue rvalue rvalue rvalue)
| (DEPARTURE [port_edge] port_instance rvalue rvalue rvalue rvalue)
| (SLACK port_instance rvalue rvalue rvalue rvalue [NUMBER])
| (WAVEFORM port_instance NUMBER edge_list)

constraint_path ::= (port_instance port_instance)

port_spec ::= port_instance | port_edge

port_edge ::= (EDGE_IDENTIFIER port_instance)

EDGE_IDENTIFIER ::= posedge | negedge | 01 |10 | 0z | z1 | 1z | z0

port_instance ::= port | PATH HCHAR port

port ::= scalar_port | bus_port

scalar_port ::= IDENTIFIER | IDENTIFIER [DNUMBER]

bus_port ::= IDENTIFIER [DNUMBER : DNUMBER]

edge_list ::= pos_pair {, pos_pair} | neg_pair {, neg_pair}

pos_pair ::= (posedge RNUMBER [RNUMBER]) (negedge RNUMBER [RNUMBER])

neg_pair ::= (negedge RNUMBER [RNUMBER]) (posedge RNUMBER [RNUMBER])

value ::= ([NUMBER]) | ([triple])

triple ::= NUMBER : [NUMBER] : [NUMBER] | [NUMBER] : NUMBER : [NUMBER]
| [NUMBER]: [NUMBER] : NUMBER

rvalue ::= ([RNUMBER]) | ([rtriple])

rtriple ::= RNUMBER :[RNUMBER]:[RNUMBER]
| [RNUMBER]: RNUMBER [:RNUMBER]
| [RNUMBER]: [RNUMBER]: RNUMBER

Apart from allowing negative numbers (RNUMBER instead of NUMBER), rvalue and rtriple are essentially the same as value and triple.

delval ::= rvalue | (rvalue rvalue) | (rvalue rvalue rvalue)

delval_list ::= delval | delval
delval | delval delval delval | delval delval delval delval [[delval]
delval] | delval delval delval delval delval delval delval [[delval]
delval] [delval] [delval] [delval]
conditional_port_expr ::= simple_expression
| (conditional_port_expr)
| UNARY_OPERATOR (conditional_port_expr)
| conditional_port_expr BINARY_OPERATOR conditional_port_expr

simple_expression ::= (simple_expression)
| UNARY_OPERATOR (simple_expression) | port | UNARY_OPERATOR
port | SCALAR_CONSTANT | UNARY_OPERATOR SCALAR_CONSTANT |
simple_expression
QM simple_expression CLN simple_expression | { [simple_expression
concat_expression] } | { simple_expression { simple_expression [concat_expression]
} }

concat_expression ::=, simple_expression

QM ::= ? // a literal question mark

CLN ::=:// a literal colon

Timing check conditional expressions are defined as follows:

timing_check_condition ::= scalar_node
| INVERSION_OPERATOR scalar_node
| scalar_node EQUALITY_OPERATOR SCALAR_CONSTANT

scalar_node ::= scalar_port scalar_net

scalar_net ::= IDENTIFIER

SCALAR_CONSTANT
::= 1`b0//logical zero
| 1`bl //logicalone
| 1` B0//logical zero
| 1`B1 // logical one
|`b0//logical zero
|`bl // logicalone
| *`B0 //* logical zero
|`B1 // logical one
| 0 // logical zero

| 1 // logical one

UNARY_OPERATOR ::= +//arithmetic identity
| - // arithmetic negation
| ! // logical negation | ~ // bit-wise unary negation | & // reduction unary
AND | ~& // reduction unary NAND | | // reduction unary OR |
~| // reduction unary NOR | ^ // reduction unary XOR | ^~ // reduction
unary XNOR | ~^ // reduction unary XNOR (alternative)

INVERSION_OPERATOR
::= ! // logical negation | ~ // bit-wise unary negation

BINARY_OPERATOR
::= + // arithmetic sum | - // arithmetic difference | * // arithmetic
product | / // arithmetic quotient | % // modulus | == // logical
equality | != // logical inequality | === // case
equality |
!== // case inequality | && // logical AND | || // logical OR
| < // relational | <= // relational | > // relational | >=
// relational | & // bit-wise binary AND | | // bit-wise binary
inclusive OR | ^ // bit-wise binary exclusive OR | ^~ // bit-wise
binary equivalence | ~^ // bit-wise binary equivalence (alternative)
| >>// right shift | << // left shift

EQUALITY_OPERATOR ::=
== // logical equality | != // logical inequality | === // case
equality | !== // case inequality

G LIST OF EXAMPLES

Example 1-1. A gate-level description of edge-sensitive d flip-flop. 2
Example 1-2. A 4-bit counter built using instances of d flip-flop defined in Figure 1 -1.....3
Example 1-3. Schematics for dff in 1-1. 3
Example 1-4. Schematic for counter in 1-2...3
Example 1-5. Behavioral description of the same counter as in 1-2. 4
Example 1-6. A test module for testing the two descriptions of counter and their equivalence..5
Example 1-7. Waveforms for the counter example. 6
Example 1-8. Factorial generation of a number. 7
Example 1-9. A system model with microprocessor, ram, and cache controller. 9
Example 1-10. Behavioral description of a cache controller with write-through scheme.... 13

Example 2-1. Data declarations. 23
Example 2-2. Reg declarations. 24
Example 2-3. Wire declarations. 25
Example 2-4. Wand (wired-AND) declarations and usage. 25
Example 2-5. wor (wired-OR) declarations and usage. 26
Example 2-6. Tri (Three-State) declarations and usage. 27
Example 2-7. Nets of tri types declared and used for multiple drivers with tri-state resolution...27
Example 2-8. supply0 and supply1 constructs. 27
Example 2-9. Trireg net and the switch-level modeling example. 28
Example 2-10. Net type declarations...29
Example 2-11. Port type declarations. 30
Example 2-12. Aggregate declarations. 31
Example 2-13. Vectored and scalared bit-vector net declarations. 31
Example 2-14. Memory declarations. 31
Example 2-15. Net declarations with delay specifications. 32
Example 2-16. Integer and time declarations...33
Example 2-17. Real declarations 33

Example 2-18 Parameter declaration examples. 34
Example 2-19. Hierarchical names. 35

Example 3-1. Levels of abstractions....38
Example 3-2. Behavioral level of abstraction – adder....39
Example 3-3. Two equivalent continuous assignments. 41
Example 3-4. RTL abstractions – adder. 41
Example 3-5. RTL abstractions – adder with boolean optimizations. 42
Example 3-6. Continuous assignments – RTL modeling....43
Example 3-7. Addition operation. 46
Example 3-8. Relational operators. 46
Example 3-9. Equality operator. 47
Example 3-10. Logical operators. 47
Example 3-11. Bit-wise operators....48
Example 3-12. Reduction operators. 48
Example 3-13. Shift operator....49
Example 3-14. Conditional operator. 49
Example 3-15. Nested conditional operator....50
Example 3-16. Concatenation operator. 50
Example 3-17. Concatenation equivalent. 50
Example 3-18. Different representations of constant 10 in Verilog....52
Example 3-19. Wire operands....52
Example 3-20. Bit-select operands....52
Example 3-21. Part-select operands. 53
Example 3-22. Function call used as an operand. 53
Example 3-23. Concatenation of operands. 53
Example 3-24. Constant valued expressions. 55
Example 3-25. Examples of operator usage....56
Example 3-26. Different ways to perform sized operations....57
Example 3-27. Datapath design using continuous assignments RTL abstractions....62
Example 3-28. Structural design – A CPU with details of adder at the gate-level. 63
Example 3-29. Example of parametrized module definitions. 65
Example 3-30. Example of a macromodule construct. 65
Example 3-31. Named ports in modules....66
Example 3-32. Module instantiations....68
Example 3-33. Module definitions and instantiation – hierarchical design example....68
Example 3-34. A structural model of R4200 processor with declarations and instances at top-level....71
Example 3-35. A structural model of UltraSPARC-IIi. 81

Example 4-1. A sample design with structure and behavior....86
Example 4-2. Log of a typical simulator with tracing. ....
Example 4-3. Ideal simulation log for a sample circuit with tracing....90
Example 4-4. Multiple events on a reg – but no cancellation (algorithm 4-7 applied) 96
Example 4-5. Multiple events on a reg resulting cancellation (algorithm 4-7 applied) 97

Example 5-1. Behavioral clock generation. 100
Example 5-2. Blocking assignments – inter-assignment delays....101

Example 5-3. Blocking Assignment – intra-assignment delays. 102
Example 5-4. Blocking assignments – interassignment delays. 103
Example 5-5. Blocking assignments – intra-assignment delays 104
Example 5-6. Blocking and non-blocking comparison – exchange of values for blocking 105
Example 5-7. Blocking and non-blocking comparison – no exchange of values for blocking 105
Example 5-8. Blocking assignments – multiple schedules. 106
Example 5-9. Blocking assignments – no multiple schedules. 106
Example 5-10. If statement example. 107
Example 5-11. Nested if statement. 107
Example 5-12. If-else-if mutually exclusive multiple conditions 108
Example 5-13. Case statement – a multiplexor model. 108
Example 5-14. Case statement with unknowns and tri-states 109
Example 5-15. Casex statement with unknowns and tri-states 110
Example 5-16. Case statement with unknowns and tri-states 111
Example 5-17. Loops – for statement usage. 111
Example 5-18. Loops – while statement usage. 112
Example 5-19. Loops – repeat statement usage. 112
Example 5-20. Loops – forever statement usage 112
Example 5-21. Sequential blocks – begin-end usage. 113
Example 5-22. Wait statement – synchronizing two processes 113
Example 5-23. Event declarations and usage 114
Example 5-24. Event declarations 114
Example 5-25. Multi-event event 115
Example 5-26. Event triggering and event based synchronization. 115
Example 5-27. Fork-join statements – modeling asynchronous reset and instruction loop concurrency in microprocessors. 118
Example 5-28. Modeling instructions with no parallelism. 118
Example 5-29. Modeling instructions with parallelism. 119
Example 5-30. Modeling instructions with pipeline – fork-join usage. 119
Example 5-31. Function definition – a mux. 120
Example 5-32. Function call – creating a multiplexor module with function mux. 120
Example 5-33. Task declaration and usage – a shift register 121
Example 5-34. Task disabling – reset modeling for a microprocessor. 123
Example 5-35. Aassign – deassign – a flip-flop model with quasi-continuous assignments. 124
Example 5-36. Force-release statements – debugging the flip-flop model. 125
Example 5-37. A behavioral processor model. 132

Example 6-1. Built-in gates modeling. 136
Example 6-2. A user-defined primitive definition. 139
Example 6-3. A combinational user–defined primitive: An adder part – carry computation. 142
Example 6-4. Level sensitive sequential UDP – a latch. 143
Example 6-5. Mixed edge and level sensitive sequential UDP – SR flip-flop with clear 145
Example 6-6. User-defined primitives – instances 146

Example 7-1. Comparisons of different levels of abstraction by mixed level design...... 152
Example 7-2. System modeling with behavioral, rtl, gates and switches mixed in one board design. 153
Example 7-3. A behavioral flag-bit generation mixed with adder of rtl or structural style. 153

Example 8-1. $display usage with different methods of capturing design data. 156
Example 8-2. Capturing simulation results of a design selectively with $monitor and $monitoron/off. 159
Example 8-3. File management in capturing results of simulating a design. 161
Example 8-4. Reading input from files – $readmem usage. 162
Example 8-5. Simulation control tasks – $stop and $finish. 163
Example 8-6. Creating a waveform data file using $dumpvars and related system tasks. 164

Example 9-1. `include compiler directive. 168
Example 9-2. Compiling code conditionally based on prior macro definitions using `ifdef. 170
Example 9-3. default_nettype compiler directive. 170

Example 11-1. 8085 microprocessor, ram, and 8251 serial IO controller system – behavioral model. 183
Example 11-2. R4200 microprocessor with instructions, bus-cycles, and registers – behavioral model. 215
Example 11-3. A cache system with a write-through policy; behavioral abstraction. 222
Example 11-4. Cache system with a write-back policy: behavioral model – refinement of Example 11-3. 229
Example 11-5. A register transfer level model of the cache system with write-through policy: with blocks. 236
Example 11-6. Detailed model of a cache controller with write-through policy 242

Example 12-1. Synthesizable combinational adder. 249
Example 12-2. Synthesizable combinational multiplexor. 249
Example 12-3. Synthesizable sequential design – traffic light controller. 250
Example 12-4. Synthesizable parts of the cache system – cache control (sequential), mux, compare. 253

Example 13-1. Parametrized design. 259
Example 13-2. Instantiating a parameterized design in your Verilog code. 259
Example 13-3. Gate-level instantiations. 260
Example 13-4. Three-state gate instantiation. 261
Example 13-5. Synthesis and comparisons to X. 262
Example 13-6. Synthesizable combinational 4-bit adder using functions and continuous assignments. 262
Example 13-7. Synthesizable combinational multiplexor. 263
Example 13-8. A behavioral description with feedback that is not real. 264
Example 13-9. Memory declarations – synthesized as bank of registers. 264
Example 13-10. Accessing bits within a memory block. 265
Example 13-11. Parameter declaration in a function. 265
Example 13-12. if statement that synthesizes multiplexor logic. 267

Example 13-13. if...else if...else structure with several mutually exclusive conditions......267
Example 13-14. Nested if and else statements. 268
Example 13-15. Synthesizing a latch for a conditionally-driven variable. 268
Example 13-16. A case statement that is both full and parallel......269
Example 13-17. A non-full but parallel case statement – synthesized with latches......269
Example 13-18. A case statement that is neither parallel, nor full. 270
Example 13-19. Example of synthesizable simple for loop. 270
Example 13-20. Example of nested for loops that can be synthesized. 271
Example 13-21. For loop used for duplicating statements. 271
Example 13-22. Equivalent expansion of for loop in Example 13-20. 271
Example 13-23. Unsupported while loop. 271
Example 13-24. Supported while loops......272
Example 13-25. Supported forever loop. 272
Example 13-26. Loop exiting using disable – a comparator model. 273
Example 13-27. Synchronous reset of state register using disable in a forever loop. 273
Example 13-28. Using tasks to describe synthesizable combinational logic. 274
Example 13-29. A simple always block describing synthesizable combinational logic. 274
Example 13-30. Event expression indicating sensitivity or input to a block. 275
Example 13-31. Event expression indicating rising edge of clock. 275
Example 13-32. Falling clock edge modeled in event-expression. 275
Example 13-33. Event expression modeling clock-edge combined with resetting condition. 275
Example 13-34. Incomplete event list – simulation and synthesis may mismatch......276
Example 13-35. Complete event list. 276
Example 13-36. Incomplete event list for edge-triggered flip-flop with asynchronous reset. 276

Example 14-1. Creating a latch......280
Example 14-2. Creating a latch with a case statement. 280
Example 14-3. Variable declared within a function-no latches inferred. 281
Example 14-4. Creating an edge-triggered D flip-flop. 281
Example 14-5. Flip-flop with asynchronous reset......282
Example 14-6. Flip-flop with synchronous reset......283
Example 14-7. Asynchronous set/reset on a design. 284
Example 14-8. Synchronous set/reset on a design. 284
Example 14-9. Using one_hot for set and reset. 285
Example 14-10. Using one_cold for set and reset......286
Example 14-11. Creation of a bus-latch. 286
Example 14-12. An 8-1 multiplexer modeled behaviorally and synthesized to a mux. 287
Example 14-13. A 3-state gate created behaviorally......288
Example 14-14. A 3-state gate with 2 drivers. 288
Example 14-15. Creation of two 3-state gates with independent controls......288
Example 14-16. Three-state with registered enable......289
Example 14-17. Three-state without registered enable. 289
Example 14-18. Sharing of adder resources. 291
Example 14-19. Shared resources have independent paths. 292
Example 14-20. Case statement sharing......292
Example 14-21. Sharing may result in feedback loop. 293

Example 15-1. Timing checks and path delay specifications in specify blocks. 296
Example 15-2. Specify block example – timing checks on module pins. 300
Example 15-3. Specify blocks – simple paths from input to output. 300
Example 15-4. Edge-sensitive paths with sensitivity to positive clock edges. 301
Example 15-5. State dependent path delay specifications: example statements. 302
Example 15-6. State dependent path delay specifications – a full module. 302
Example 15-7. Edge sensitive state dependent path delays. 303
Example 15-8. Multiple edge and state delay specifications for the same simple path. 303
Example 15-9. Illegal state dependent delay specification. 303
Example 15-10. ifnone statements in a state dependent specification. 303
Example 15-11. ifnone statement – opcode dependent delays for execution unit. 304
Example 15-12. Delay specifications – one delay for all cases and a min-typ-max specification 304
Example 15-13. Two, three, six, and twelve different delays for the same path. 305

Example 17-1. Unidirectional transistors: a dynamic mos serial shift register. 317
Example 17-2. Bidirectional transistors and modeling with strengths – a static ram cell. 319

Example 19-1. Analog resistor described in Verilog's analog extensions. 366
Example 19-2. Electrical type declarations in analog or mixed signal modules. 366
Example 19-3. Example of analog block. 367
Example 19-4. Mixed signal design with Verilog MS. 369

H REFERENCES

1. The IEEE 1364-1995 Verilog Language Reference Manual, IEEE Standards Press, 1995.
2. Digital Design with the Verilog HDL -by Vivek Sagdeo, Class Notes – UC Berkeley Ext, 1996
3. Designing with Verilog – Class Notes by Vivek Sagdeo, PerformanCAE Corporation Training
4. SDF 3.0 Reference Manual, Open Verilog International, May 1995
5. Semantic Model for Verilog HDL by Vivek Sagdeo PerformanCAE Corporation - Internal Papers
6. Analog Design with Verilog-A –I Miller et al, IVC 97 Proceedings, 1997.
7. Verilog-A Language Reference Manual, OVI1996
8. Verilog Support for Cycle-Based Simulation – Speedsim Inc., 1996
9. Open Model Interface Draft Standard V1.0, CFI – http://www.cfi.org/OMF/, 1997
10. DCL Proposal, CFI 1997
11. R4200 Microprocessor Reference Guide, MIPS Technologies Inc, 1995
12. comp.lang.verilog newsgroup archives – 1997
13. http://www.veri-log.com PerformanCAE Corporation Web Site
14. Synthesis Subsets-Descriptions from Synopsys, Synplicity, Exemplar Logic, and Silicon Automation Systems
15. OVI Directory for Verilog, 1996
16. DAC Proceedings –1986-1996

INDEX

-,44,51

!,44
!=,44,51

$display, 156
$dumpfile, 163
$dumpoff, 163
$dumpon, 163
$dumpvars, 163
$fclose, 160, 161
$fdisplay, 161
$finish, 162
$fmonitor, 161
$fopen, 160
$fstrobe, 160, 161
$fwrite, 161
$gr_waves, 174
$hold, 297
$keys, 174
$log, 174
$monitor, 158
$period, 298
$readmemb, 161
$readmemh, 162
$recovery, 298
$setup, 297
$setuphold, 299
$showvars, 174
$skew, 298
$stop, 162
$time, 162
$width, 297

%, 44,51
%b, 157
%d, 157
%e, 157
%g, 157
%m, 157
%o, 157
%s, 157
%t, 157
%v, 157

&, 44, 51
&&, 44, 51

*, 44

/, 44, 51

?, 51
?:, 45
?:, 51

^, 44, 45, 51
^~, 44, 51

`default_nettype, 170
`define, 168
`else, 169
`endif, 170
`ifdef, 170
`include, 168
`resetall, 171
`timescale, 171
`undef, 169

{ }, 44
{},51

|, 44, 45, 51
||, 51

~ 44
~&, 45
~^, 45
-|, 45

+, 44, 51

<, 44, 51
<<, 45, 51
<=, 44, 51

=, 44, 51
==, 44, 51

>, 44, 51
>=, 44, 51
>>, 45, 51

128-state types, 21

4-state values, 21

8085 Based System : Sio85.V, 177
8251 serial io controller, 183

A

A Register transfer level model of the cache, 236
Abstraction Levels, 37, 38
acc routines, 309
Aggregates, 31
Algorithm, 94
algorithmic style, 99
always, 100
and, 136
architectural, 15, 243
Arithmetic Operators, 44
ASIC library, 255
Assign, 123
Assignments, 101

B

Begin-End, 112
Behavioral Abstractions, 39
behavioral descriptions, 100
Behavioral Synthesis, 243
bits, 21
Bit-Selects, 52
bit-vectors, 21
block diagram of r4200, 69
blocking assignments, 101
Boolean Operators, 44
Boolean/conditional values, 22
Bottom up methodology, 14
Browser, 174
buf, 136
bufif0, 136
bufifl, 136
built-in gates, 136

C

Cache Design, 217
called continuous assignments, 40
Case, 108
Casex, 109
casez, 109
character strings, 22
Class A, 248
clock generation, 100
cmos, 138
Combinational UDPs, 141
Commands, 174
Compiler Directives, 167
Concatenation, 44
concurrent processes, 95
Conditional Paths, 302
Conditional Statement, 107
constant-valued expression, 55
constraints, 243
continuous assignment, 41
Control flow modeling, 39
Counter, 2

D

Data Declarations, 22
Data Types, 21
Datapath design, 62
debugging, 125
default, 108
define, 168
defparam, 34
Delay Specifications, 304
delay values, 22
delays, 32
Design Cycle, 154
Design Flow, 15, 243
designer driven resource sharing, 293
disable, 122
Display System Tasks, 156
double specification, 305

E

Edge Sensitive Sequential UDPs, 143
else, 170

end, 112
endcase, 108
endfunction, 119
endmodule, 64
endprimitive, 138
endspecify, 296
endtable, 138
endtask, 119
equal to, 44
Evaluate Events, 90
Evaluation Event, 92
Event, 92, 114
Event Declaration, 34
Event declarations, 114
Event Driven Simulation, 90
Event Generalization, 116
Event OR, 115
event triggering, 115
Expressions, 43

F

factorial generation, 7
fdisplay, 161
fdisplayb, 161
fdisplayh, 161
fdisplayo, 161
File Input, 161
File Management, 160
Flip-Flop Inference, 281
floating point types, 21
for, 111
for Task Enabling, 122
Force, 124
forever, 111
Fork-Join, 117
formal syntax definition conventions, 18
fstrobeh, 161
fstrobeh, 160, 161
Full Analysis, 87
Full Case, 268
Functions, 119
fwriteb, 160, 161

G

Gates, 135
ground, 27

H

H, 135
Hierarchical Names, 35
highz0, 138
highzl, 138
Hold, 297

I

if, 107
ifnone, 303
inertial delay model, 95
initial, 100
inout, 30
input, 29
instantiations, 67
Integer And Time, 33
integers, 21
Intel, 177
Interactive Simulation, 173
inter-assignment delays, 102
Internal Data Structure, 90
intra-assignment delays, 102
IVC, 1

K

keywords, 17

L

L, 135
large, 28
latch inference, 279
Level-Sensitive Sequential UDPs, 142
lexical conventions, 18
Log Of A Typical Simulator, 87
Log Of An Ideal Simulator, 88
Logic Synthesis, 243
logic values, 21
Logical Operators, 44
Loops, 111

M

Macromodules, 65
mapping, 243
Mealy machine, 248
medium, 28
memories, 31
Microprocessor, 125

Mips, 67
Mixed Level and Edge Sensitive
  Sequential UDPs, 144
Mixed Modeling, 151
Model Execution, 93
Modeling Pipelines, 118
Modern view, 248
Module Definitions, 64
Module Instantiation, 67
Monitor System Tasks, 158
Multi-driver Nets, 305
Multi-Event Event, 115

N

named ports, 66
nand, 136
negedge, 117
net, 24
Network Representation, 92
nmos, 138
non-blocking assignments, 103
non-determinism, 91
non-synthesizable code, 256
nor, 136
not, 136
not equal to, 44
notif0, 136
notifl, 136
Numbers, 51

O

Open Verilog International, 1
operands in expressions, 51
Operator Precedence, 56
Operators, 46
Operators on Reals, 55
optimize design, 243
or, 136
output, 29

P

Parallel Case, 268
parallelism, 118
parameter, 22, 34
Part-Selects, 52
Path, 300
Period, 298
Pin Timing, 295
pipelines, 118
PLI, 307
pmos, 138
Port Types, 29
posedge, 117
power supply, 27
precedence of operators, 56
Primitive, 138
Process or Evaluation Block, 92
processing an event, 94
Programming Language Interface, 307
Project Planning, 154
pull0, 138
pull1, 138
pulldown, 136, 138
pullup, 136, 138
Pulse Specification, 305

Q

quasi-continuous assignment, 123

R

R4200, 183
ram, 183
range, 31
rcmos, 137
Real, 33
Real Declaration, 33
Recovery, 298
Reg Declaration, 23
register inference, 279
Register Transfer Level Abstractions, 40
Relational Operators, 44
Release, 124
repeat, 111
resistive transistors, 138
Resource sharing, 290
retargetting, 243
rnmos, 138
rpmos, 138
rtran, 138
rtranif0, 138
rtranifl, 138
Rules for the Delay Models in SDF, 361

S

Sagdeo machine, 248
scalared, 32

Schedule, 92
scheduling an event, 95
Schematic for counter, 3
Schematics for dff, 3
sdf ABSOLUTE, 331
sdf ARRIVAL, 354
sdf CELL, 329
sdf CELLTYPE, 329, 341
sdf COND, 337
sdf CONDELSE, 337
sdf DATE, 326
sdf DELAY, 330
sdf delay model, 361
sdf DELAYFILE, 324
sdf delval, 335
sdf DEPARTURE, 354
sdf DESIGN, 326
sdf DEVICE, 340
sdf DIFF, 352
sdf DIVIDER, 327
sdf File Examples, 359
sdf header, 325
sdf HOLD, 344
sdf INCREMENT, 331
sdf INSTANCE, 330, 332
sdf INTERCONNECT, 339
sdf IOPATH, 335
sdf NOCHANGE, 349
sdf PATHCONSTRAINT, 350
sdf PATHPULSE, 332
sdf PATHPULSEPERCENT, 333
sdf PERIOD, 348
sdf PERIODCONSTRAINT, 351
sdf PORT, 339
sdf PROCESS, 328
sdf PROGRAM, 326
sdf RECOVERY, 345
sdf RECREM, 347
sdf REMOVAL, 346
sdf RETAIN, 338
sdf SETUP, 344
sdf SETUPHOLD, 345
sdf SKEW, 347
sdf SKEWCONSTRAINT, 353
sdf SLACK, 355
sdf SUM, 352
sdf temperature, 328
sdf timescale, 328
sdf Timing Checks, 323
sdf Timing Models, 323
sdf TIMINGCHECK, 341
sdf TIMINGENV, 350
sdf VENDOR, 326
sdf VERSION, 326
sdf VOLTAGE, 327
sdf WAVEFORM, 357
SDFVERSION, 325
Semantic Model, 85
sequential statements, 99
Setup, 297
Shift Operators, 44
Signal, 92
simple paths, 300
Simulation Control Tasks, 162
Simulation Time Functions, 162
small, 28
Specify Blocks, 295
specparam, 302
Standard Delay Format, 321
State-dependent paths, 301
stepwise refinement, 16, 244
stime, 162
strong0, 138
strong1,138
Structural, 135
structural model of R4200, 71
SUN, 67
supply0, 24,138
supplyl, 24, 138
Switch Declarations, 137
Switches, 138
synchronization primitives, 99
syntax conventions, 18
Synthesis, 243
Synthesis Components, 246
synthesis subset, 255
synthesizable bus latch, 286
synthesizable disable, 272
synthesizable event specification, 276
synthesizable forever loop, 272
synthesizable Task statements, 273
synthesizable adder, 249
synthesizable behavioral statements, 265
synthesizable Behavioral Constructs, 262
synthesizable case statement, 268
synthesizable continuous assignments, 258
synthesizable declarations, 257
synthesizable gates, 259
synthesizable if, 267

synthesizable multiplexer, 249
Synthesizable Operand Types, 261
synthesizable parameterized designs, 258
Synthesizable parts of the cache system, 253
synthesizable structure, 256
synthesizable three-state gates, 260
synthesizable while loop, 271
Synthesizable wire types, 257
synthesizer driven resource sharing, 294
Synthesizer ignores delay, 258
System Examples, 177
system model, 9
System Modeling, 152
System Tasks, 155

T

table, 139
Tasks, 121
technology dependent., 247
technology independent, 247
tf routines, 309
The Programming Language Interface, 307
three-state gate-inference, 287
time, 33
Timing Checks, 296
Top-Down Methodology, 14
Traditional View, 248
Traffic Light Controller, 250
tran, 138
tranif0, 138
tranifl, 138
Transistors, 315
transition values, 22
transport delay model, 97
tri, 24
tri0, 24
tril, 24
triand, 24
trior, 24
trireg, 24

U

UDP Instances, 145
UltraSPARC-IIi, 81, 82
Unary Operators, 44
units, 22
Update Event, 90, 92
User Defined Primitives, 138

V

Value Systems, 21
vdd, 27
vectored, 32
Verilog objects, 92
vpi routines, 309
vss, 27

W

Wait, 113
Wait Statements, 113
wand, 24
Waveform Interface, 163
Waveforms for the counter, 6
weak0, 138
weakl, 138
while, 111
Width, 297
wire, 24
wor, 24
write-back policy, 223
Write-Through Policy, 217, 222

X

xnor, 136
xor, 136