ANALOG
BEHAVIORAL MODELING
WITH THE VERILOG-A LANGUAGE

ANALOG BEHAVIORAL MODELING WITH THE VERILOG-A LANGUAGE

by

Dan FitzPatrick
Apteq Design Systems, Inc.

and

Ira Miller
Motorola

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

Disk only available in print edition
eBook ISBN: 0-306-47918-4
Print ISBN: 0-7923-8044-4

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

Contents

Foreword

Verilog-A is a new hardware design language (HDL) for analog circuit and systems design. Since the mid-eighties, Verilog HDL has been used extensively in the design and verification of digital systems. However, there have been no analogous high-level languages available for analog and mixed-signal circuits and systems.

Verilog-A provides a new dimension of design and simulation capability for analog electronic systems. Previously, analog simulation has been based upon the SPICE circuit simulator or some derivative of it. Digital simulation is primarily performed with a hardware description language such as Verilog, which is popular since it is easy to learn and use. Making Verilog more worthwhile is the fact that several tools exist in the industry that complement and extend Verilog's capabilities.

Although SPICE is very effective in the simulation of analog and digital integrated circuits, it is limited to the use of primitives such as transistors, resistors, and capacitors. Hence, SPICE lacks the ease that Verilog HDL possesses of describing and simulating higher-levels of abstraction of the design. In the past, this gap has been filled with such programs as Mathcad and Matlab that allow description of electronic functions based upon numeric computation and data analysis. Although these programs are useful for studying electronic and non-electronic systems at higher levels of abstraction, they do not tie into other tools such as SPICE and Verilog. The Verilog-A language enables description directly using mathematical relationships, thus easily allowing system descriptions other than electrical. Additionally, Verilog-A interfaces to numeric computation programs, such as SPICE and Verilog.

Analog Behavioral Modeling with the Verilog-A Language provides a good introduction and starting place for students and practicing engineers with interest in understanding this new level of simulation technology. This book contains numerous examples that enhance the text material and provide a helpful learning tool for the reader. The text and the simulation program included can be used for individual study or in a classroom environment.

High level languages such as Verilog-A are evolving to enable simulation of complex mixed analog and digital for both electrical and non-electrical systems. This book will get you started now.

Dr. Thomas A. DeMassa
Professor of Engineering
Arizona State University

Preface

The Verilog HDL was introduced in 1984 as a means for specifying digital systems at many levels of abstraction, from behavioral to the structural. Accepted for standardization in 1995 by the IEEE, Verilog HDL continues to grow in acceptance and play an increasing role in the specification and design of digital systems. For analog systems analysis and design, Spice, developed by the University of California at Berkeley in 1971, became the defacto standard used to simulate the performance of electronic circuits. While Spice provides a high-level of accuracy as a simulation tool, designs can only be represented on a structural level. As such, the ability to handle large analog and mixed-signal systems, as well as explore design ideas at the behavioral level, is fairly limited.

The Verilog-A language is derived from Verilog HDL for the description of high-level analog behaviors. Used in conjunction with a Spice simulator, The Verilog-A language expands the simulation capabilities for analog and mixed-signal systems to top-down and bottom-up methodologies. The proposed Verilog-A language is described in the Language Reference Manual (LRM) draft prepared by a standards working group of the Open Verilog International (OVI) organization. The LRM Version 1.0, August 1, 1996 is not yet fully defined and is subject to change. As such, the material in this book focuses on the core aspects of the Verilog-A language as presented in the LRM and the work within the OVI Verilog-A Technical Subcommittee.

The goal of this book is to provide the designer a brief introduction into the methodologies and uses of analog behavioral modeling with the Verilog-A language. In doing so, an overview of Verilog-A language constructs as well as applications using the

language are presented. In addition, the book is accompanied by the Verilog-A Explorer IDE (Integrated Development Environment), a limited capability Verilog-A enhanced Spice simulator for further learning and experimentation with the Verilog-A language. This book assumes a basic level of understanding of the usage of Spice-based analog simulation and the Verilog HDL language, although any programming language background and a little determination should suffice.

Certain typographical conventions are used to emphasize different kinds of text used in this book. Both Spice and Verilog-A code fragments are in Courier font, keywords in the respective languages are also in bold. `This is an example of Courier font with a` **`keyword`** `in bold.`

The organization of the book is such that it hopefully presents a connection between the motivation behind the development of the Verilog-A language and the capabilities it provides. Chapter 1 provides an introduction on motivations and benefits for standard analog HDLs such as the Verilog-A language. Chapter 2 is designed to provide an outline of the Verilog-A language in terms of structural and behavioral definitions. In Chapter 3 we investigate more thoroughly the behavioral aspects of the Verilog-A language, while Chapter 4 does the same for the structural constructs within the language. Chapter 5 brings these concepts together in a variety of applications presented in their entirety. The appendices provide detailed reference for those that wish to probe further into the usage and capabilities of the language.

Examples, when they are presented, are done so in terms of the Verilog-A Explorer IDE input format. The Verilog-A Explorer uses standard Spice design netlist description and simulation control constructs. A summary of Spice input file descriptions is provided for reference in Appendix E.

The Verilog-A Explorer IDE, is a Windows '95 / NT application designed to provide sufficient capabilities to the designer and/or model developer with enough capability to learn analog behavioral modelling with the Verilog-A language. The Verilog-A Explorer IDE incorporates context sensitive editors, waveform display, and simulator based on Spice3 from the University of California Berkeley along with Apteq Design Systems's Spice Analog HDL Extension Kernel and Verilog-A compiler integrated. In addition, the package is accompanied with examples to provide starting points for experimenting with the Verilog-A language.

The Verilog-A Explorer IDE is provided for *educational purposes only.* As such, there is no direct software warrantee or support provided either by Apteq Design Systems or Kluwer Academic Publishers and its dealers. It is our hope that the benefits of using the tools provided will greatly outweigh any inconvenience you may have in

using them. Detailed information regarding installation, setup, and usage of the Verilog-A Explorer IDE is presented in Appendix D. For bug reports, availability of updates, additional modeling information and/or modeling examples in the Verilog-A language, contact:

Apteq Design Systems, Inc.
652 Bair Island Rd. Suite 300
Redwood City, CA 94063-2704
support @ apteq.com

Or visit the company website at:

http://www.apteq.com

Analog and mixed-signal extensions are currently being developed under Open Verilog International via the Verilog-AMS Technical Subcommittee. You can find information regarding the Verilog-A standard, such as the Language Reference Manual via:

Open Verilog International
15466 Los Gatos Boulevard, Suite 109071
Los Gatos, CA 95032
(408) 358-9510
http://www.ovi.org.

You can participate in the Verilog-AMS Technical Subcommittee by joining the mail reflector. To join in the discussion, send a request to:

Verilog-ms@galaxy.nsc.com

Giving credit to all who contributed to the development of the Verilog-A language is difficult and we apologize to anyone we have neglected to mention. We gratefully acknowledge support from the members of board of directors and the of the OVI and especially the support of Vasilious Gerouisis of Motorola, Chairman of Technical Coordinating Committees. The Verilog-AMS Committee is chaired by Ira Miller of Motorola, and co-chairman is James Spoto of Enablix Design. The participating members of the Verilog-AMS committee included (in alphabetical order): Ramana Aisola of Motorola, Graham Bell of Viewlogic, William Bell of Veribest, Kevin Cameron of Antrim Design Systems, Raphael Dorado of Apteq Design Systems, John

Downey of Viewlogic, Dan FitzPatrick of Apteq Design Systems, Vassilious Gerousis of Motorola, Ian Getreu of Analogy, Kim Hailey of Santolina, William Hobson of Cadence Design Systems, Ken Kundert of Cadence Design Systems, Oskar Leuthold of GEC Plessy, S. Peter Liebmann of Antrim Design Systems, Ira Miller of Motorola, Tom Reeder of Viewlogic, Steffen Rochel of Simplex, James Spoto of Enablix Design, Richard Trihy of Cadence Design Systems, Yatin Trivedi of Seva Technologies, and Alex Zamfirescu of Veribest.

Special thanks in review of this book go to Dr. Richard Shi from the University of Iowa, Clem Meas of QuickStart, Peter Hunt from Portability, Dr. Robert Fox from the University of Florida, and Dr. Thomas A. DeMassa from Arizona State University for their special efforts. The following people also provided reviews of the initial drafts of this book and participated in the beta evaluation of Verilog-A Explorer (in the order their reviews were received): Ed Cheng, Xian Meng of Littlefuse, George Corrigan of Hewlett Packard, and Norman Dancer of Gigatronics, Dale Witt of Createch, and John Wynen of Research In Motion.

CHAPTER 1

Introduction

1.1 Motivation

The rapidly evolving markets in communications, computers, automotive and consumer electronics, driven by feature and cost-competition, are driving the demand for higher levels of integration of analog and digital functionality. This dynamic is pushing the need for more effective product development methodologies for analog and mixed-signal IC and electronic systems manufacturers. The scope and magnitude of new product innovations, the push towards system-on-chip integration levels, and decreasing product life cycles have all exacerbated the need for more effective design tools and methodologies which best utilize the limited availability of analog and mixed-signal IC and systems developers.

From the technical requirements perspective of product design, the increasing levels of integration required for these products and the high degree of interaction between analog and digital circuitry has moved the design into the mixed-signal realm. In digital systems design, hierarchical approaches incorporating hardware description languages (HDLs), synthesis, and use of third-party IP (intellectual property), and cell libraries have been used to alleviate the increasing demands and complexities of product design. Conversely, in analog and mixed-signal design, the approach has been bottom-up at the transistor level, effectively limiting design reuse to the particular targeted process technology (Figure 1.1).

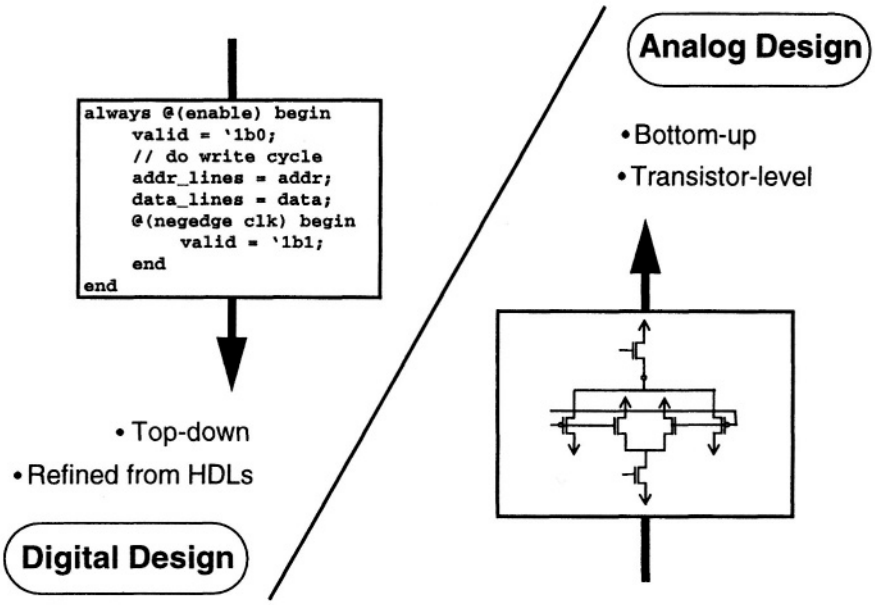


FIGURE 1.1 Illustration of the fundamental differences between the methodologies used in digital and analog design.

As the level of integration increases, it becomes increasingly necessary to provide a means to not only abstract the design, but to allow for partitioning of the design further into subsystems and components. Analyzing the architectural and technical trade-offs required for high-functionality analog and mixed-signal systems design, requires a hierarchical methodology that is tightly integrated from system specifications to silicon.

Along the way, partitioning the design into subsystems and components enables the exchange and reuse of design intellectual property from both within, and external, to the organization. Verifying the finished design performance to specifications requires determining trade-offs associated with design architecture, algorithms, and implementation - all of which can involve multiple simulation cycles. Employing higher levels of abstraction for the analog and mixed-signal components of the design is necessary for addressing the limited capacity and capabilities of traditional analog and mixed-signal simulation tools.

The Verilog-A language, the result of a two year process of development and standardization through Open Verilog International (OVI) and now continuing through IEEE, was defined to address these issues. The Verilog-A language extends the syntax and semantics of the Verilog HDL language for the description and simulation of analog and mixed-signal systems from behavioral to the circuit level.

A comprehensive set of objectives for the Verilog-A language definition were gathered by the OVI Verilog-A committee and incorporated into the OVI Design Objective Document (DOD). These objectives were used in developing the Verilog-A Language Reference Manual (LRM) by the OVI Verilog-A Technical Subcommittee. These design objectives of the Verilog-A language were considered in the context of meeting the goals of the use model of the language, including:

- Enable the communication of high-level design information including electronic and electro-mechanical or other system aspects.
- Apply behavioral approaches in the design at the architectural level.
- Encourage the exchange and reuse of design intellectual property.
- Provide a standard analog and mixed-signal description language for tool compatibility and for protecting investments in models and libraries.

1.2 Product Design Methodologies

The analog and mixed-signal product development cycle (digital, analog, and mixed-signal) for electronic IC and systems is a process involving many steps from concept to final product as shown in Figure 1.2. From the initial product concept, specifications are developed in terms of the performance and conditions under which the product is to operate. These specifications are then used to qualify an architecture, typically requiring multiple iterations in the development of a new product. Final verification of the IC is completed and verified against the extracted layout design database. After fabrication of the IC, the design is subjected to test criteria based on the original specifications. A failure to meet specifications after layout and fabrication constitutes a *design iteration* which requires repeating significant and time-consuming steps in the design process and additional costs of multiple mask set making and fab runs.

In the increasing specialization in the IC and systems design markets, such as the fabless business model, in which one or more companies performs the design role and another manufacturing and test, a similar sequential flow of product development is

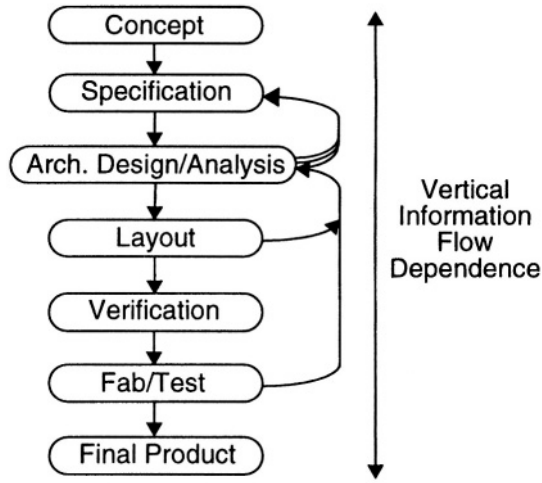


FIGURE 1.2 Standard sequential product development methodology used in IC and electronic systems products emphasizing the importance of accurate and effective tool-to-tool communication.

followed as illustrated in Figure 1.3. However, in cases where one or more organizations are involved, the need for effective and accurate communication of the design representation among different phases in the design process now crosses multiple different organizations. Here, a design house or system integrator can utilize pre-characterized process libraries from its manufacturing partner, as well as sub-chip building blocks acquired from a component library provider.

Communication of the design information between the different organizations such as these relies on a standardized means of representing the design. Increasingly, as higher levels of integration are being pursued, the type and content of this information as it encompasses analog and mixed-signal designs will also change. For example, a high-level representation of the design enables the ability to effectively make incremental changes in the design functionality, such as higher frequency operation, noise immunity, etc. (Figure 1.4). High-level representations of designs can facilitate functional and/or process portability. Hence, the representation of the behavioral or structural aspects of a design can and should be independent of a underlying fabrication process technology, allowing and encouraging maximum re-targetability of the design to a new target process.

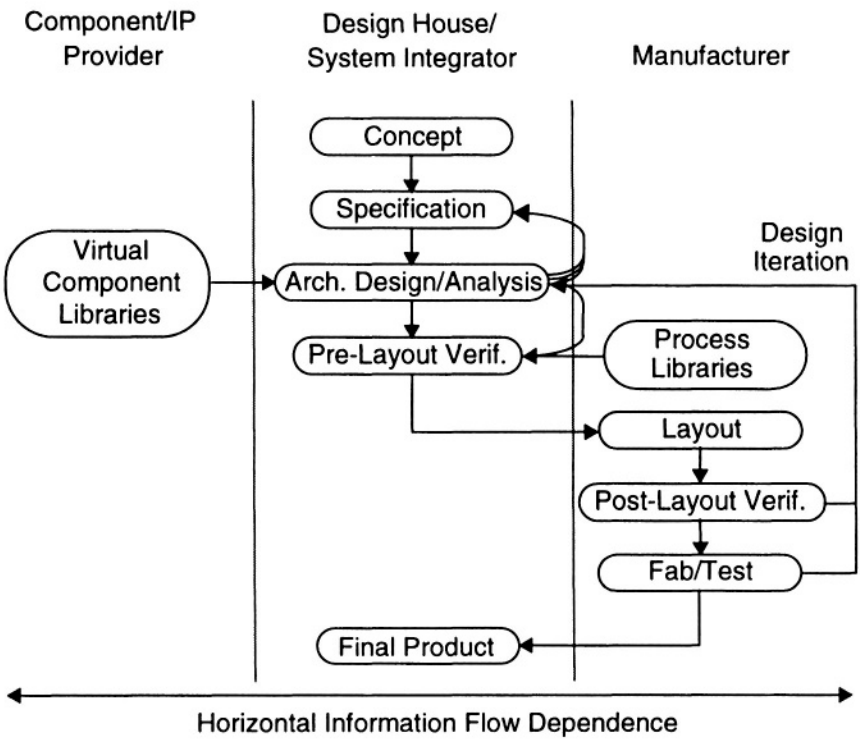


FIGURE 1.3 Horizontal information flow dependencies between a design house and both component and process library technology providers emphasizing the dependence of communication between organizations.

In both of these product design flows, the sequential and iterative nature of the product development process highlight some very important aspects:

- As in digital systems design, top-down methodologies are required to perform analyses of architectural trade-offs, evaluate library options, and reduce costly design iterations.
- Accurate and effective communication of the representation of the design is crucial between the hierarchical stages of the design.

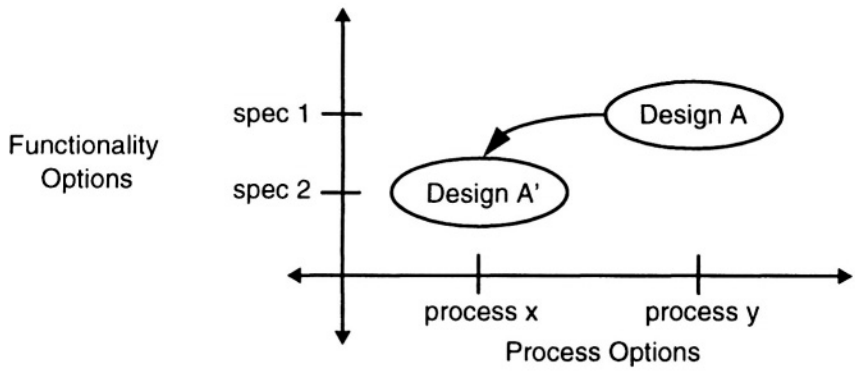


FIGURE 1.4 Higher-level representations of the design facilitate the functional and fabrication process portability of the design (here, design A from spec 1 on process y to spec 2 on process x).

- Among multiple groups participating in the design, accuracy in the representation of the design is crucial as the complexity of the development, as well as the diversity in the tools, becomes greater.
- The sequential nature of the product development process has a two-fold impact in that steps within the process typically do not occur concurrently and errors at any stage can require costly backtracking in terms of time and money.
- High-level design methodologies enable concurrent activities in the development flow, such as in design, verification, and test, enabling shorter product development cycles.

In addition to technical considerations, the business model dictates that the design information exchanged can incorporate proprietary information - either from the foundry in terms of process libraries, the design house in terms of the design, or a third party vendor whose primary function is solely to provide intellectual property. The proprietary nature of the information is typically reflected in terms of implementation - further emphasizing the need for different levels of design abstraction.

One of the primary focus of the Verilog-A language is towards enhancing the portability of designs between suppliers and customers as well as allowing for best-in-class tool solutions. A high level of design abstraction such as the Verilog-A language for analog and mixed-signal designs, maximizes the effectiveness of communication between different levels of designers within product design, verification, test, as well as IP providers and foundries. The high-level description can also be used for verify-

ing the implementation against the original specifications. This capability has one of its most profound effects in minimizing the design iterations by simply allowing for system-level verification.

1.3 The Role of Standards

Representation of design information, including specifications, has evolved from specialized tools targeted towards accomplishing specific roles in the product development process. Generally speaking, these can be categorized based on the types of designs for which they represent and the level of abstraction in which those designs are described as shown in Figure 1.5.

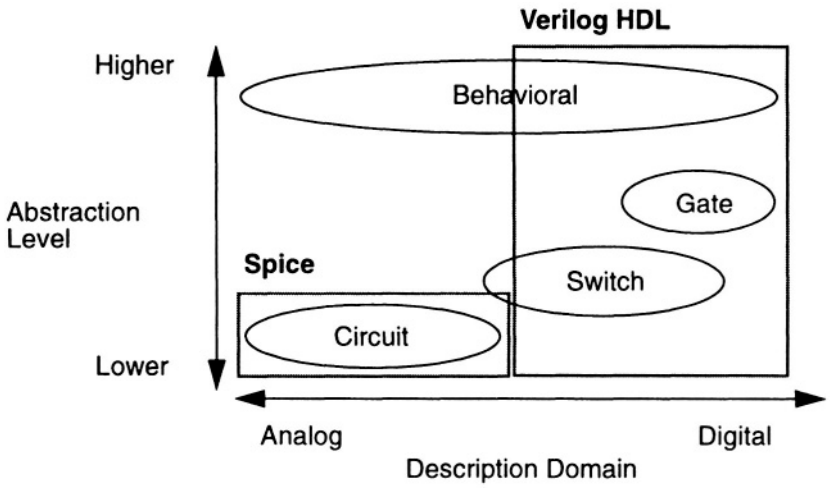


FIGURE 1.5 The scope of Spice and Verilog HDL within the abstraction level/description domain continuum.

Some design representations are capable of spanning multiple abstraction levels. Verilog HDL, for instance, is able to represent digital designs at switch, gate, and behavioral levels. Conversely, more structural representations of design, such as SPICE, are only capable of representing a design at the lowest circuit-level.

Whether the scope of a standard is an industry, or a defacto standard limited to a single company or tool, design representations such as Spice or Verilog HDL, and the

libraries and infrastructure built on top of them, represent some of the largest investments in design methodologies within companies. The use of non-standard solutions, able to target specific niches within an industry or application area, must be balanced against the risks proprietary solutions impose on these investments. The ability to maintain these investments becomes a crucial consideration in the adoption of a design methodology.

1.3.1 Verilog-A as an Extension of Spice

The Verilog-A language was designed to be compatible as an extension of Spice for both low- and high-abstraction levels. Similar to Verilog HDL and its ability to span the range of abstraction levels for digital descriptions, the Verilog-A language was designed to function just as effectively at describing high-level analog behaviors as well as circuit level descriptions. The syntactic heritage of Verilog-A is Verilog HDL, but semantics derived from standard Spice in terms of capabilities such as the types of designs and analyses supported (Figure 1.6).

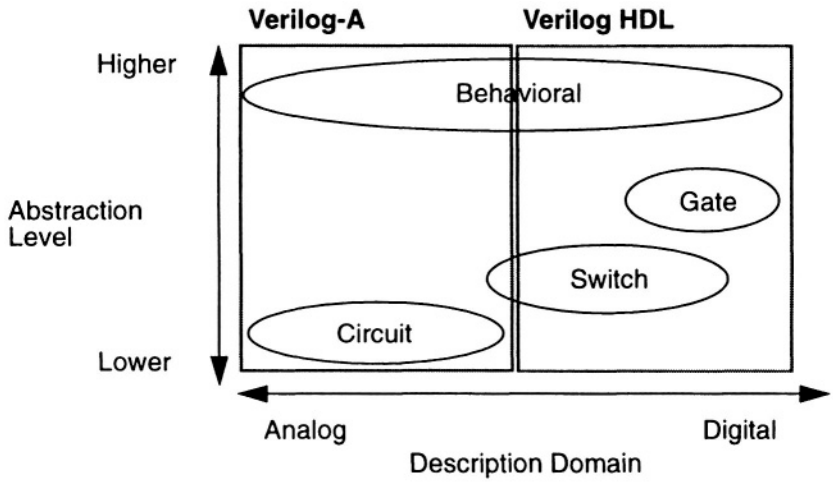


FIGURE 1.6 The extended scope of Verilog-A encompassing that of traditional Spice as well as high-level behavioral representations of analog circuitry.

Building on the standards of Spice and Verilog HDL provides an opportunity not only to address product development needs in a technological sense, but also provide a

transition path from current design methodologies and infrastructure. Based on traditional Spice design methodologies, the Verilog-A language allows utilization of existing frameworks, libraries, models, and training.

1.4 The Role of Verilog-A

The Verilog-A language allows the description of analog and/or mixed-signal systems with varying amounts of detail. The analog behavioral capability allows the designer to span the abstraction levels, allowing direct access to the underlying technology while maintaining the capability of system-level modelling and simulation. As such, the analog and mixed-signal system can be described and simulated at a high-level of abstraction early in the design cycle to facilitate full-chip architectural trade-offs. The resulting Verilog-A description, as an executable specification, promotes communication and consistency throughout the design process (from specification to implementation).

A standardized analog behavioral modeling language such as the Verilog-A language, with capabilities from the behavioral to circuit-level provides:

- An enabling technology for analog and mixed-signal top-down design
  - Managing complexity and significant performance factors within the design
  - Specification, documentation, and simulation
- A compact and clear expression of design intent
  - Independent of the implementation
  - Behavioral model reuse enabling design reuse
- Standardized form of communication of design information
  - Between tools within the design flow
  - Between product development groups for exchange and reuse
  - Virtual component IP providers
  - Semiconductor foundries
- Concurrent development for shortening product development life cycles
  - Design, verification, and test program development

1.4.1 Looking Ahead to Verilog-AMS

The Verilog-AMS specification, currently under development by Open Verilog International, is targeted to be a single-language solution for the specification and simulation of analog, digital, and mixed-signal systems. The objectives of the Verilog-AMS specification are to facilitate portable mixed-signal system description and simulation. In addition, a design described with the Verilog-AMS language will provide the capability to integrate system and circuit-level aspects of the design allowing the design intent to be maintained throughout the entire mixed-signal design process.

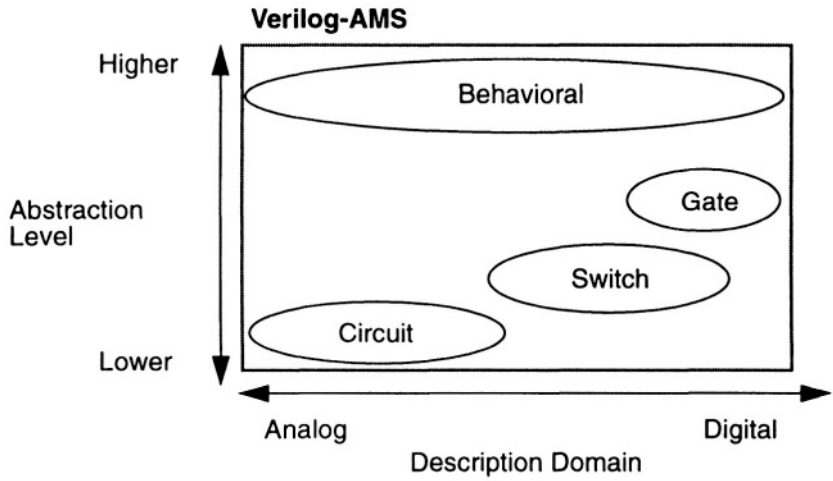


FIGURE 1.7 The scope of the design descriptions targeted by Verilog-AMS. Full mixed-signal specification, design, and simulation within a single language.

The Verilog-AMS specification, yet to be finalized at the time of this writing, is beyond the scope of this introductory book.

CHAPTER 2

Analog System Description and Simulation

2.1 Introduction

The Verilog-A language gives designers the flexibility to describe systems at multiple levels of abstraction for architectural definition, verification, and analysis. The basis for both structural and behavioral descriptions in the Verilog-A language are *modules.* A module definition can incorporate parametric and/or structural declarations (instantiation or creation of other modules), behavioral descriptions, or all three.

Structural descriptions allow system definition via pre-defined, user-defined, or third-party-defined components. The instantiation of a module definition in a larger system defines a component or instance of that system. In an analog HDL such as Verilog-A, behavioral descriptions map directly to the mathematical relationships of the system. Both the structural and behavioral abstractions of system definitions share the signals of the system. Signal definitions in Verilog-A have their basis in both the requirements of their usage for behavioral descriptions and the underlying requirements of analog simulation.

This chapter introduces the fundamental aspects of the representation of analog and mixed-signal systems with the Verilog-A language.

2.2 Representation of Systems

In general, systems are considered to be a collection of interconnected components that are acted upon by a stimulus and produce a response. The Verilog-A language allows analog and mixed-signal systems to be described by a set of components or modules. The module definition declares the mechanisms by which it is connected as well as the behavior that it contributes in the system performance. Each of these modules in the system can be described by specifying the following:

- Structural descriptions in which a module is comprised of other child modules. Each module in the structural definition of the system connects to one or more signals through the module's ports or connection points.
- Behavioral descriptions in a programmatic fashion with the Verilog-A language. The behavior of a module is defined in terms of the values for each signal.
- Mixed-level descriptions combine aspects of both structural and behavioral descriptions at a variety of different abstraction levels.

The behaviors comprising a system described in the Verilog-A language can be at various levels of abstraction depending upon the level of detail required.

An example of an analog and mixed-signal system is a modem, as shown in Figure 2--1. A system-level simulation and verification would encompass not only simulation

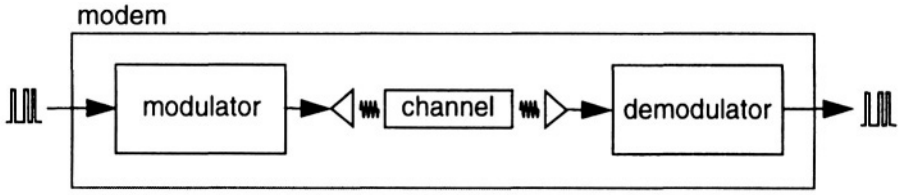


FIGURE 2.1 A modem as an example of an analog and mixed-signal system.

of the hardware comprising the system (modulator, demodulator, carrier recovery circuits, etc.) but also include effects of the environment in which the system is to operate. In this particular example, a model of the channel is used to determine the effect of channel distortion on the transmission integrity.

In defining a system such as a modem, it is useful to encapsulate the components of the system into manageable sub-components. In the Verilog-A language, the mechanism by which this is done is the **`module`** definition.

2.2.1 Anatomy of a Module

A Verilog-A module definition not only defines the behavior of the component, but must declare the interface necessary to configure and connect the component. These interface declarations are used in the composition of the structural descriptions of systems. The interface declarations for a module include signal as well as parameter declarations. We will be using the module definition of the modem as an example (Listing 2.1). Later we will expand this module definition into a more detailed example using a 16-QAM (Quadrature Amplitude Modulation) architecture.

LISTING 2.1 Example module definition for the modem of Figure 2-1.

```
module modem(dout, din);
    inout dout, din;
    electrical dout, din;

    parameter real fc = 100.0e6;

    // structural description (section 2.2.2)

    // behavioral description (section 2.2.3)

endmodule
```

port signal declarations and connections

parameter declarations

The connection points of the module are defined by the *port* signal interface declarations. The module defines the external ports or signals to which the module can connect as a component in the system. In this example, these signals are the indicated by the identifiers `dout` and `din`. The module also defines any directionality associated with those connection points (in Listing 2.1, the connection points `dout` and `din` are defined as **`inout`** or bidirectional), as well as the type of the analog signals (`electrical`).

The other facet of the interface declarations are parameter definitions which allow the characterization of the behavior of the component when it is used within a design. In the modem example of Listing 2.1, a real-valued parameter `fc` is declared with a default value of 100.0e6. Signal and parameter interface declarations are covered in more detail in Chapter 4.

2.2.2 Structural Descriptions

Structural descriptions of a module for defining system behavior can also be done with the Verilog-A language. A structural description in Verilog-A is any description in which a module *instantiates* or creates another module within its definition. A structural definition for the modem will define an explicit hierarchy, or parent-child relationship between modules in the system. The example 16-QAM modem has a parent-child relationship with its the modulator, channel, and demodulator instances (Figure 2.1). The module definition of the modem will declare the names, and assign parameter values and connections for each of its child-modules via *instantiation statements.*

The structural definition of systems allows the designer to pass parametric specifications, as well as connections, throughout the levels of hierarchy in the design. The assignment of the parameters and connections of child modules is done via *parameter* and *port association.* The Verilog-A language allows parameters to be assigned and ports to be connected by position or name. Structural definitions such as for the 16-QAM modem are derived from Verilog HDL, and are done in a programmatic fashion as illustrated in Listing 2.2.[1]

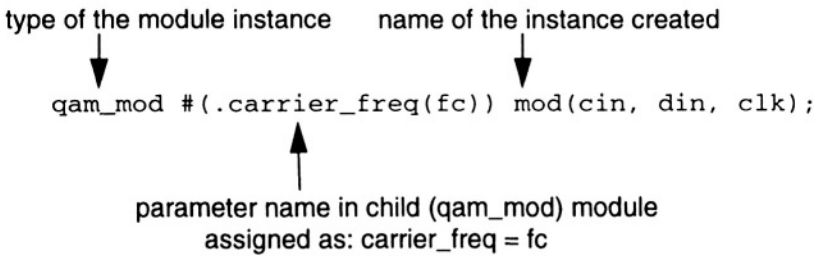
LISTING 2.2 Verilog-A definition of the modem system in Figure 2-1.

```
`include "std.va"

module modem(dout, din);
    inout dout, din;
    electrical dout, din;

    parameter real fc = 100.0e6;

    electrical clk, cin, cout;

    qam_mod #(.carrier_freq(fc)) mod(cin, din, clk);
    channel c1(cout, cin);
    qam_demod #(.carrier_freq(fc)) demod(dout, cout,
        clk) ;
```

1. Verilog-A language extends the Verilog HDL specification for structural definition via the addition of named association for parameters. This is discussed in more detail in the following chapters. In addition, parameter types can be specified in Verilog-A as opposed to taking the type of the initializer expression.

```
endmodule
```

A module instantiation in the Verilog-A language is similar to a variable declaration in programming languages. The module type name declares the module instance type, followed by optional parameter settings (within the "#(...)" construct), the instance name, and the connection list. From Listing 2.2, the following is used to illustrate the module instantiation syntax:



The module type name `qam_mod` creates the instance named `mod`. The `mod` instance is passed the value `fc` as the value for the parameter `carrier_freq` to the instance. The instance is connected to signals `cin, din` and `clk` within the definition of the module `modem`. The instantiation for the `qam_mod` instance `mod`, and the other two component instantiations within the `modem` module definition in Listing 2.2 declares the design hierarchy of Figure 2.2.

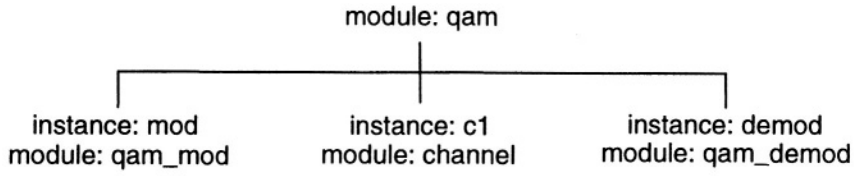


FIGURE 2.2 Hierarchical view of the modem system.

Structural definitions in the Verilog-A language facilitate the use of top-down design methodologies. As architectural design progresses, structural and behavioral definitions with finer details of description can be substituted for determining the system

performance to specifications. Utilizing this capability requires no more than an understanding of the parameter and port definitions of a module.

2.2.3 Behavioral Descriptions

The Verilog-A language provides for describing the behavior of analog and mixed-signal systems. The analog behavioral descriptions are encapsulated within **`analog`** statements (or blocks) within a module definition. The behavioral descriptions are mathematical *mappings* which relate the input signals of the module to output signals in terms of a large-signal or time-domain behavioral description. The mapping uses the Verilog-A language *contribution operator* "<+" which assigns an expression to a signal. The assigned expression can be linear, non-linear, algebraic and/or differential functions of the input signals. These large-signal behavioral descriptions define the *constitutive* relationship of the module, and take the form:

```
output_signal <+ f( input_signal );
```

In signal contribution, the right-hand side expression, or `f( input_signal )`, is evaluated, and its value is assigned to the output signal. Consider, for instance, the representation of a resistor connected between electrical nodes `n1` and `n2`:

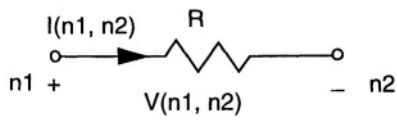


FIGURE 2.3 Resistor model and reference directions.

The constitutive relationship of the element could be encapsulated as a module definition in the Verilog-A language as shown in the `resistor` module definition of Listing 2.3.

LISTING 2.3 Verilog-A module of the resistor in Figure 2.3.

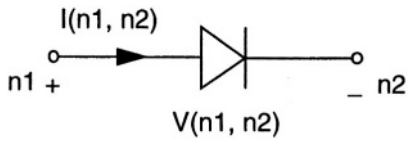
```
module resistor(n1, n2);
    inout n1, n2;
    electrical n1, n2;

    parameter real R = 1.0;

    analog
```

```
        I(n1, n2) <+ V(n1, n2)/R;

endmodule
```

where `V(n1, n2)` is the voltage across the resistor connected between nodes `n1` and `n2` of the module, and `I(n1, n2)` is the current through the branch connecting nodes `n1` and `n2`. The behavior of the module is defined by the **`analog`** statement within the module definition. In the `resistor` of Listing 2.3, the **`analog`** statement is a single line description of the voltage and current relationship of the resistor related by the contribution operator.

It is important to note that the contribution operator is a concise description of the behavior of the element in terms of its terminal voltages and currents. The simulator becomes responsible for making sure that the relationship established by the contribution operator is satisfied at each point in the analysis. This is accomplished via the strict enforcement of conservation laws that the Verilog-A language semantics defined for the simulation of analog systems.

This simple way of representing the behavior of the system allows designers to easily explore more complex constructs like non-linear behaviors, as in the diode in Figure 2.4.



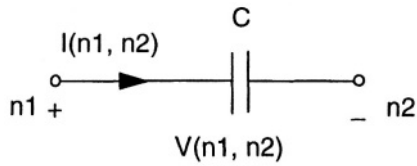
$$i_d = i_{sat}\,(\exp(V(n1, n2)/V_T) - 1.0)$$

FIGURE 2.4 Diode model and reference directions.

The behavior of the diode can be defined in the Verilog-A language as,

```
I(n1, n2) <+ isat*(exp(V(n1, n2)/$vt()) - 1.0);
```

where `$vt ()`[1] is a Verilog-A system task that returns the thermal voltage. Time-differential constructs as in the capacitor:



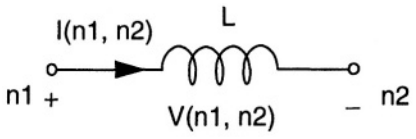
$$i_c = \frac{d}{dt}(C \cdot V(n1, n2))$$

FIGURE 2.5 Capacitor model and reference directions.

can be expressed in the Verilog-A language as,

```
I(n1,n2) <+ ddt(C*V(n1, n2));
```

where **ddt**() performs time-differentiation of its argument. The time-integral description of an inductor:



$$i_l = \int_0^t \frac{1}{L}(V(n1, n2))\,dt$$

FIGURE 2.6 Inductor model and reference directions.

is represented simply as,

```
I(n1, n2) <+ idt(V(n1, n2)/L);
```

1. System tasks are a general class of functions within the Verilog language that are prefixed by ($). $**vt**() is a system task associated with the Verilog-A language. Refer to Appendix B for more information on this and other system tasks.

Using **idt**() for time-integration of its argument.

Higher level representations of behavior can be defined similarly in a simple programmatic fashion using the Verilog-A language. In the following example, a simple signal-flow representation is used to represent the system such as in Figure 2.7.

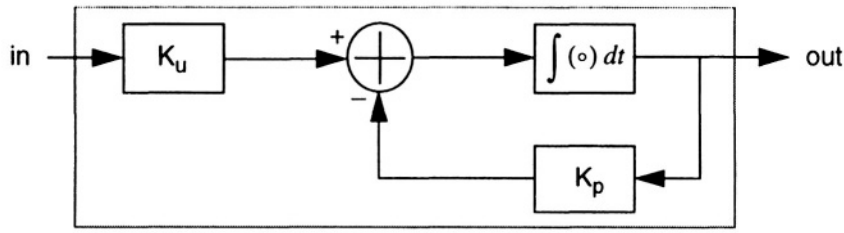


FIGURE 2.7 High-level system schematic.

The behavior of Figure 2.7 can be expressed compactly in the Verilog-A language as (derived in terms of the signal at `V(out)`),

```
V(out) <+ idt(Ku*V(in) - Kp*V(out));
```

where the behavior is formulated in terms of `V(out)`. Alternatively, using other constructs within the Verilog-A language, the behavior can also be expressed as,

```
V(out) <+ laplace_nd(V(in), { Ku }, { Kp, 1 });
```

where **laplace_nd**() is a transfer function representation of the behavior. These behavioral constructs will be discussed in more detail in Chapter 3.

2.3 Mixed-Level Descriptions

The Verilog-A language allows the designer the flexibility to model components at various levels of abstraction. Mixed-level descriptions can incorporate behavior and structure at various levels of abstraction. Flexibility in choosing the level of abstraction allows the designer to examine architectural trade-offs for design performance and physical implementation.

One of the techniques available to designers for mixing levels of abstractions are mixed-level descriptions themselves - module definitions that incorporate both structural and behavioral aspects. In addition to mixing structure and behavior, the Verilog-A language is designed to accommodate the structural instantiation of Spice primitives and subcircuits, within the module definition[1]. This methodology provides a path to final verification within the design cycle, when detailed models are necessary for insuring adherence to performance specifications.

For example, for the 16-QAM modem system, a block diagram of the modulator module, `qam_mod`, could be defined as shown in Figure 2.8.

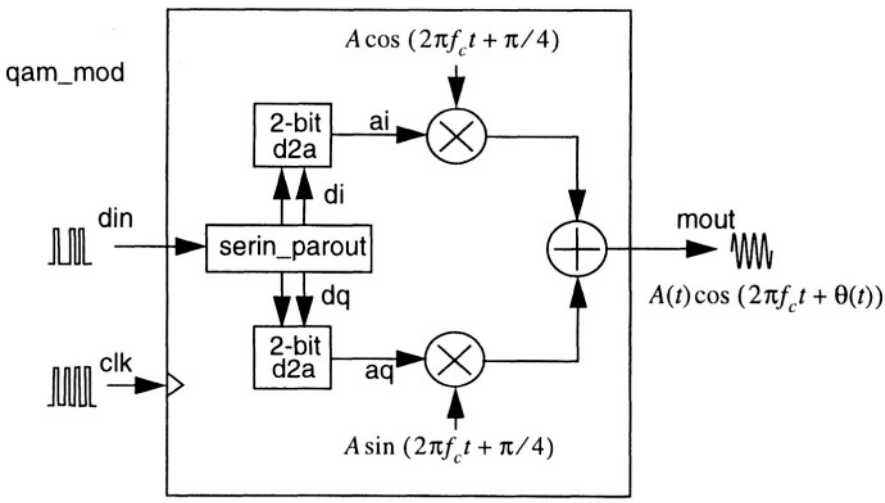


FIGURE 2.8 Architecture of the 16-QAM modulator.

The definition of module `qam_mod` can include behavioral and structural aspects. In Listing 2.4, the module definition instantiates components that provide the serial-to-parallel conversion of the incoming digital data stream. The QAM modulation is defined behaviorally in terms of its mathematical representation. The signals and parameters declared within the module definition can be shared between both the

1. One method is demonstrated in this book, but the specification permits some flexibility in this aspect.

structural and behavioral aspects within the same module, providing a high-degree of flexibility within the design process. For example, in Listing 2.4, the signals `ai` and `aq` are used within both the structural and behavioral aspects of the 16-QAM module definition.

LISTING 2.4 Verilog-A definition of 16-QAM modulator

```
`include "std.va"
`include "const.va"

module qam_mod(mout, din, clk);
   inout mout, din, clk;
   electrical mout, din, clk;

   parameter real fc = 100.0e6;

   electrical di1, di2, dq1, dq2;
   electrical ai, aq;
   serin_parout sipo(di1, di2, dq1, dq2, din, clk);
   d2a d2ai(ai, di1, di2, clk);
   d2a d2aq(aq, dq1, dq2, clk);

   real phase;

   analog begin
      phase = 2.0*`M_PI*fc*$realtime() + `M_PI_4;
      V(mout) <+ 0.5*(V(ai)*cos(phase) +
         V(aq)*sin(phase));
   end

endmodule
```

The signals `ai` and `aq` are the outputs of the 2-bit D/A converters. The behavioral definition of the QAM modulation is defined:

```
V(out) <+ 0.5*(V(ai)*cos(phase) + V(aq)*sin(phase));
```

Expressed mathematically, the behavior is ideal, de-emphasizing any non-idealities in the multiplier implementations. During the later parts of the design cycle, it may be necessary to determine the impact on the performance of these non-idealities in the modulator description.

2.3.1 Refining the Module

Including the nonlinearities of the multipliers found in the behavioral description for the modulator may be required in simulations for evaluating the modem system performance. One method of doing this is to allow the mixing of Verilog-A and Spice built-in primitives and subcircuits via a generalization of the module concept. A corresponding structural view to the behavioral representation of the modulator is shown in Figure 2.9.

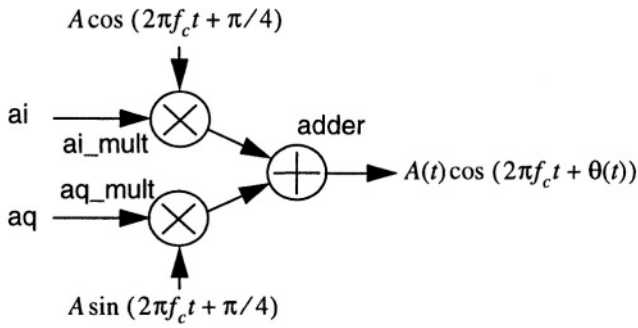


FIGURE 2.9 Structural view of modulator emphasizing the in-phase and quadrature multipliers and adder.

If the structural definition of the `qam_mod` module is expanded further to account for the multipliers used in each branch of the modulator (done behaviorally), the instance hierarchy shown in Figure 2.10 would result. Here we indicate two different modules that could be used for the `ai_mult` instance, `gilbert_va` and `gilbert_ckt`. The following Verilog-A description of the `gilbert_va` module is shown in Listing 2.5.

LISTING 2.5 Verilog-A description of multiplier

```
module gilbert_va(outp, outn, in1p, in1n, in2p, in2n);
   inout outp, outn, in1p, in1n, in2p, in2n;
   electrical outp, outn, in1p, in1n, in2p, in2n;

   parameter real gain = 1.0;

   analog begin
      V(outp, outn) <+ gain*V(in1p, in1n)*
```

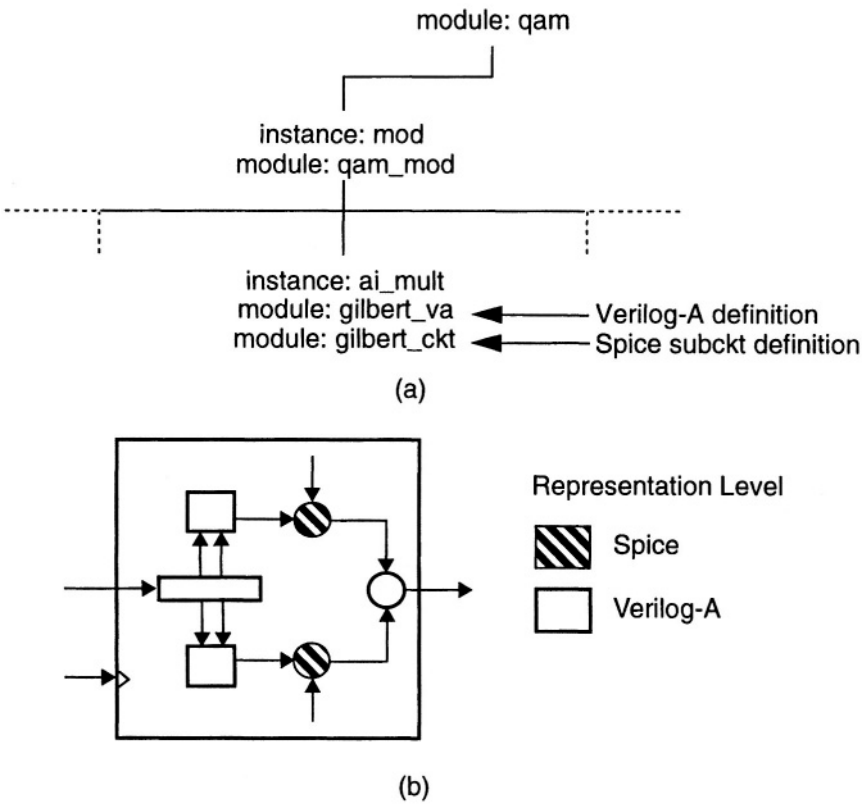


FIGURE 2.10 Hierarchical (a) and schematic (b) views of a mixed-level representation of the 16-QAM modulator.

```
        V(in2p, in2n);
    end

endmodule
```

as one representation for the multiplier behavior within the modulator. Given the schematic representation in Figure 2.11 of a four-quadrant Gilbert Cell multiplier, a structural representation of the multiplier can be defined in Spice netlist syntax as in Listing 2.6.

LISTING 2.6 Spice netlist of Gilbert Cell of Figure 2.11.

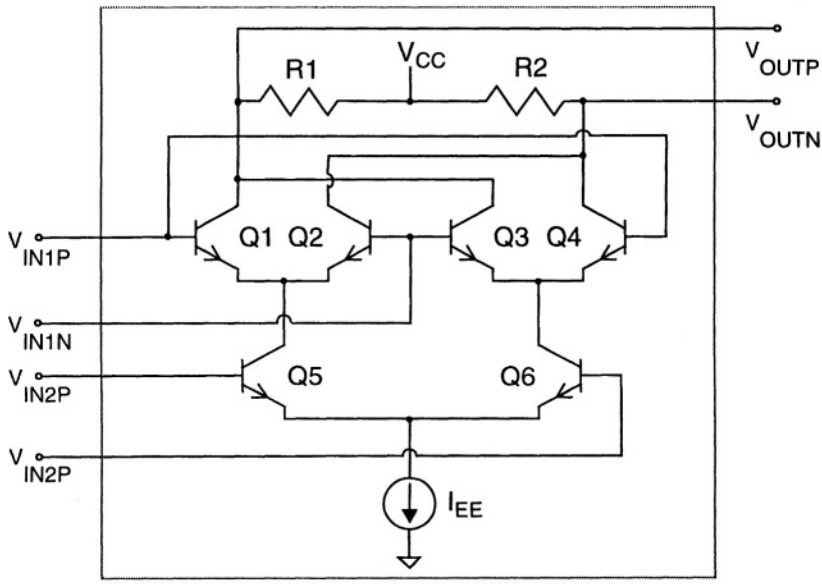


FIGURE 2.11 Schematic representation of the Gilbert Cell multiplier.

```
.subckt gilbert_ckt outp outn in1p in1n in2p in2n
   Q1 outp out1p n1 npn_mod
   Q2 outp in1n n2 npn_mod
   Q3 outp in1n n2 npn_mod
   Q4 outn in1p n2 npn_mod
   Q5 n1 in2p n3 npn_mod
   Q5 n2 in2n n3 npn_mod

   iee n3  0 dc 1m
   vcc vcc 0 dc 5.0

   r1 vcc outp 200
   r2 vcc outn 200
.ends
```

The structural description of the modulator can now be described utilizing both the behavioral and physical representations of the multipliers. For the physical represen-

tation of the multiplier, either the Verilog-A representation or the Spice subcircuit netlist can be used. Thus, in general, mixed levels of behavioral and structural descriptions can be used in the description and analysis of the system.

2.4 Types of Analog Systems

The structure of the components in an analog system, and behavioral descriptions of those components define the system of ordinary differential equations, or ODEs, that govern the response of the analog system to an external stimulus. The process by which the system of equations is derived is known as formulation. From a systems perspective, two types of systems can be described - *conservative* and *signal-flow*. A conservative type of system, which includes those described by conventional Spice, incorporates a set of constraints within the system that insure conservation of charges, fluxes, etc. within the system. Signal-flow systems employ a different level of formulation, which focuses only on the propagation of signals throughout the system.

2.4.1 Conservative Systems

Conservative systems are those that are formulated using conservation laws at the connection points or nodes. An important characteristic of conservative systems is that there are two values associated with every node or signal (and hence every port of a component of the system) - the potential (or across value) and the flow (or thru value). In electrical systems, the potential is also known as *voltage* and the flow is known as the *current*. The Verilog-A language uses potential and flow as a generalization for the description of multi-disciplinary systems (e.g., electrical, mechanical, thermal, etc.).

The potential of the node is shared with all ports connected to the node in such a way that all ports see the same potential. The flow is shared such that flow from all terminals at a node must sum to zero. In this way, the node acts as an infinitesimal point of interconnection in which the potential is the same everywhere on the node and on which no flow can accumulate. The node embodies the conservation laws, Kirchoff's Potential Law (KPL) and Kirchoff's Flow Law (KFL) in the equations that describe

the system. KPL and KFL are a generalization of KVL and KCL for electrical systems which allow the conservation laws to be applied to any conservative system.

Electrical Systems	Any Conservative System
KVL	KPL
KCL	KFL

FIGURE 2.12 Relation between Kirchoff's laws for electrical systems (conservative) and the generalized forms of KPL and KFL for any conservative system.

In a conservative system, when a component connects to a node through a port, it may either affect, or be affected by, either the potential at the node, and/or the flow onto the node through the port. A basic example of a conservative component is a resistor. The voltage across the resistor is dependent on the current flow and vice-versa. Changes in the potential at, or flow into, either end of the device would affect the other end to which the resistor component is connected.

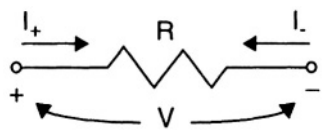


FIGURE 2.13 Definitions of the conservative relationships of a resistor.

2.4.2 Branches

With conservative systems it is also useful to define the concept of a branch. A *branch* is a path of flow between two nodes. Every branch has an associated potential (the

potential difference between the two branch nodes), and a flow. The reference directions for a branch are as follows:

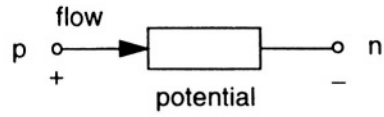


FIGURE 2.14 Associated potential and flow reference directions for a branch.

The reference direction for a potential is indicated by the plus and minus symbols at each end of the branch. Given the chosen reference direction, the branch potential is positive whenever the potential of the branch marked with a (+) sign is larger than the potential of the branch marked with a minus (–) sign. Similarly, the flow is positive whenever it moves in the direction of the arrow (in this case from + to –). In the Verilog-A language, for an electrical device, the potential would be represented by `V(p,n)`, and the associated flow would be represented by `I(p,n)`:

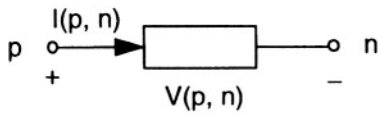


FIGURE 2.15 Associated potential (voltage) and flow (current) reference directions for a branch in an electrical system.

The potential of a single node is given with respect to a *reference node.* The potential of the reference node, which is called *ground* in electrical systems, is always zero.

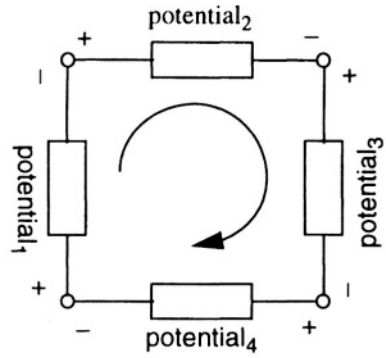
2.4.3 Conservation Laws In System Descriptions

There are two types of relationships used for defining conservative systems. The first of these are the constitutive relationships that describe the behavior of each instance of the design. Constitutive relationships for a component or module can be described in the Verilog-A language or built into a simulator as Spice-level primitives.

The second set of relationships for conservative systems are the interconnection relationships which describe the structure of the network. *Interconnection relationships* contain information on how the components are connected to each other, and are only

a function of the system topology. The interconnection relationships define the conservation of energy within the analog system.

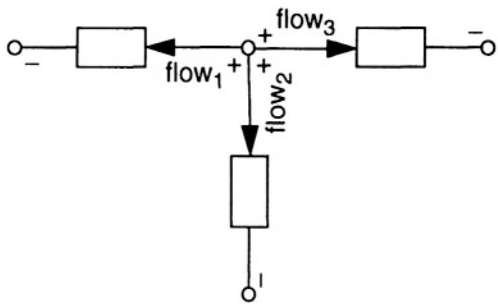
KPL and KFL can be used to determine the interconnection relationships for any type of system. KPL, is illustrated in Figure 2.15, where the sum of the potentials around



$$potential_1 + potential_2 + potential_3 + potential_4 = 0$$

FIGURE 2.16 Illustration of Kirchoff's Potential Law; a generalized form of Kirchoff's Voltage Law for conservative systems.

the loop are zero. KFL, likewise, is illustrated in Figure 2.16, in which the sum of the



$$flow_1 + flow_2 + flow_3 = 0$$

FIGURE 2.17 Illustration of Kirchoff's Flow Law; a generalization of Kirchoff's Current Law for conservative systems.

flows into a node is zero. Both KPL and KFL are used to relate the values on nodes and branches. The application of both KPL and KFL imply that a node is infinitely

small so that there is negligible difference in potential between any two points on the node and there is a negligible accumulation of flow.

2.4.4 Signal-Flow Systems

Unlike conservative systems, signal-flow systems only have a potential associated with every node. As a result, a signal-flow port must be unidirectional. It may either read the potential of the node (input), or it may assign it (output). Signal-flow terminals are either considered input ports if they pass the potential of the node into a component, or output ports if they specify the potential of a node.

A typical signal-flow component is an amplifier (Figure 2-18) with an input port

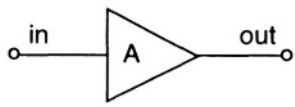


FIGURE 2.18 Simple signal-flow gain block representation.

defined as `in`, and an output port defined as `out`. The behavior would be expressed as,

```
V(out) <+ A*V(in);
```

Changes in potential of the `in` port would be reflected as `A*V(in)` on the port `out`. However, any changes on the output port `out` would not be seen by the input port `in`.

Signal-flow ports support a subset of the functionality of conservative ports in that KFL is not enforced. As such, one can always use conservative semantics to represent signal-flow components.

2.5 Signals in Analog Systems

The Verilog-A language supports the description and simulation of systems used in many disciplines such as electrical, mechanical, fluid dynamics, and thermodynamics. To accomplish this, Verilog-A uses the concepts of a **`discipline`** and **`nature`**

for encapsulating the characteristics and physical quantities associated with the different types of analog signals. A **`nature`** definition defines the characteristics of quantities to the simulator, while a **`discipline`** definition composes one or more **`nature`** definitions into the definition of an analog signal.

In Verilog-A, the primary motivation for providing this level of detail using **`disciplines`** and **`natures`** within the language is to support model portability amongst different analog simulators. Standard definitions of disciplines and natures predefined within the standard definitions[1] and are summarized in the following table:

discipline	potential nature	flow nature	potential access	flow access
electrical	Voltage	Current	V	I
magnetic	Magneto_Motive_Force	Flux	MMF	Phi
thermal	Temperature	Power	Temp	Pwr
kinematic	Position	Force	Pos	F
rotational	Angle	Angular_Force	Theta	Tau

For conservative analog systems, a discipline definition will have two natures associated with it - the potential nature and the flow nature. The definition will also define how these different components of the signal are utilized within behavioral descriptions. From the perspective of a user of the Verilog-A language, a discipline type is used for declaration of, and accessing quantities composing analog signals.

To use the quantities associated with an analog signal, *access functions*[2] are used to access values on nodes, ports, or branches. The name of the access function for a signal is taken from the discipline definition of the node, port or branch to which the signal is associated. The signal access functions are used in both reading and assigning signal values.

The Verilog-A language uses these access functions, together with the concept of probes and sources (described in the Section 2.6), to describe analog behaviors. If the

1. The examples in this book reference the standard definitions via the Verilog pre-processor mechanism, **`'include`** `"std.va"`.

2. The term access function is a syntactic description.

access function is used in an expression, the access function returns the value of the signal. If the access function is being used on the left side of a branch assignment or contribution statement, it assigns a value to the signal.

In most cases, the standard definitions of disciplines and natures can be used without modification.

2.5.1 Access Functions

The access functions for an analog signal are derived from the **discipline** definition that declares it. For conservative nodes (or ports and branches of conservative nodes), there are access functions for both the potential and flow.

For example, given the following portions **nature** and **discipline** definitions for type `electrical` (taken from the standard definitions file):

LISTING 2.7 Nature and discipline definitions for electrical systems.

```
// Current in amperes
nature Current
   units      = "A";
   access     = I;
   abstol     = 1e-12;
endnature

// Potential in volts
nature Voltage
   units      = "V";
   access     = V;
   abstol     = 1e-6;
endnature

discipline electrical
   potential    Voltage;
   flow         Current;
enddiscipline
```

The **discipline** `electrical` consists of two **nature** declarations, `Voltage` and `Current`, respectively, with the **potential** and **flow** aspects of the quantities of signals of type `electrical`. These **nature** definitions define the physical

quantities of signals of type `electrical` as well as how to access (reading and assigning) those signal values.

From this definition of the **`discipline`** `electrical`, the following

```
electrical n1, n2;
```

declares two nodes `n1`, and `n2`. To determine the **`potential`** or `Voltage` between the two nodes, `n1` and `n2`, the Verilog-A language uses the construct:

```
V(n1, n2)
```

where `V` is the access function for the *potential nature* of signals of type `electrical`. Similarly, to determine the **`flow`** or `Current` between the two nodes, use:

```
I(n1, n2)
```

where `I` is the access function for *the flow nature* of signals of type `electrical`. The relationship between the **`discipline`** definition and the access functions of the **`nature`** definition are elaborated in more detail later in Section 2.5.2 on implicit branches.

2.5.2 Implicit Branches

From the two nodes, `n1` and `n2`, we can form an implicit branch[1]. To access the potential across the branch, refer to the **`potential`** binding in the `electrical` discipline definition. The **`potential`** of `electrical` is bound to the nature definition of `Voltage`, which defines an access function of `V`. Hence, to access the potential across the branch, use:

```
V(n1, n2)
```

In other words, accessing the potential from a node or port to a node or port defines the implicit branch. Accessing the potential on a single node or port defines an implicit branch from the node or port to ground. So,

```
V(n1)
```

1. A branch is formed implicitly by use within the behavioral definition.

accesses the potential on the implicit branch from `n1` to ground. Likewise, the **`flow`** of `electrical` is bound to the nature definition of `Current`, which defines an access function of `I`. Hence, to access the **`flow`** through the branch from `n1` to `n2`, use:

```
I(n1, n2)
```

and,

```
I(n1)
```

accesses the flow on the implicit branch from `n1` to ground.

2.5.3 Summary of Signal Access

The following table shows how access functions can be applied to nodes and ports. In this table, `n1` and `n2` represent either nodes or ports belonging to the `electrical` discipline.

Example	Access Description
`V(n1)`	Voltage of `n1` and ground (node or port).
`V(n1, n2)`	Voltage difference between `n1` and `n2` (nodes or ports)
`I(n1)`	Current flowing from `n1` (node or port) to ground.
`I(n1, n2)`	Current flowing between `n1` and `n2`.

The arguments to an access function must be a list of one or two nodes or port identifiers. If two node identifiers are given as arguments to an access function, they must not be the same identifier. The access function name must match the discipline declaration for the nodes or ports given in the argument expression list.

2.6 Probes, Sources, and Signal Assignment

The Verilog-A language uses what is referred to as the *probe-source formulation* for describing analog behaviors. The probe-source formulation is a simple (and probably familiar) concept that allows for describing high-level behaviors. The contribution operator "<+", in conjunction with the probe-source formulation, form the basis for the description of analog behaviors in the Verilog-A language. The contribution oper-

ator is only valid within the **analog** block. Contribution statements are statements that use the contribution operators to describe behavior in terms of the mathematical relationships of input signals to output signals.

In general, a contribution statement consists of two parts, a left-hand side, and a right-hand side separated by the contribution operator. The right-hand side can be any expression that evaluates to a real value. The left-hand side specifies the source branch signal that the right hand side is to be assigned. It must consist of an access function applied to a branch. Hence, analog behaviors can be described using:

```
V(n1, n2) <+ expression;
```

or

```
I(n1, n2) <+ expression;
```

where `(n1, n2)` represents an implicit source branch, and the `V(n1, n2)` access function refers to the potential across the branch while the `I(n1, n2)` access function refers to the flow through the branch. Access functions within `expression` can be used within linear, nonlinear, algebraic, or dynamic functions and become probes in the equation formulation.

Branch contribution statements implicitly define source branch relations. The branch is directed from the first node of the access function to the second node. If the second node is not specified, the second node is taken as the ground or the reference node.

2.6.1 Probes

Probes are formed from access functions whenever the access function is used in an expression. If the flow of the branch is used in an expression anywhere in the module,

the branch is a flow probe, otherwise the branch is a potential probe. The models for probe branches are shown in Figure 2.19.

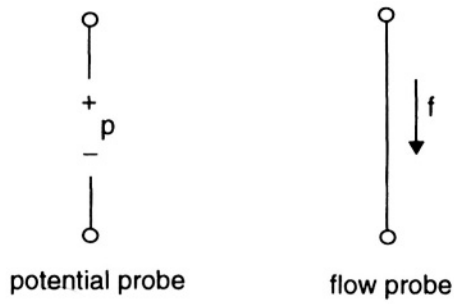


FIGURE 2.19 Schematic representations of equivalent circuits for potential and flow probes.

The branch potential of a flow probe is zero. The branch flow of a potential probe is zero. When a flow probe is introduced between two nodes, it introduces a zero-impedance path between the nodes. Conversely, a potential probe introduces an infinite impedance path between the two nodes it connects.

2.6.2 Sources

Sources are formed from access functions whenever the access function appears as a target on the left-hand side of the contribution operator, "<+". It is a potential source if the branch potential is specified, and it is a flow source if the branch flow is specified. For the following,

```
V(n1, n2) <+ .... ;
I(n3, n4) <+ .... ;
```

define a potential source on the implicit branch between nodes `n1` and `n2`, and a flow source between the implicit branch between nodes `n3` and `n4`.[1]

1. A branch cannot simultaneously be both a potential and a flow source, though it can switch between them, in which case it is referred to as a *switch branch.*

Both the potential and the flow of a source branch are accessible in expressions anywhere in the module. The models for potential and flow sources are shown in Figure 2.20, where `f` is a probe that measures the flow through the branch, and `p` is a probe

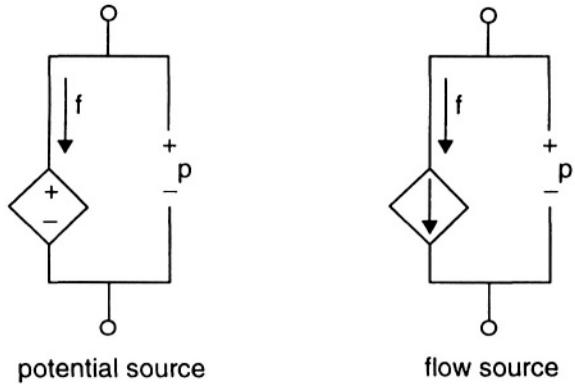


FIGURE 2.20 Schematic representations of equivalent circuits for potential and flow sources.

that measures the potential across the branch. Hence, both potential and flow sources can assign as well as measure quantities across their respective branches.

2.6.3 Illustrated Examples

The concepts of probes and sources in the Verilog-A language is readily illustrated using examples of the controlled sources that are the staple of behavioral modeling in Spice. These include the current- and voltage-controlled current and voltage sources. The Verilog-A module definitions of these elements, with their corresponding network representations, are shown in Figures 2-21 thru 2-24.

```
module vccs(p,n,pc,nc);
   inout p,n,pc,nc;
   electrical p,n,pc,nc;

   parameter real g = 1.0;
   analog
      I(p,n) <+ g*V(pc,nc);

endmodule
```

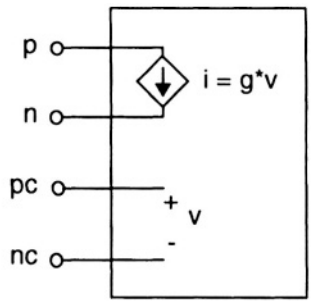


FIGURE 2.21 Module definition of a voltage-controlled current source and schematic representation.

```
module cccs(p,n,pc,nc);
   inout p,n,pc,nc;
   electrical p,n,pc,nc;

   parameter real g = 1.0;
   analog
      I(p,n) <+ g*I(pc,nc);

endmodule
```

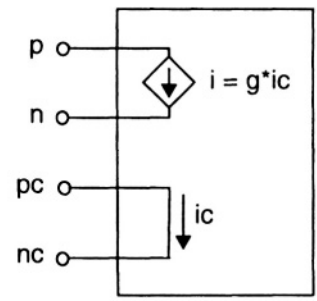


FIGURE 2.22 Module definition of a current-controlled current source and schematic representation.

```
module vcvs(p,n,pc,nc);
   inout p,n,pc,nc;
   electrical p,n,pc,nc;

   parameter real g = 1.0;
   analog
      V(p,n) <+ g*V(pc,nc);

endmodule
```

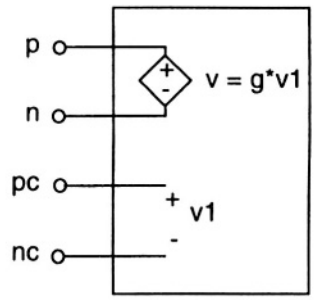


FIGURE 2.23 Module definition of a voltage-controlled voltage source and schematic representation.

```
module ccvs(p,n,pc,nc);
   inout p,n,pc,nc;
   electrical p,n,pc,nc;

   parameter real g = 1.0;
   analog
      V(p,n) <+ g*I(pc,nc);

endmodule
```

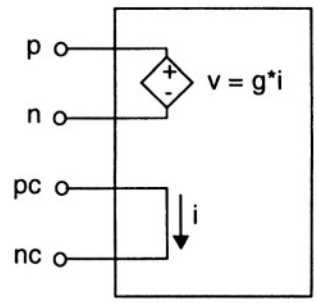


FIGURE 2.24 Module definition of a voltage-controlled current source and schematic representation.

2.7 Analog System Simulation

Analog simulation involves solving systems of ordinary differential equations that describe the system. The system of equations that define the circuit is of the

form: $F(\dot{x}, x, u) = 0$, where x is a vector representing the unknowns, $\dot{x}$ the time rate of change of the unknowns, and u is a vector representing the external stimulus to the system. The system of equations is derived from both the *behaviors* describing the components and the *interconnection* or structure of the components as shown in Figure 2-25.

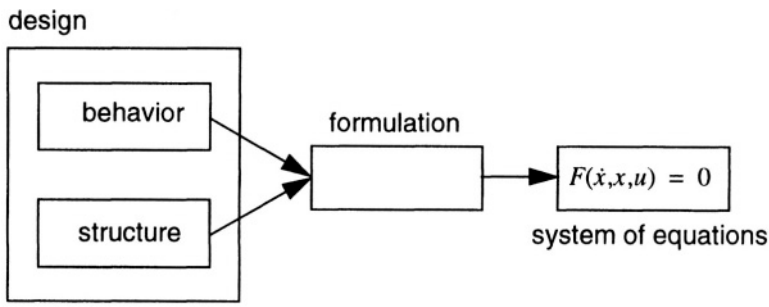


FIGURE 2.25 The formulation of a system of equations from the behavioral and structural aspects of a design using via formulation (based on KPL and KFL for conservative systems).

Simulation of an analog system requires an analysis of all nodes in the system to develop the equations that define the complete set of potentials and flows in the network. During transient analysis, these equations are solved incrementally with respect to time. Given the nonlinear nature of the system, at each time increment, equations are iteratively solved until they converge to a solution in which error criteria is satisfied.

The standard approach to analog circuit simulation involves:

- Formulate the differential-algebraic equations for the circuit
- Applying implicit integration methods to the sequence of nonlinear algebraic equations
- Iterative methods such as Newton-Raphson to reduce the problem to a sparse set of linear equations
- Using sparse-matrix techniques, solve the system of linear equations

2.7.1 Convergence

To determine when iterative methods such as Newton-Raphson have converged to a sufficiently accurate solution, tolerance criteria is used. Tolerance criteria can be applied to conservative systems in two ways:

- Solution between the current and previous iterations at a time point converge, i.e.,

$$|x_k - x_{k-1}| < reltol \cdot max(|x_k|, |x_{k-1}|) + x_{abstol}$$

- Conservation of the constraints on charges, fluxes, currents, potentials, etc. within the system for KFL and KPL are satisfied:

$$|F(x_k)| < reltol \cdot \max(|f_i|) + f_{abstol}$$

where x_k is the k^{th} iteration of the solution for *x*. The `reltol` within the convergence criteria, is a global option of the simulation. The x_{abstol} and f_{abstol} are associated with the type, or **`nature`** of the unknown *x*.

CHAPTER 3

Behavioral Descriptions

3.1 Introduction

The Verilog-A language allows analog and mixed-signal systems to be described by a set of components or modules and the signals that interconnect them. Modules are the fundamental user-defined primitive in the Verilog-A language. Modules can be defined structurally and/or behaviorally. The designer is free to choose the level of abstraction within the structural or behavioral components of the definition of the module. Once a module is defined, it can be instantiated as a child module within a parent module definition or as a top-level component in the design.

The behavioral description is done in a programmatic fashion in the Verilog-A language within the **`analog`** statement or block of the module. The **`analog`** block encapsulates the behavioral description, or the mathematical relationships between input signals and output signals. A behavioral model of the component allows the designer a degree of freedom over the amount of detail to be used in the design for architectural exploration and verification.

This chapter overviews the basic statements and expressions used the description of analog behaviors with the Verilog-A language. A concise description of the mathematical, trigonometric, and hyperbolic functions available for describing analog behaviors are listed in Appendix A. A special class of expression-level constructs

known as analog operators in the Verilog-A language are introduced and illustrated via example.

3.2 Behavioral Descriptions

The Verilog-A language provides the fundamental capabilities necessary for describing the behavior of modules comprising analog and mixed-signal systems. Within a module definition, the analog behavioral descriptions are encapsulated within **analog** statements or blocks:

```
analog begin
    <behavioral_statements>
end
```

The **analog** statement encapsulates a *large-signal* behavior for the model valid for all time. The large-signal model of a component is the behavior expressed in the time domain. From this large-signal definition of the model, representations required for the simulation of other types of analyses can be derived (Figure 3.1). For example, the linearization of the large-signal model about its operating point allows small-signal AC analysis to be performed[1].

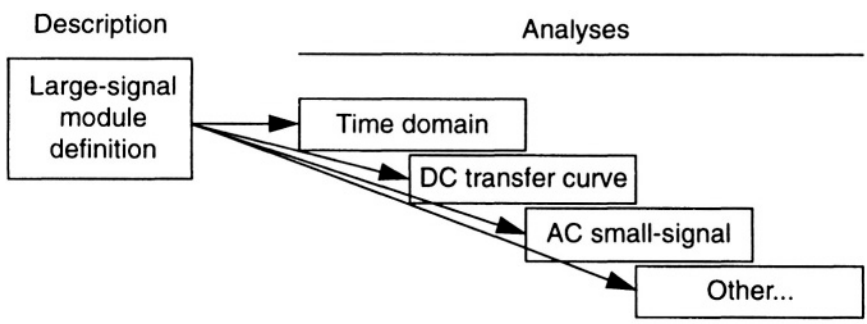


FIGURE 3.1 Mapping of a modules' behavioral descriptions into representations suitable for different types of analyses.

1. Linearization of the large-signal model about the operating point is the same technique utilized by Spice.

The behavioral statements within the **analog** block can include control-flow or looping constructs for defining the behavior of the module. These statements are similar to those found in many programming languages. Additionally, the Verilog-A language provides different language constructs that can be utilized for representing equivalent behavior. For example, in Figure 3.2 three different ways of formulating the behavioral construct for assigning the maximum of two values to a variable are presented[1]. All of these are equivalent representations using statement, expression,

statement

```
if (x > y)
   o = x;
else
   o = y;
```

expression

```
o = (x > y) ? x : y;
```

functional

```
o = max(x, y);
```

FIGURE 3.2 Equivalent statement, expression, and functional Verilog language representations of o = f(x, y).

and functional techniques respectively for determining the maximum value and assigning it to a variable.

The Verilog-A language introduces a class of behavioral constructs known as *analog operators* that are used in defining the large-signal behavioral characteristics of the module. Because of mathematical and other properties associated with analog operators, there are special considerations in their usage. These and other issues associated with the use of analog operators will be discussed in Section 3.4.

3.2.1 Analog Model Properties

The behavioral descriptions with the Verilog-A language can be used to represent different types of behaviors. These can include:

- Linear
- Nonlinear
- Piecewise linear

1. There is at least one more, albeit more obscure, equivalent statement method of determining the maximum of two values in the Verilog language.

- Integro-differential
- Event-driven analog

or combinations thereof. All behavioral models rely on the understanding of the modeler in terms of the formulation of the model and the models' valid regions of operation[1]. The model must also show stable and/or continuous behavior between its' various regions of operation. For example, the behavior of a resistor connected between two `electrical` type nodes `p` and `n` could be represented in one of two ways. First as,

```
V(p, n) <+ res*I(p, n);
```

or,

```
I(p, n) <+ V(p, n)/res;
```

From a perspective of the description of behavior for a resistor, the two formulations are equivalent. However, note that for the condition,

```
res == 0.0
```

the first formulation handles this case correctly, but the second generates a divide by zero. Similar conditions can occur for the signals from which the behavior is formulated. For example, a voltage divider could be represented behaviorally as,

```
V(out) <+ V(numer)/V(denom);
```

If `V(denom)` ever goes to zero, which can happen during the course of a simulation or at initialization, a similar divide by zero condition can occur. It is up to the modeler, with an understanding of the task at hand, to insure that the model is mathematically valid.

The model developer must also insure the model is stable or well-behaved. This property is most obvious in the continuity in both time and value that the model shows. For example,

```
if (x > 2.5)
   V(out) <+ 5.0;
```

1. The iterative nature of the solution of the system of equations used in analog simulation essentially dictates that the model be valid or well-behaved for *any* potential region of operation or input signal values.

```
else
   V(out)  <+ 0.0;
```

for the variable x as some arbitrary function of time, is discontinuous at the output about the condition `x == 2.5` for `V(out)`, in both time and value. This may or may not be a problem, depending upon the type of network to which the output signal, `V(out)` is attached. For resistive loads, these types of discontinuities do not present problems. However, for capacitive or inductive loads, this type of behavior will potentially cause problems for the simulation. The Verilog-A language provides capabilities for the model developer to effectively handle such cases but still relies on the developer for recognizing and utilizing these capabilities.

The mathematical validity and stability of the formulation of a model are important issues to consider when developing a behavioral model, particularly during the test and validation of the model.

3.3 Statements for Behavioral Descriptions

In the Verilog-A language, all analog behavior descriptions are encapsulated within the **`analog`** statement. The **`analog`** statement encompasses the contribution statement(s) that are used to define the relationships between the input and output signals of the module. Statements within the Verilog-A language allows these contribution statements used in defining the analog behaviors to be sensitive to procedural and/or timing control.

This section describes the statements used in formulating analog behavioral descriptions.

3.3.1 Analog Statement

The **`analog`** statement is used for defining the behavior of the model in terms of contribution statements, control-flow, and/or analog event statements. All the statement(s) comprising the **`analog`** statement are evaluated at each point during an analysis. The **`analog`** statement is the keyword **`analog`** followed by a valid Verilog-A statement.

```
analog
   <statement>
```

Where <statement> is a single statement in the Verilog-A language as in the module `resistor` of Listing 3.1.

LISTING 3.1 Resistor module illustrating a single statement attached to the **analog** statement.

```
module resistor(p, n);
   inout p, n;
   electrical p, n;

   parameter real res = 1.0;

   analog
      V(p, n) <+ res*I(p, n);

endmodule
```

The statement attached to an **analog** statement is usually a *block statement* delimited by a **begin**–**end** pair.

```
analog begin
   <statements>
end
```

The block or compound statement defines the behavior of the module as a procedural sequence of statements. The block statement is a means of grouping two or more statements together so that they act syntactically like a single statement. For example, the module `resistor` of Listing 3.1 could be re-written using a block statement as in Listing 3.2.

LISTING 3.2 Resistor module illustrating a block statement attached to the **analog** statement.

```
module resistor(p, n);
   inout p, n;
   electrical p, n;

   parameter real res = 1.0;
   real volts;
```

```
    analog begin
       volts = res*I(p, n);
       V(p, n) <+ volts;
    end

endmodule
```

The group of statements within the **analog** block are processed sequentially in the given order and at each timepoint during a transient simulation. This aspect of the Verilog-A language allows the module developer the ability to define the flow of control within the behavioral description[1] .

Statements of any block statement are guaranteed to be evaluated if the block statement is evaluated. This property, in conjunction with properties of analog behaviors described in the Verilog-A language to be discussed in Section 3.4, has implications in the formulation of the analog behaviors for stability and robustness.

3.3.2 Contribution Statements

The *contribution statements* within the **analog** block of a module form the basis of the behavioral descriptions used to compute flow and potential values for the signals comprising the analog system. The behavioral or large-signal description is the mathematical relationships of the input signals to output signals. In the probe-source model described in Section 2.6, the relationships between input and output signals is done with contribution statements of the form:

```
output_signal <+ f(input_signals);
```

Where `output_signal` is a branch potential or flow source that is the target of the contribution operator (<+) assigned by the value of the right-hand side expression, `f (input_signals)`. For example,

```
V(pout1, nout1) <+ expr1;
I(pout2, nout2) <+ expr2;
```

1. The evaluation of the entire group of statements within the **analog** block at every timepoint is a departure from the semantics of the **always** statement in digital Verilog. In digital Verilog, the evaluation of the behavioral model is determined by monitoring and blocking on events of the (digital) signals.

are examples of potential and flow branch contributions respectively. The right-hand side expressions, `expr1` and `expr2`, can be any combination of linear, nonlinear, algebraic, or differential expressions of module signals, constants and parameters.

A contribution statement is formed such that the output is isolated[1]. For example, given the following transfer function for $H(s)$:

$$Y(s) = H(s) \cdot X(s)$$

$$Y(s) = \frac{R}{R - s} \cdot X(s)$$

the transfer function relationship can be formulated in terms of the output, $y(t)$, for the large-signal response as,

$$y(t) = \frac{1}{R} \cdot \frac{d}{dt} y(t) + x(t)$$

from which, the behavioral relationship can be expressed in the Verilog-A language contribution statement as

```
V(y)  <+ ddt(V(y))/R + V(x);
```

Where `V(y)`, the potential of the signal `y`, or $y(t)$ and `V(x)` is the potential of the signal `x`, or $x(t)$. Note that Only a potential or flow source branch can be the target of a contribution operator, i.e., no **`real`** or **`integer`** variables.

3.3.3 Procedural or Variable Assignments

In the Verilog-A language, branch contributions and indirect branch contributions[2] are used for modifying signals. The procedural assignments are used for modifying integer and real variables. A procedural assignment in the Verilog-A language is similar to that in any programming language:

1. The probe-source formulation does not require that the output *cannot* also appear on the right-hand side of the contribution operator. In addition, an alternative equation formulation construct is presented in Section 3.6.2 for such cases when it is not easy to isolate the output.
2. Described later in section 3.6.

```
real x;
real y[1:12];

analog begin
   ...
   x = 5.0;
   y[i] = x;
   ...
end
```

In general, the left-hand side of the assignment must be an integer or a real identifier or a component of an integer or real array. The right-hand side expression can be any arbitrary expression constituted from legal operands and operators in the Verilog-A language.

3.3.4 Conditional Statements and Expressions

The Verilog-A supports two primary methods of altering control-flow within the behavioral description of a module which are the conditional statement and the ternary or **?:** operator. The control-flow constructs within the Verilog-A language are used for defining piece-wise behaviors (linear or nonlinear). The conditional statement (or **if-else** statement) is used to make a decision as to whether a statement is executed or not. The syntax of a conditional statement is as follows:

```
if ( expr )
   <statement>
else
   <statement>
```

where the **else** branch of the **if-else** statement is optional. If the expression evaluates to true (that is, has a non-zero value), the first statement will be executed. If it evaluates to false (has a zero value), the first statement will not be executed. If there is an **else** statement and expression is false, the **else** statement will be executed.

As previously described, the **if-else** statement can be used to define an analog behavior that determines the maximum of two input signals (or values) as in Listing 3.3.

LISTING 3.3 Module definition illustrating use of **if-else** statements.

```
module maximum(out, in1, in2);
    inout out, in1, in2;
    electrical out, in1, in2;

    real vout;

    analog begin
        if (V(in1) > V(in2))
            vout = V(in1);
        else
            vout = V(in2);

        V(out) <+ vout;
    end

endmodule
```

Because the **else** <statement> part of an **if**-**else** is optional, there can be confusion when an **else** is omitted from a nested if sequence. This is resolved by always associating the **else** with the closest previous **if** that lacks an **else**. In Listing 3.4, the **else** goes with the inner **if**, as shown by indentation.

LISTING 3.4 Proper association of **else** <statement> within a nested **if**.

```
if ( expr1 )
    if ( expr2 )
        <statement2a>
    else
        <statement2b>
```

If that association is not desired, a **begin**-**end** block statement must be used to force the proper association, as shown in Listing 3.5.

LISTING 3.5 Forced association of an **else** <statement> using a block statement.

```
if ( expr1 ) begin
    if ( expr2 )
        <statement2>
end else
    <statement1b>
```

The ternary operator (**?:**) can be used in place of the **if** statement when one of two values is to be selected for assignment. The general form of the expression is:

```
conditional_expr ? expr1 : expr2
```

If the `conditional_expr` is non-zero, then the value of the ternary expression is `expr1`, else the value is `expr2`. The maximum module definition of Listing 3.3 can be written much more compactly using the ternary operator as in Listing 3.6.

LISTING 3.6 Module definition illustrating use of ternary operator.

```
module maximum(out, in1, in2);
    inout out, in1, in2;
    electrical out, in1, in2;

    analog
       V(out) <+ ((V(in1) > V(in2)) ? V(in1) : V(in2));

endmodule
```

The distinction between the **if-else** and the ternary operator is that the ternary operator can appear anywhere an expression is valid in the Verilog-A language. Conversely, the **if-else** statement can only appear in the body of an **analog** or a block statement.

3.3.5 Multi-way Branching

The Verilog language provides two ways of creating multi-way branches in behavioral descriptions; the **if-else-if** and the **case** statements. The most general way of writing a multi-way decision in Verilog-A is with an **if-else-if** construct as illustrated in Listing 3.7.

LISTING 3.7 Multi-way branching using the **if-else-if** statement construct.

```
if ( expr1 )
   <statement1>
else if ( expr2 )
   <statement2>
else
   <statement3>
```

The expressions are evaluated in order; if any of the expressions are true (`expr1`, `expr2`), the statement associated with it will be executed, and this will terminate the whole chain. Each statement is either a single statement or a sequential block of statements. The last **`else`** part of the **`if-else-if`** construct handles the none-of-the-above or default case where none of the other conditions are satisfied. Sometimes there is no explicit action for the default; in that case, the trailing **`else`** statement can be omitted or it can be used for error checking to catch an unexpected condition.

For example, the behavior of a dead-band amplifier (Figure 3.3) using the **`if-else-`**

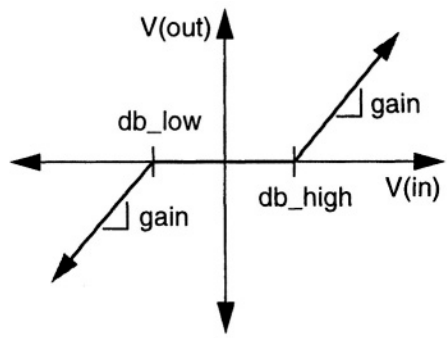


FIGURE 3.3 Input-output relationship for a dead-band amplifier.

`if` construct, the behavior can be represented in the Verilog-A language as in Listing 3.8.

LISTING 3.8 Dead-band amplifier behavior using the **`if-else-if`** statement construct.

```
analog begin
   if (V(in) >= db_high)
      vout = gain*(V(in) - db_high);
   else if (V(in) <= db_low)
      vout = gain*(V(in) + db_low);
   else
      vout = 0.0;

   V(out) <+ vout;
end
```

Note that the variable `vout`, will be piece-wise continuous in value across the range of `V(in)`.

3.4 Analog Operators

Analog operators in the Verilog-A language are used for formulating the large-signal behavioral descriptions of modules. Used in conjunction with the standard mathematical and transcendental functions (Appendix A), with analog operators the modeler can define the components constitutive behavior. Similar to functions, analog operators take an expression as input and return a value. However, analog operators differ in that they maintain internal state and their output is a function of both the current input and this internal state.

The Verilog-A language defines analog operators for:

- Time derivative
- Time integral
- Linear time delay
- Discrete waveform filters
- Continuous waveform filters
- Laplace transform filters
- Z-transform filters

3.4.1 Time Derivative Operator

The **`ddt`** operator computes the time derivative of its argument.

```
ddt(expr)
```

$$\frac{d}{dt}(\text{expr})$$

FIGURE 3.4 Prototype of **`ddt`** time derivative analog operator and mathematical representation.

In DC analysis, **ddt** returns zero. Application of the **ddt** operator results in a zero at the origin. Consider the example module definition of Listing 3.9 taking the time derivative of the input signal.

LISTING 3.9 ddt analog operator example.

```
module ddt_op(out, in);
    inout out, in;
    electrical out, in;
    parameter real scale = 1.0e-6;

    analog
       V(out) <+ scale*ddt(V(in));

endmodule
```

The results of applying a 100KHz sinusoidal signal, with amplitude of 1.0V, to the `in` signal of the module, with `scale` set to its default value of 1.0e-6 are shown in Figure 3.5.

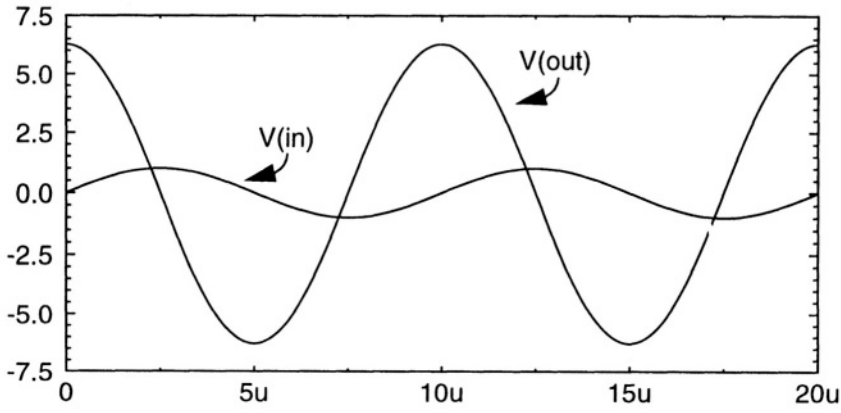


FIGURE 3.5 Time-domain analysis with the **ddt** analog operator of Listing 3.9.

It is important to consider the input signal characteristics when doing when using the **ddt** operator (as with all analog operators). Without setting the parameter `scale` to 1.0e-6, the output of the module would have been 6.28e6 volts with the same input

signal applied. The model developer should be aware that when differentiating an unknown input signal, a fast varying 'noise' component can dominate the true derivative of the signal of interest.

3.4.2 Time Integral Operator

The **idt** operator computes the time-integral of its argument.

idt(expr, ic, reset)

$$\int_0^t (\text{expr})\, dt + \text{ic}$$

FIGURE 3.6 Prototype of **idt** time integral analog operator and its mathematical representation.

When specified with initial conditions, the **idt** operator returns the value of the initial condition in DC. Without initial conditions, **idt** multiplies it's argument by infinity in DC analysis. Hence, without initial conditions, **idt** *must* be used in a system description with feedback that forces its argument to zero[1] . The optional argument `reset` allows resetting of the integrator to the initial condition or `ic` value. Application of the **idt** operator results in a pole at the origin.

The module definition of Listing 3.10 illustrates the use of **idt** operators with different values of initial conditions specified.

LISTING 3.10 **idt** analog operator example.

```
module idt_op(out1, out2, in);
   inout out1, out2, in;
   electrical out1, out2, in;
   parameter real scale = 1.0e6;

   analog begin
      V(out1) <+ idt(scale*V(in), 0.0);
```

1. Failure to do so will result in a system description that is not solvable - i.e., convergence will not likely be achieved.

```
        V(out2) <+ idt(scale*V(in), 2.0);
    end

endmodule
```

The results of applying `V(in)` as a clock with a pulse period of `50n` to the input of the module of Listing 3.10 which differ only in the initial condition parameter `ic` (0.0 and 2.0), are shown in Figure 3.7. Both integrator modules were applied `scale`

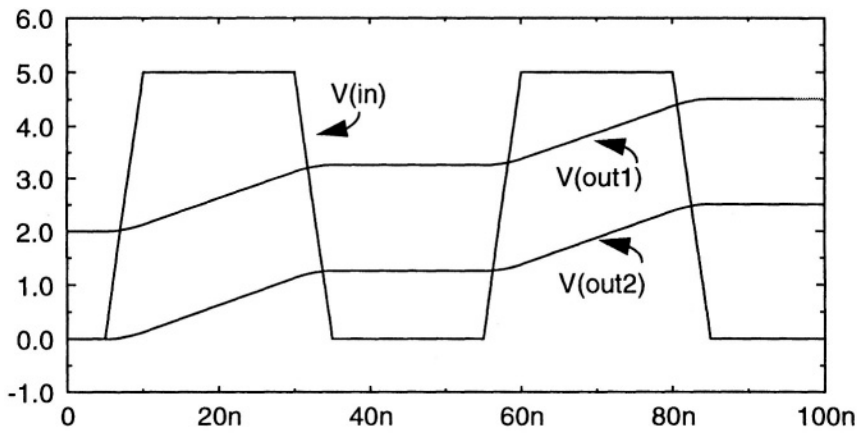


FIGURE 3.7 Time domain analysis with the **idt** analog operator with and without initial conditions specified as per Listing 3.2.

parameter values of 1.0e6.

3.4.3 Delay Operator

The **delay** operator implements a *transport,* or linear time delay for continuous waveforms (similar to a transmission line).

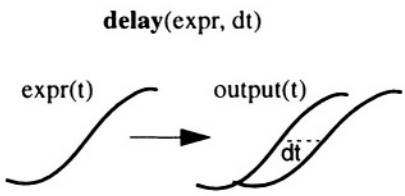


FIGURE 3.8 Prototype of **delay** analog operator and graphical representation.

The parameter `dt` must be nonnegative and any changes to the parameter `dt` are ignored during simulation (the initially specified value for `dt` is used). The effect of the **delay** operator in the time domain is to provide a direct time-translation of the input. An example of the delay analog operator is illustrated in Listing 3.11.

LISTING 3.11 **delay** analog operator example.

```
module delay_op(out, in);
   inout out, in;
   electrical out, in;

   analog
      V(out) <+ delay(V(in), 50n);

endmodule
```

The results of applying a signal `V(in)` (two-tone sinusoidal) to the input of the module of Listing 3.11 is shown in Figure 3.9. For AC small-signal analysis, the delay

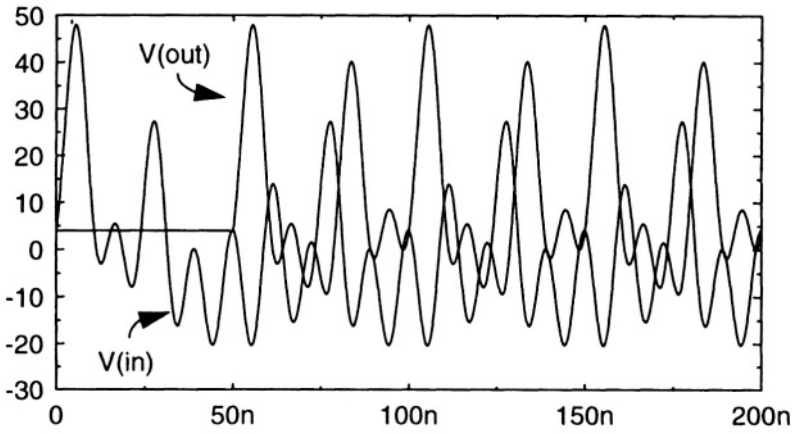


FIGURE 3.9 Time domain analysis with the **`delay`** analog operator.

operator introduces a $e^{-j\omega dt}$ phase shift.

3.4.4 Transition Operator

The **`transition`** operator smooths out piece-wise constant waveforms. The **`transition`** filter is used to imitate transitions and delays on discrete signals.

```
transition(expr, dt, tr, tf)
```

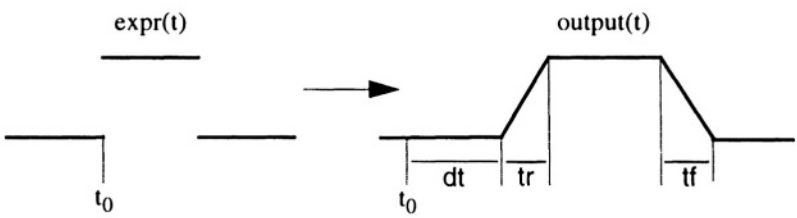


FIGURE 3.10 Prototype of **`transition`** analog operator and graphical representation.

The input `expr` to the **`transition`** operator must be defined in terms of *discrete* states[1]. The parameters `dt`, `tr`, and `tf` are optional to the **`transition`** analog operator. If `dt` is not specified, it is taken to be zero. If only the `tr` value is specified, the simulator uses it for both rise and fall times. In DC analysis, **`transition`** passes the value of the `expr` directly to its output.

Consider the example of Listing 3.12 illustrating the effect of the transition time parameters versus the magnitude of different input step changes.

LISTING 3.12 `transition` operators with different step changes.

```
module transition_op(out1, out2, in);
    inout out1, ou2, in;
    electrical out1, out2, in;

    real vin;
    analog begin
       // discretize the input into two states
       if (V(in) > 0.5)
          vin = 1.0;
       else
          vin = 0.0;

       V(out1) <+ transition(vin, 2n, 5n, 5n);
       V(out2) <+ transition(2*vin, 2n, 5n, 5n);
    end

endmodule
```

The input expression to the **`transition`** operator, `vin`, is a discretization of the input signal and results in the pulse shown in Figure 3.11 with the resulting outputs.

Note that the rise and fall times are independent of the value being transitioned. In addition, the input to transition operators is best kept under the control of the modeler - in this example with a simple **`if-else`** construct is applied to some arbitrary input signal `V(in)` to generate the discrete states that become the input to the **`transition`** operator.

1. For smoothing piece-wise continuous signals see the **`slew`** analog operator.

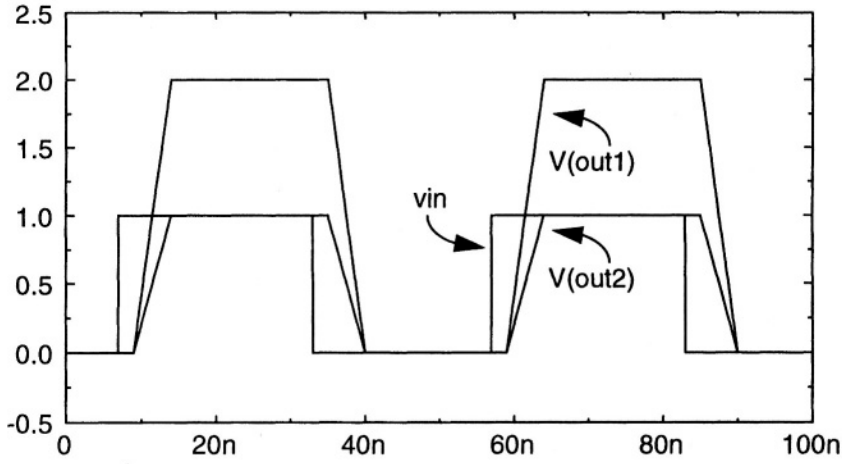


FIGURE 3.11 Time domain response of the **`transition`** analog operator of Listing 3.12 to discontinuous input changes of different magnitudes.

Another characteristic of the **`transition`** operator is exhibited when the rise and fall times are longer than the specified delay. If interrupted on a transition, **`transition`** will try to complete the transition in the specified time.

- If the new final value level is below the value level at the point of the interruption (the current value), **`transition`** uses the old destination as the origin.
- If the new destination is above the current level, the first origin is retained.

In Figure 3.12, a rising transition is interrupted near its midpoint, and the new destination level of the value is below the current value. For the new origin and destination, **`transition`** computes the slope that completes the transition from the origin (not

the current value) in the specified transition time. It then uses the computed slope to transition from the current value to the new destination.

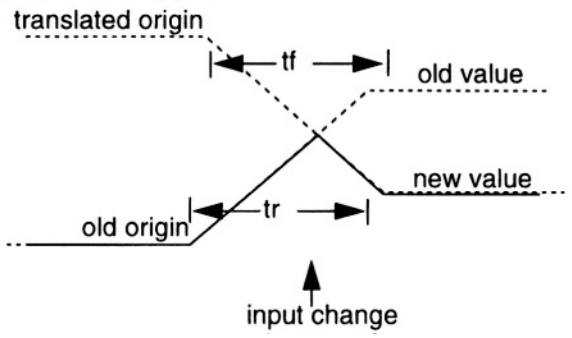


FIGURE 3.12 Response of the transition analog operator to a quickly-varying input (less than `tr` and/or `tf`).

Taking the module definition of Listing 3.12, the input signal is changed to create pulses of shorter duration. The result is shown in Figure 3.13.

Because the **`transition`** function cannot be linearized in general, it is not possible to accurately represent a **`transition`** function in AC analysis. The AC transfer function is approximately modeled as having unity transmission for all frequencies in all situations. Because the transition function is intended to handle discrete-valued signals, the small signals present in AC analysis rarely reach **`transition`** functions.

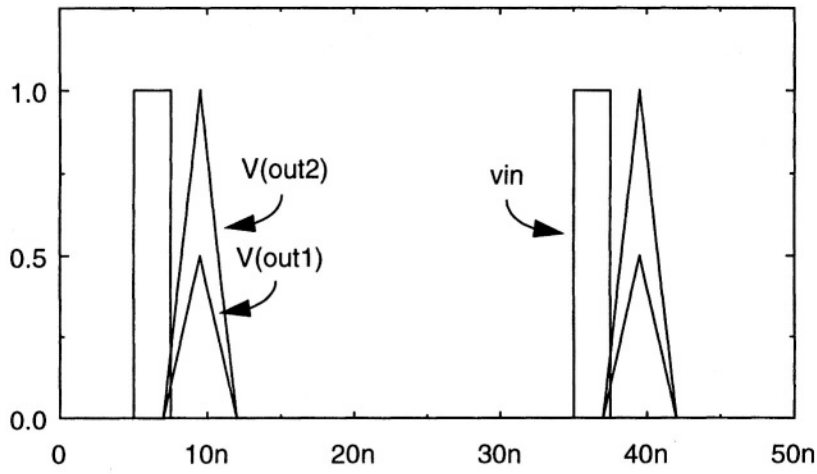


FIGURE 3.13 Time domain response of the **`transition`** analog operator of Listing 3.12 to discontinuous input changes of different magnitudes as in Figure 3.12 but with pulse widths shorter than the rise and fall times of the **`transition`** operator.

3.4.5 Slew Operator

The **`slew`** operator bounds the rate of change (slope) of the waveform. A typical use for **`slew`** is to generate continuous signals from piece-wise continuous signals[1].

slew(expr, mpsr, mnsr)

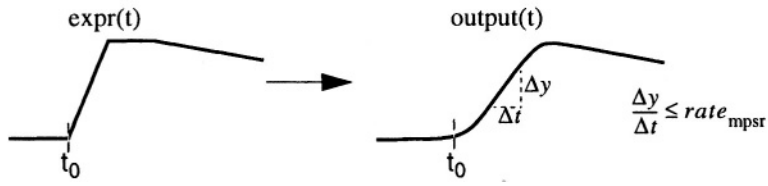


FIGURE 3.14 Prototype of **`slew`** analog operator and graphical representation.

When applied, **slew** forces all transitions of the input expr faster than mpsr to change at mpsr for positive transitions and limits negative transitions to mnsr. The mpsr and mnsr arguments are optional. The parameter mpsr must be greater than 0 and mnsr must be less than 0. If only one rate is specified, its absolute value is used for both rates. If no rates are specified, **slew** passes the signal through unchanged. If the rate of change of expr is less than the specified maximum slew rates, slew returns the value of expr.

Consider the following example for the effect of different slew rates on the **slew** analog operator:

LISTING 3.13 **slew** analog operators with different slew rates.

```
module slew_op(out1, out2, in);
    inout out1, out2, in;
    electrical out1, out2, in;

    analog begin
       V(out1) <+ slew(V(in), 5e8, -5e8);
       V(out2) <+ slew(V(in), 1e9, -1e9);
    end

endmodule
```

The results of applying a sinusoid of 5 Vpp and a frequency of 25MHz (which defines a maximum slew rate of about 1.6e9 V/s) to the signal in are shown in Figure 3.15.

In DC analysis, **slew** simply passes the value of the destination to its output. In AC small-signal analyses, the **slew** function has unity transfer function except when slewing, in which case it has zero transmission through the **slew** operator.

1. For discrete-valued signals see **transition.**

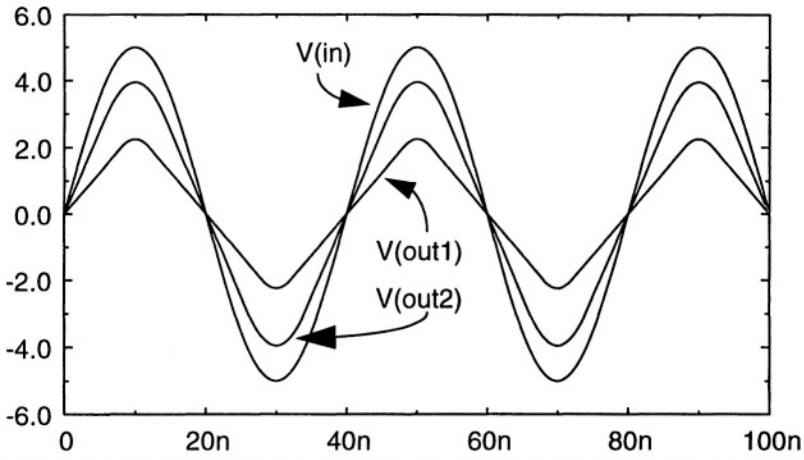


FIGURE 3.15 Time domain analysis of **slew** analog operator with differing slew rates specified as in Listing 3.4.

3.4.6 Laplace Transform Operators

The Laplace transform operators implement lumped, continuous-time filters.

laplace_zp(expr, numer, denom)
laplace_zd(expr, numer, denom)
laplace_np(expr, numer, denom)
laplace_nd(expr, numer, denom)

$$H(s) = \frac{N(s)}{D(s)}$$

FIGURE 3.16 Prototypes of different forms of the laplace analog operators and the transfer function representation.

The laplace transform analog operators take *vector* arguments that specify the coefficients of the filter. The vectors `numer` and `denom` represent the numerator and

denominator of the transfer function of the filter. The numerator or denominator can be expressed in the forms:

- **laplace_zp** in which the zeros and poles of the filter are specified as pairs of real numbers, specifying the real and imaginary components of each zero or pole.
- **laplace_nd** in which the zeros and poles of the filter are specified as polynomial coefficients from lowest order term to the highest.
- **laplace_zd** in which the zeros of the filter are specified as pairs of real numbers, specifying the real and imaginary components of each zero. The poles of the filter are specified as polynomial coefficients from lowest order term to the highest.
- **laplace_np** in which the zeros of the filter are specified as polynomial coefficients from the lowest order term to the highest. The poles of the filter are specified as pairs of real numbers, specifying the real and imaginary components of each pole.

These different forms of specifications for the numerator and denominator allow for four different variants on specifying the filter coefficients. [1] All of the laplace analog operators represent linear time-invariant (LTI) filters which require that the values of the filter coefficients cannot change during a simulation. Hence, only numeric literals, parameters or expressions of these are allowed for defining the filter coefficients. The coefficients are arrays specified using the Verilog HDL concatenation operator ({ }) for creating arrays from these scalar constant expressions.

For example, consider the pole locations of a normalized 5'th order Butterworth low-pass filter with a 3-dB bandwidth of 1 rad/s as shown in Figure 3.17.

The Verilog-A numerator-pole laplace operator representation of Butterworth filter would be:

LISTING 3.14 Laplace analog operator example using **laplace_np.**

```
module laplace_op(out, in);
    inout out, in;
    electrical out, in;

    analog
```

1. Appendix C includes references to Matlab scripts which are useful for generating Verilog-A continuous and discrete domain filters from the filter specifications such as pass/stop bands and ripple and sampling rate (for discrete filters).

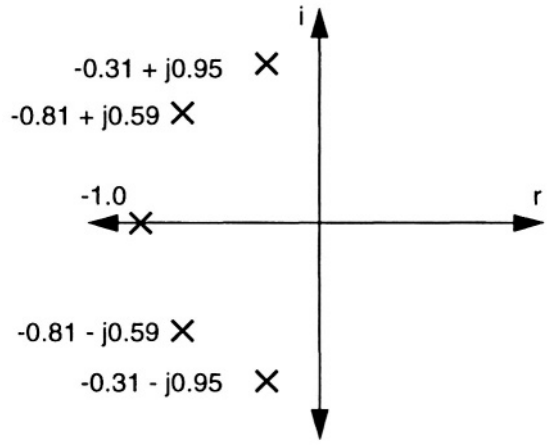


FIGURE 3.17 Pole locations of 5'th order Butterworth low-pass filter.

```
        V(out) <+ laplace_np(V(in), { 1 }, {
                -0.81,  0.59,  -0.81,  -0.59,
                -0.31,  0.95,  -0.31,  -0.95,
                -1.0,  0.0 }) ;

endmodule
```

Note that the real and imaginary pairs for the zeros can be specified in any order. Conversely, the laplace transform operator can be expressed in the polynomial form. The corresponding Butterworth polynomial of the filter of Listing 3.14 is:

$$H(s) = \frac{1}{s^5 + 3.236s^4 + 5.236s^3 + 5.236s^2 + 3.236s + 1}$$

The Butterworth polynomial can be expressed in the polynomial or numerator-denominator form as shown in Listing 3.15.

LISTING 3.15 Laplace analog operator using **laplace_nd**.

```
module laplace_op(out,  in);
    inout  out,  in;
    electrical out, in;
```

```
analog
   V(out) <+ laplace_nd(V(in), { 1.0 },
      { 1.0, 3.236, 5.236, 5.236, 3.236, 1.0 });

endmodule
```

The coefficients are specified from lowest to highest-order term (in this case from s^0 to s^5). The laplace analog operators are valid for both transient and small-signal analyses. Shown in Figure 3.18 is the step response for order = 2 to order = 6 for Butterworth low-pass filters with 3dB bandwidth of 1 rad/s. The Bode plots for the same

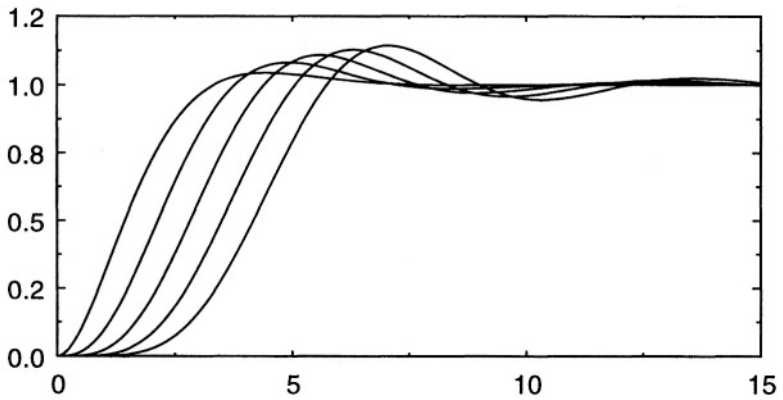


FIGURE 3.18 Time domain step response of Butterworth low-pass filters for orders of 2, 3, 4,5, and 6 using the **laplace_nd** analog operator.

Butterworth are shown in Figure 3.18.

All of the different variants of the laplace analog operators are described in more detail in Appendix C.

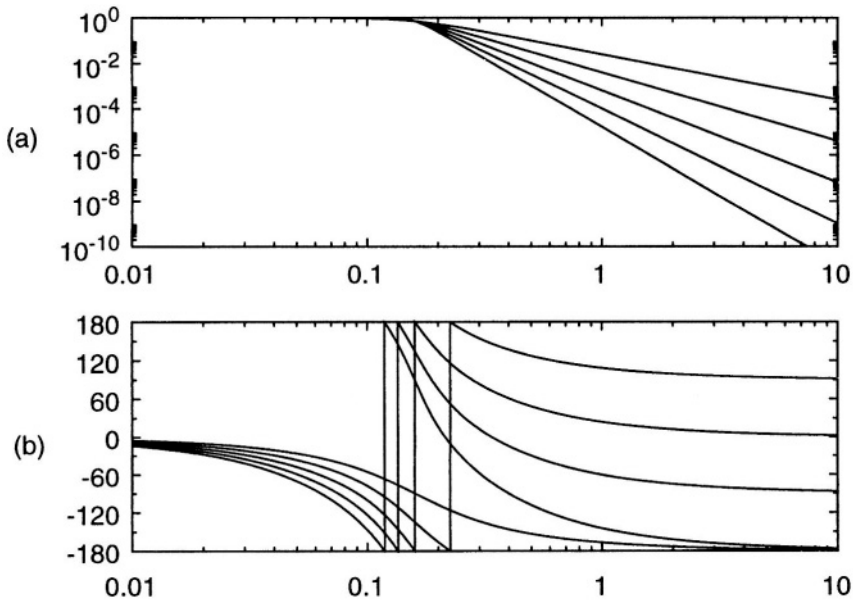


FIGURE 3.19 AC small-signal magnitude (a) and phase response (b) of Butterworth low-pass filters for orders of 2, 3, 4,5, and 6 using the **laplace_nd** analog operator.

3.4.7 Z-Transform Operators

The Z Transform operators implement linear discrete-time filters.

zi_zp(expr, numer, denom, T, trf, t0)
zi_zd(expr, numer, denom, T, trf, t0)
zi_np(expr, numer, denom, T, trf, t0)
zi_nd(expr, numer, denom, T, trf, t0)

$$H(z) = \frac{N(z)}{D(z)}$$

FIGURE 3.20 Prototypes of different forms of the Z-transform analog operators and the transfer function representation.

Like the laplace analog operators, the Z-transform analog operators take vector arguments that specify the coefficients of the filter. The vectors `numer` and `denom` represent the numerator and denominator of the transfer function of the filter. The numerator or denominator can be expressed in the forms:

- **`zi_zp`** in which the zeros and poles of the filter are specified as pairs of real numbers, specifying the real and imaginary components of each zero or pole.
- **`zi_nd`** in which the zeros and poles of the filter are specified as polynomial coefficients from lowest order term to the highest.
- **`zi_zd`** in which the zeros of the filter are specified as pairs of real numbers, specifying the real and imaginary components of each zero. The poles of the filter are specified as polynomial coefficients from lowest order term to the highest.
- **`zi_np`** in which the zeros of the filter are specified as polynomial coefficients from the lowest order term to the highest. The poles of the filter are specified as pairs of real numbers, specifying the real and imaginary components of each pole.

All Z-transform filters share three common arguments, `T`, `trf`, and `to`. The parameter `T` specifies the period of the filter, and is mandatory, and must be positive. A filter with unity transfer function acts like a simple sample-and-hold that samples every `T` seconds and exhibits no delay. For example, the zero-order sample-and-hold of Listing 3.16 is applied a sinusoidal input. The corresponding response is shown in Figure 3.21.

LISTING 3.16 Discrete analog operator using **`zi_nd`**.

```
module discrete_op(out, in);
   inout out, in;
   electrical out, in;

   analog
      V(out) <+ zi_nd(V(in), { 1.0 }, { 1.0 }, 10u);

endmodule
```

Both `tau` and `t0` apply to the output of the discrete filter (and are similar to the characterization of the **`transition`** operator). The parameter `trf` specifies the optional transition time and must be non-negative. If the output transition time `trf` is specified as 0, then the output is abruptly discontinuous. A Z-transform filter with 0 transition time assigned directly to a source branch can generate discontinuities. Finally, `t0` specifies the time of the first transition and is optional. If not given, the first transition occurs at t = 0. Consider the example analog operators in 3.17:

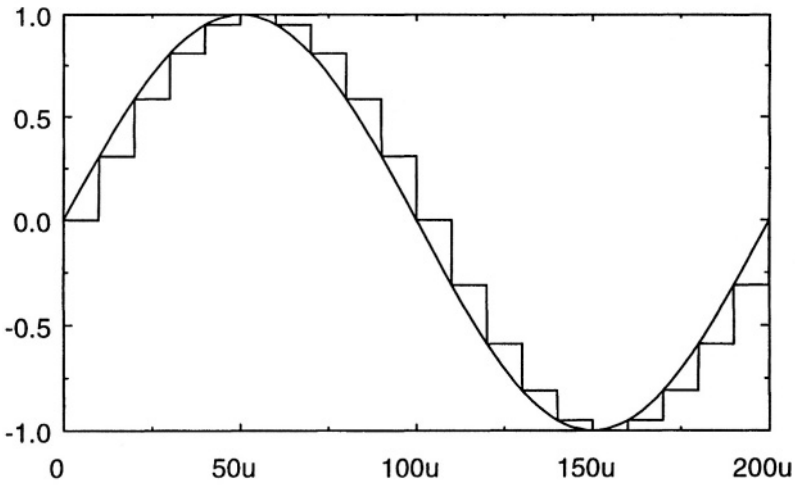


FIGURE 3.21 Time domain sample-and-hold characteristics of a discrete filter with unity transfer function.

LISTING 3.17 Variation of transition parameters for Z-transform operators.

```
module discrete_op(out1, out2, out3, in);
   inout out1, out2, out3, in;
   electrical out1, out2, out3, in;

   analog begin
      V(out1) <+ zi_nd(V(in),
          { 1.0 }, { 1.0 }, 10u);
      V(out2) <+ zi_nd(V(in),
          { 1.0 }, { 1.0 }, 10u, 2u);
      V(out3) <+ zi_nd(V(in),
          { 1.0 }, { 1.0 }, 10u, 2u, 4u);
   end

endmodule
```

The variation of the `trf` and `t0` parameters to the discrete filter is illustrated in Figure 3.22 for the zero-order sample and hold block.

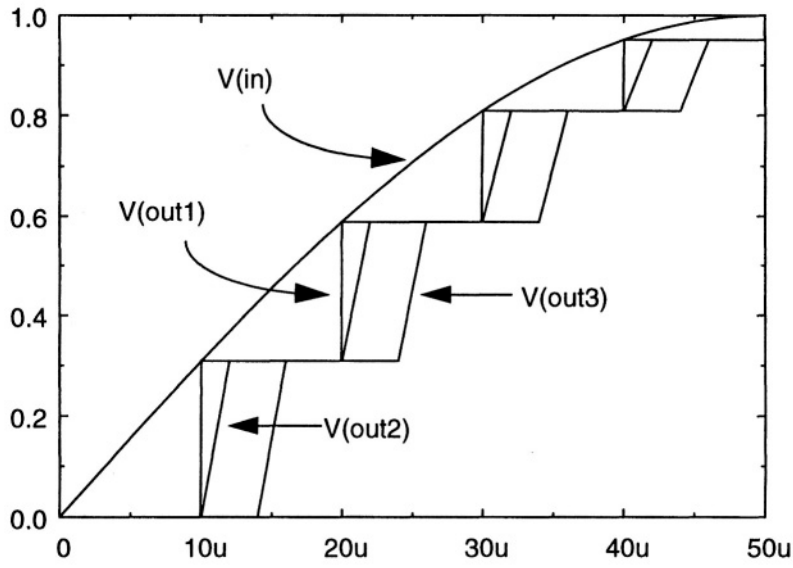


FIGURE 3.22 Variation in the time domain response for discrete analog operators with variations in the output transition and delay parameters as in Listing 3.17.

Similar to the laplace filters, the Z-transform filters are specified in terms of poles and zeros and have representation in both the time and frequency domains. For example, a Chebyshev Type II lowpass filter specified as:

Pass Band = 18KHz
Stop Band = 22KHz
Ripple in Pass Band = 3dB
Ripple in Stop Band = 60dB
Sampling Rate = 100KHz

would be specified in the Verilog-A language as in Listing 3.18.[1]

LISTING 3.18 Verilog-A definition of a Chebyshev Type II lowpass filter.

1. This Verilog-A module was generated using the MATLAB scripts found in appendix C.

```
`include "std.va"
`include "const.va"

// order=10 Chebyshev Type-II Low Pass Filter.

module filter(out, in);
   inout out, in;
   electrical out, in;

   analog begin
      V(out) <+  0.01136415726096424*zi_zp( V(in), {
         0.1756441185917541, 0.9844537285236532,
         0.1756441185917541, -0.9844537285236532,
         -0.9309202894238989, 0.3652224181768157,
         -0.9309202894238989, -0.3652224181768157,
         -0.5369117667994778, 0.8436384027960446,
         -0.5369117667994778, -0.8436384027960446,
         -0.1554308159507361, 0.9878467803525443,
         -0.1554308159507361, -0.9878467803525443,
         0.07433175677690351, 0.9972335683953182,
         0.07433175677690351, -0.9972335683953182
      }, {
         0.3844405110017898, 0.8271728406050872,
         0.3844405110017898, -0.8271728406050872,
         0.01384422707688819, 0.1216675978586536,
         0.01384422707688819, -0.1216675978586536,
         0.07803296529534992, 0.3470777160617312,
         0.07803296529534992, -0.3470777160617312,
         0.1797392266505038, 0.5330681862512341
         0.1797392266505038, -0.5330681862512341,
         0.2881635143364021, 0.6850968689804718,
         0.2881635143364021, -0.6850968689804718
         }, 1.0e-05, 1.0e-7);
      end

endmodule
```

The large-signal transfer characteristics of the filter in Listing 3.18, to the application of a two-tone source, 18KHz and 22KHz, thru the filter is shown in Figure 3.23.

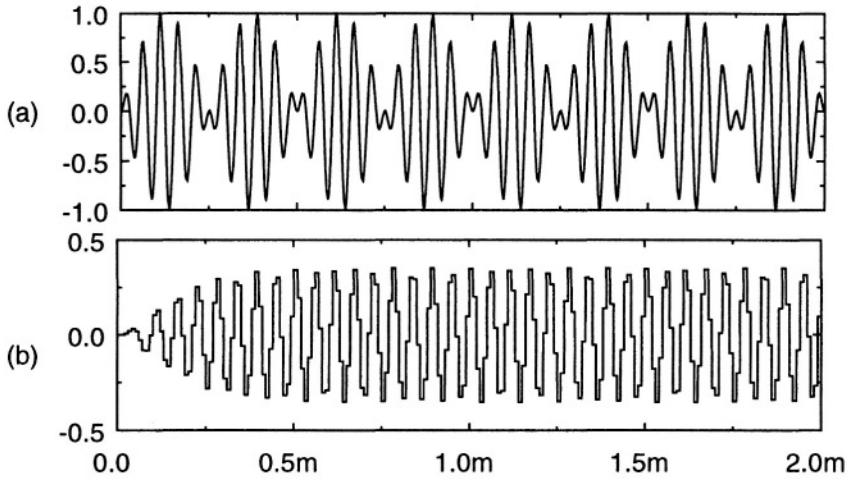


FIGURE 3.23 Input (a) and output (b) of the Chebyshev Type II filter specified in Listing 3.18.

The magnitude response of the same filter exhibits the affects of aliasing at the sampling rate of 100KHz as shown in Figure 3.24.

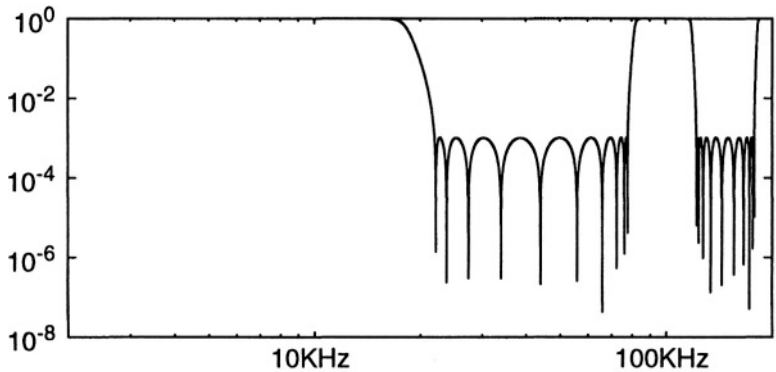


FIGURE 3.24 AC small-signal magnitude of the Chebyshev Type II lowpass filter specified in Listing 3.18.

3.4.8 Considerations on the Usage of Analog Operators

Because analog operators are used in the definition of the large-signal response of the model, they maintain internal state. As such, they are subject to several important restrictions:

- Analog operators can only be used within an analog block.
- Analog operators should not be used inside conditional (e.g., **`if, ?:`** and **`case`**) or looping (**`repeat`**, **`while`**, and **`for`**) statements.

The exception to the latter restriction being that analog operators are allowed if the expression controlling the condition does not change during a simulation or is *statically* defined. Static expressions consist only of expressions consisting of literals, parameters, and the **`analysis`**() function (Section 3.6.1). These restrictions are present to prevent usage that would cause the internal state of the operator to become out-of-date, which can result in inconsistent behavior.[1]

3.5 Analog Events

The analog behavior of a component can be controlled using analog events. The analog events have the following characteristics:

- Analog events can be triggered and detected in the behavioral model
- Analog events do not block the execution of an analog block
- Analog events are detected using the "@" operator

Analog events differ from standard control-flow constructs (**`if`**-**`else`** or **`case`**) in the Verilog-A language in that the event generation and detection requires satisfying accuracy constraints. The accuracy constraints can be either in value or time. The Verilog-A language provides two analog operators for this purpose: **`cross`** and **`timer`**. Detection of an analog event generated by these analog operators requires using the "@" operator. It takes the form:

```
@ ( event_expression ) statement
```

1. These limitations are inherent in any analog HDL. Analog HDLs can enforce this restriction syntactically. The Verilog-A language, however, uses this semantic restriction.

The statement following the event expression is executed whenever the event expression triggers. Analog event detection in the Verilog-A language is non-blocking, meaning that the execution of the statement is skipped unless the analog event has occurred. This non-blocking behavior is a general characteristic of any statement within the **analog** statement.

The event expression consists of one or more monitored events separated by the **or** operator. The "or-ing" of any number of events can be expressed such that the occurrence of any one of the events trigger the execution of the event statement that follows it, as:

```
@(analog_event_1 or analog_event_2)
   <statement>
```

3.5.1 Cross Event Analog Operator

The **cross** event analog operator is used for generating a monitored analog event to detect threshold crossings in analog signals.

```
cross(expr, dir, timetol, valuetol)
```

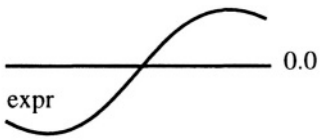


FIGURE 3.25 cross analog operator and its graphical representation.

The **cross** function generates events when the expression argument crosses zero in the specified direction. **cross** controls the timestep to accurately resolve the crossing within a time resolution of `timetol` and value resolution of `valuetol`. Both `timetol` and `valuetol` are optional.

If the direction argument, `dir`, is 0 or not specified, then the event and timestep control occur on both positive and negative crossings of the signal. If the direction indicator is +1 (-1), then the event and timestep control only occurs on positive (negative) transitions of the signal. These cases are illustrated graphically in Figure 3.26. For any other transitions of the signal, the **cross** function does not generate an event.

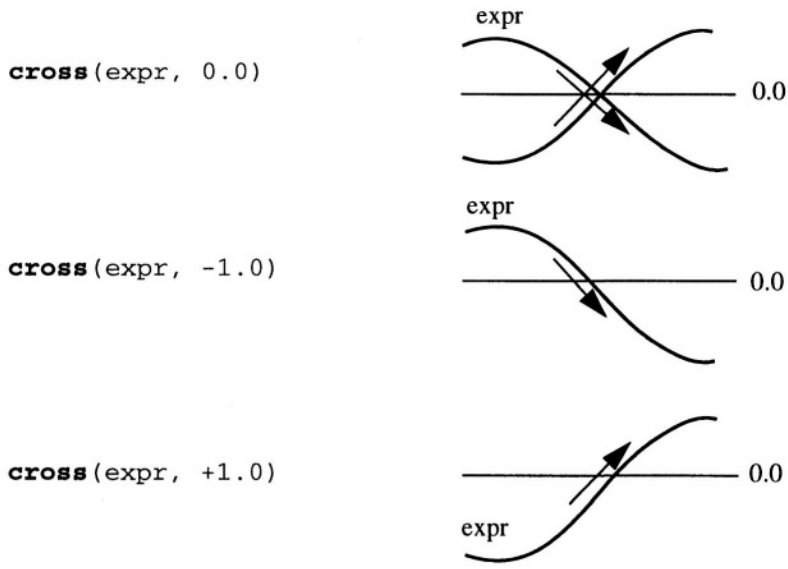


FIGURE 3.26 Illustration of the different specifications of the direction parameter for the **cross** analog operator.

The use of `timetol` is relevant for rapidly changing signals. If the **cross** analog operator is used to delineate regions of behavior in the model, then `valuetol` criteria can also be applied to define the appropriate level of accuracy.

The example Listing 3.19 illustrates a clocked sample-and-hold and how the **cross** operator is used to set the sample value when the rising transition of the clock passes through 2.5.

LISTING 3.19 Verilog-A definition of sample-and-hold based on **cross.**

```
module sah(out, in, clk);
    output out;
    input in, clk;
    electrical out, in, clk;
    real state = 0;

    analog begin
```

```
    @ (cross(V(clk) - 2.5, +1.0)) begin
        state = V(in);
    end
    V(out) <+ transition(state, 1m, 0.1u);
end

endmodule
```

The analog event statement is specified such that it is triggered by a **cross** analog operator when the value of its' expression, `V(clk) - 2.5`, goes from positive to negative.

The 1 millisecond delay specified in the **transition** operator for the output signal, is seen in the simulation results between the sample taken at the rising edge of the clk signal passing through 2.5 volts shown in Figure 3.27.

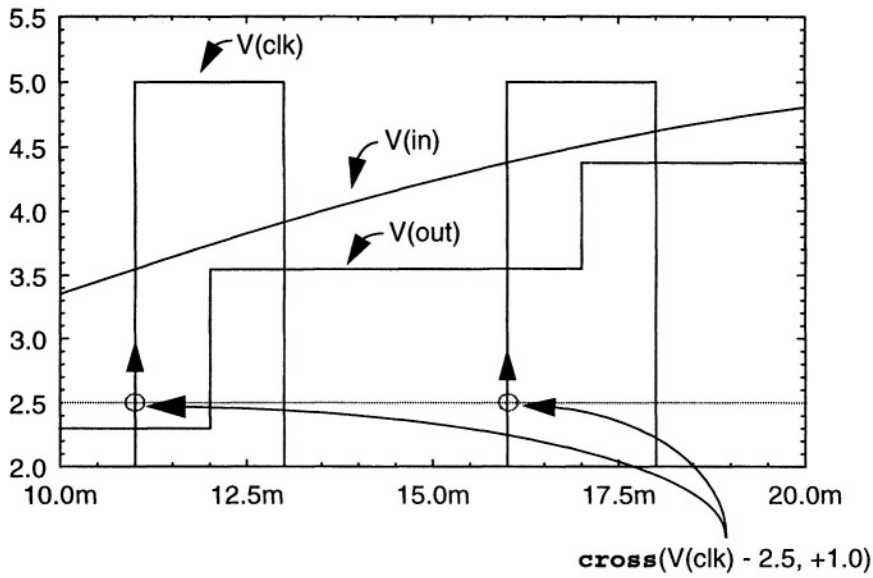


FIGURE 3.27 Sample-and-hold using **cross** analog operator simulation response to sinusoidal input.

The **cross** analog operator maintains internal state and thus has the same restrictions as other analog operators.

3.5.2 Timer Event Analog Operator

The timer event analog operator is used to generate analog events to detect specific points in time.

timer(start, period)

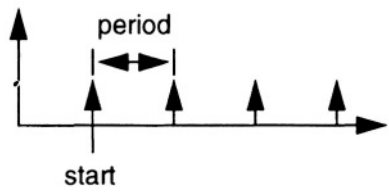


FIGURE 3.28 **timer** analog operator and its graphical representation.

The **timer** event analog operator schedules an event that occurs at an absolute time (as specified by the `start`). The analog simulator places a time point at, or just beyond, the time of the event. If period is specified, then the timer function schedules subsequent events at multiples of `period`.

For example, the following module uses the **timer** operator to generate a pseudo-random bit sequence. To do this, a shift register of length m bits is clocked at some fixed rate period as shown in Figure 3.29. An exclusive-OR of the m-th and n-th bits form the input of the shift-register and the output is taken from the m-th bit.

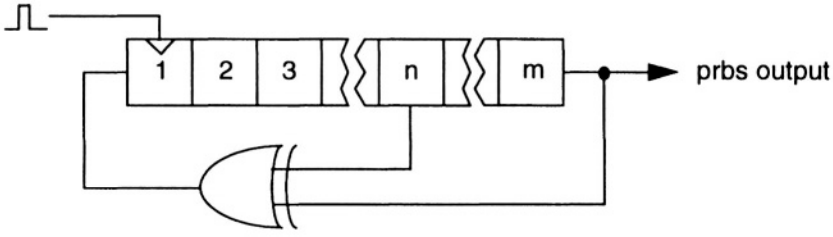


FIGURE 3.29 Pseudo-random bit stream (prbs) generator using exclusive-or and shift register where m is the width of the register and n the tap.

In the definition of the behavior of the pseudo-random bit sequence generator, an integer array is used to represent the shift register and the exclusive-OR operation is done using the (^) operator as shown in Listing 3.20.

LISTING 3.20 Verilog-A behavioral definition of pseudo-random bit stream generator using the **timer** analog operator.

```
analog begin
   @(timer(start, period)) begin
      res = ireg[width - 1] ^ ireg[tap];
      for (i = width - 1 ; i > 0 ; i = i - 1 ) begin
         ireg[i] = ireg[i - 1];
      end
      ireg[0] = res;
   end
   V(out) <+ transition(ireg[width-1], 1n, 1n, 1n);
end
```

The outputs of two pseudo-random bit sequence generators are shown in Figure 3.30 for a **period** of 100n.

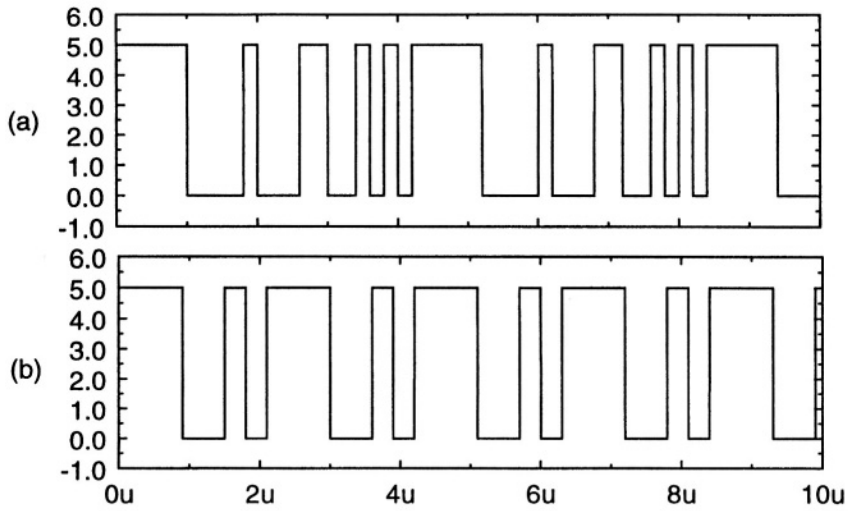


FIGURE 3.30 Time domain results of pseudo-random bit stream generator using the **timer** analog operator and two different shift-register widths and taps, (a) with `width` = 10 and `tap` = 7 and (b) with `width` = 9 and `tap` = 5.

3.6 Additional Constructs

The Verilog-A language provides some additional behavioral constructs, especially useful in the definition of high-level behavioral models. These include access to the simulation environment, additional methods of formulating behaviors, and iterative statements. Some of these have already been introduced indirectly, but will be discussed in more detail in this section.

3.6.1 Access to Simulation Environment

Access to the simulation environment can be necessary for describing behaviors that can be dependent upon external simulation conditions. For example, the following

```
$realtime()
$temperature()
```

are Verilog-A defined system tasks that provide access to the conditions under which the component is being evaluated. The $**realtime**() system tasks accesses the current simulation time and allows custom independent sources to be defined in the language. The ambient temperature, returned by the $**temperature**() system task, can be used to define temperature dependent models such as semiconductor devices.

The modeler has some degree of control over the timesteps utilized during the course of a transient simulation via use of the **bound_step**() function. The real-valued argument to **bound_step**()indicates the maximum timestep that the module requires for meeting its own accuracy constraints; the simulator can make a smaller timestep based on its own accuracy constraints or those of other modules. An example of the use of **bound_step**() is provided in Section 5.5 of the applications chapter.

Additionally, it becomes useful to define behaviors conditionally upon the current analysis. For this purpose, the **analysis**() function is provided. **analysis**()takes a string argument that is a descriptor of the analysis type to test for. For example,

```
analysis("dc")
```

returns 1 during DC analysis, such as that prior to transient analysis in order to determine the initial operating point, and 0 otherwise. Similarly,

```
analysis("tran")
```

returns 1 during a transient analysis, and 0 otherwise. An example of the use of **`analysis`**() for initial conditions is provided in an example of Section 3.6.2.

3.6.2 Indirect Contribution Statements

The probe-source formulation is the primary method of formulating analog behaviors. It provides a clear and tractable description of inputs, outputs, and their relationships in the module definition. However, in all cases it is not necessarily possible nor convenient to formulate behaviors as a function of the output signals. These cases occur commonly while developing purely mathematical models or modeling multi-disciplinary components.

In these cases, the Verilog-A language provides the *indirect contribution statement.* The indirect contribution statement allows for the specification of a behavior in terms of a condition that must be solved for (as opposed to defining an output). The indirect contribution statement allows descriptions of an analog behavior that implicitly specifies a branch potential in *fixed-point* form. This does not require that behavioral relationships be formulated in terms of the outputs.

The general form of the indirect contribution statement is:

```
target : branch == f( signals );
```

Where target represents the desired output, branch can be either of the following:

- An implicit branch such as `V(out)`.
- A derivative of an implicit branch such as **`ddt`**`(V(out))`.
- A integral of an implicit branch such as **`idt`**`(V(out))`.

As with contribution statements, `f ( signals )` can be any combination of linear, nonlinear, algebraic, or differential expressions of a modules input or output signals, constants, and parameters. For example, the ideal op amp, in which the output is driven to the voltage that results in the input voltage being zero. Using indirect contribution assignments, the opamp model could be written[1]:

```
V(out) : V(in) == 0.0;
```

1. The behavior can also be expressed in the probe-source formulation as: `V(out) <+ V(out)+V(in);`

which can be read as: "*determine* `V(out)` *such that* `V(in) == 0`". The indirect contribution statement indicates that the signal `out` should be driven with a voltage source and the source voltage value should be solved such that the given equation is satisfied. Any branches referenced in the equation are only probed and not driven.

For example, the following differential equation and initial condition has a known solution of sin $(\omega_0 t)$:

$$\frac{d^2x}{dt^2} = \omega_0{}^2 \cdot x(t)$$

$$\left.\frac{dx}{dt}\right|_{t=0} = \omega_0$$

Using indirect contribution statements, the behavior would be represented as:

LISTING 3.21 Indirect contribution statement example.

```
analog begin
    if  (analysis ("dc"))
        V(dx) <+ w0;
    else
        V(dx) <+ ddt(v(x));

    V(x)  : ddt(V(dx)) == -w0*w0*V(x);
end
```

For DC (which includes transient analysis initialization), the signal `dx` is set to the initial condition of `w0` by using the **`analysis`**() function within the conditional of the **`if-else`** statement. Note that the **`else`** statement branch of the **`if-else`** statement contains a **`ddt`** operator. This is permissible because the **`analysis`**() statement has static properties (refer to Section 3.4.8).

The contribution statements and indirect contribution statement modelling methodologies provide similar functionality. Use of one or the other depends upon the particular modelling task at hand. However, as a general rule, the two different methodologies are not mixed.

3.6.3 Case Statements

As introduced in Section 3.3.5, the **case** statement is another statement-level construct that allows for multi-way decision tests. The statement tests whether an expression matches one of a number of other expressions, and branches accordingly. The **case** statement is generally used in the following form:

```
case (p1)
   0: $strobe("p1 == 0");
   1: $strobe("p1 == 1");
   default: $strobe("p1 == %d", p1);
endcase
```

The expression within the **case** statement (`p1`) is evaluated and compared in the exact order to the **case** items (`0, 1, and` **default**) in which they are given. During the linear search of the **case** items, if one of the **case** item expressions matches the **case** expression given in parenthesis, then the statement associated with that case item is executed. In this example, if `p1 == 0` or `p1 == 1`, then we will print a message corresponding to `p1` being either `0` or `1`.

If all comparisons fail, and the **default** item is given, then the **default** item statement is executed. If the **default** statement is not given, and all of the comparisons fail, then none of the **case** item statements are executed. In the example, for any case other than `p1` being either `0` or `1`, we print a message indicating the value of `p1`.

3.6.4 Iterative Statements

The Verilog-A language supports three kinds of iterative statements. These statements provide a means of controlling the execution of a statement zero, one, or more times.

repeat executes `<statement>` a fixed number of times. Evaluation of the constant `loop_cnt_expr` decides how many times a statement is executed.

```
repeat ( loop_cnt_expr )
   <statement>
```

while executes a `<statement>` until the `loop_test_expr` becomes false. If the `loop_test_expr` starts out false, the `<statement>` is not executed at all.

```
while ( loop_test_expr )
   <statement>
```

for is an iterative construct that uses a loop variable.

```
for ( init_expr ; loop_test_expr ; post_expr )
   <statement>
```

for controls execution of its associated statement(s) by a three-step process as follows:

- Execute `init_expr`, or an assignment which is normally used to initialize an integer that controls the number of times the `<statement>` is executed
- Evaluate `loop_test_expr` - if the result is zero, the for-loop exits, and if it is not zero, the for-loop executes the associated `<statement>`
- Execute `post_expr`, or an assignment normally used to update the value of the loop-control variable, then continue.

As the state associated with analog operators cannot be reliably maintained, analog operators are not allowed in any of the three looping statements.

3.7 Developing Behavioral Models

For both novice and seasoned model developers, a methodology for developing and validating behavioral models is essential. The process of developing a behavioral model should provide for a development of an intuitive understanding of the model as well as the system in which it will operate. In contrast to digital simulation which is activity-directed and the signals that effect a model can be easily isolated, behavioral models defined for analog simulation must account for the loading and timing (or lack thereof) in the whole system.

3.7.1 Development Methodology

A methodology for developing behavioral models should encourage a process of stepwise refinement from the concept, to implementation and validation of the model. The conceptual stage involves developing an understanding of what the behavioral model is to accomplish in terms of capabilities and performance and other specifications. The formulation, preferably beginning from an existing model, is the factorization of

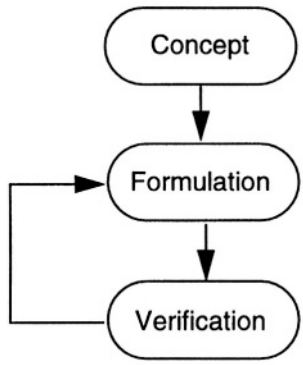


FIGURE 3.31 Step-wise methodology for development and refinement of behavioral models.

that specification into structural and behavioral components and its actual implementation. Verification and/or validation of the model includes the development of test benches that can be used to test the behavior of the module to the original specifications. The methodology and development used for verification and validation of the model can also be applicable in the verification of the final circuit-level implementation.

3.7.2 System and Use Considerations

A module defined in a behavioral language such as Verilog-A can be used as a component within different types of systems and this should be reflected in the verification phase of the module. For example, developing a behavioral model for use as a component within a library would require much more rigorous formulation, as well as have more stringent criteria for validation, than a model developed strictly for use as a component in one specific design.

Understanding the context of use can also help the model developer make appropriate decisions for accuracy as well as for efficiency in simulation. The **`transition`** analog operator, which converts a discrete input to a piece-wise linear output, is characterized in terms of rise (`tr`) and fall (`tf`) times of the output.

```
pwl_output = transition(disc, td, tr, tf) ;
```

Using very small values for `tr` and `tf`, relative to the overall length of the simulation can be very costly in terms of simulation time. Moreover, the resulting fast-changing

output will not necessarily reflect the physical system or the underlying implementation. Similar considerations can be made when defining the sensitivity of a a model to its inputs as in the use of tolerances in the **`cross`** operator.

3.7.3 Style

One of the major benefits of HDL-based design is the ability to convey and reuse designs that are represented at a high-level of abstraction. This ability to communicate the design information effectively amongst a group of designers is enhanced by adopting consistent and agreed-upon techniques of style for the development of models. For example, the following are some of the common denominators in the development of behavioral models that are easily defined:

- Port ordering convention (inputs first, then outputs or vice-versus).
- Degree of parameterization of the model and naming conventions.
- Use of the Verilog pre-processor for enabling consistency and code documentation purposes.
- Coding style (layout) conventions for the module definitions.

The basic underlying theme is to *plan* for model reuse.

CHAPTER 4

Declarations and Structural Descriptions

4.1 Introduction

Structural definitions in the Verilog-A language are the primary mechanism by which a hierarchical design methodology such as top-down is facilitated for analog and mixed-signal designs. The Verilog-A language allows analog and mixed-signal systems to be described by a set of components or modules and the signals that interconnect them. The connection of these modules is defined in terms of the parameters, as well as the ports or connection points, declared within the module definitions. The declaration of parameters and ports within the module definition define the interface. The interface definition determines how the module will be instantiated as part of a structural module definition or as a component within a Spice netlist.

This chapter overviews the parameter, port, local variable and signal declarations, as well as module instantiations within the Verilog-A language. This chapter also looks at how module definitions relate to their instantiations.

4.2 Module Overview

A module in the Verilog-A language represents the fundamental user-defined type. A module definition can be an entire system, or only a component within a system. A

module definition can be an active component in the system in which it effects the signals in the system, either dependently or independently. Conversely, a module can be a passive component which only monitors activity in the system, performing functions associated with test benches.

Other than adhering to the constructs of the Verilog-A language, there are no restrictions on the type of systems that can be represented and how the representation is defined. Module descriptions can include any number and type of parameters, be an entirely structural or behavioral description, or include aspects of both structure and behavior.

The general constituents of a module definition include the interface declarations and the contents. The interface declarations consist of both the port and parametric declarations of the module. The module contents can be composed of structural instantiations, behavioral relationships, or both. For illustration purposes, the Verilog-A description for a phase-lock loop system is used as shown in Figure 4.1.

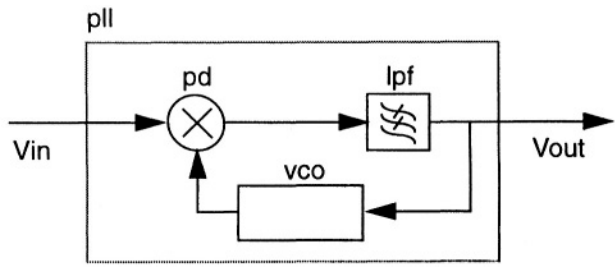


FIGURE 4.1 Schematic of phase-locked loop subsystem.

The corresponding Verilog-A definition of the system is listed in Listing 4.1.

LISTING 4.1 Verilog-A definition of VCO and PLL.

```
`include "std.va"
`include "const.va"

module pd(out, in1, in2);
   inout out, in1, in2;
   electrical out, in1, in2;

   parameter real gain = 1.0;
```

```
    analog
       V(out)  <+ gain*V(in1)*V(in2);

endmodule

module lpf(l, r, gnd);
    inout l, r, gnd;
    electrical l, r, gnd;

    parameter real res = 1k;
    parameter real cap = 1u;

    analog begin
       I(l, r)  <+ V(l, r)/res;
       I(r,gnd) <+ ddt(V(r, gnd)*cap);
    end

endmodule

module vco(out, in);
    inout out, in;
    electrical out, in;

    parameter real ampl = 1.0; // V
    parameter real fc = 10.0k; // Hz
    parameter real kv = 1.0k; // V/s

    real freq_v;  // local variable declaration

    analog begin
       freq_v = fc + kv*V(in);
       V(out) <+ ampl*sin(2*`M_PI*idt(freq_v));
    end

endmodule

// structural definition of the pll system
module pll(out, in, gnd);
    inout out, in, gnd;
```

```
    electrical out,  in,  gnd;

    electrical pdout, vcoout; // local signals

    pd pd1(pdout, in, vcoout);
    lpf 1pf1(pdout,  out,  gnd);
    vco vco1(vcoout, out);

endmodule
```

The module definitions listed above consist of four primary components:

- interface declarations (port and parameter declarations for all modules).
- structural instantiations (module instantiations declared in the `pll` module).
- local variable declarations (`freq_v` in the `vco` module).
- behavioral relationships (`vco` constitutive relationship).

The module definition of the phase-locked loop (`pll`), declares (instantiates) the phase detector (`pd`), low-pass filter (`lpf`), and voltage-controlled oscillator (`vco`) components. The definition of the phase-locked loop module defines the connectivity of the other components comprising the system. In addition, parameters specified at one level in the hierarchy can be passed down to lower levels during instantiation.

The structural instantiation of components within the Verilog-A language is dependent upon the nature of the interface port and parameter declarations of the module, and the language constructs used for instantiation.

4.2.1 Introduction to Interface Declarations

The interface declarations for a module definition include both port and parameter declarations. The port declarations define the *type* and *direction* of signals and indicate how that component can be instantiated within a structural description. The parameters to the module can be used to characterize both behavior and structure.

Analog signals in the Verilog-A language are defined in terms of the quantities composing the signal. As described previously, the definition of a signal type is encapsulated in the **`nature`** and **`discipline`** definitions and must be known before used within a module definition. The following Verilog-A preprocessor construct is used to

include the standard definitions of signals and physical constants prior to the definition of any modules.

```
`include "std.va"
`include "const.va"
```

The vco uses the standard definition for electrical systems of the discipline of type electrical for declaring the types of the ports.

```
module vco(out, in);
    inout out, in;
    electrical out, in;
```

The electrical discipline characterizes the type of the signals between components in the system. This example declares the ports of the vco to be of types **inout** (bidirectional) which is a characteristic of conservative systems.

Parameter declarations include both the name and default values. Parameters for the vco, include the amplitude of the output sinusoid (ampl), the center frequency of the oscillator (fc), and the conversion gain of the oscillator (kv), are declared by the following[1]:

```
parameter real ampl = 1.0; // V
parameter real fc = 10.0k; // Hz
parameter real kv = 1.0k; // V/s
```

For both port and parameter declarations, the order within the module definition is significant as this defines how the module is instantiated within a hierarchical design. These concepts will be expanded upon further in section 4.3.2.

4.2.2 Introduction to Local Declarations

The Verilog-A language supports local declarations of variables of type **integer** and **real**, as well as analog signals. In the vco module, the line

```
real freq_v; // local variable declaration
```

declares a local variable freq_v for use within the module. Similarly, local signals, or signals that do not appear within the modules port or connection list, are declared

1. The Verilog-A language specification extends the Verilog HDL specification to include optional type specifiers on parameter declarations. This is described in Section 4.3.2.

in the same manner: a discipline type name followed by a list of one or more identifiers of the signals. The difference between local and port signal declarations is that the local signal declarations do not have directions associated with them. Local signals are typically used for defining intermediate structure within the module or for higher-order formulations of analog behaviors. In the `pll` module definition of Listing 4.1,

```
electrical pdout, vcout;
```

declares signals `pdout` and `vcout` that are local to the `pll` module and used only for connecting the `pll`'s instantiated components.

4.2.3 Introduction to Structural Instantiations

The Verilog-A language supports hierarchical descriptions by allowing modules to be embedded within other modules. Higher level modules create instances of lower-level modules and communicate with them through **`input`**, **`output`**, and **`inout`** ports.

The `pll` module definition instantiates the modules `pd`, `lpf`, and `vco` via the declaration statements:

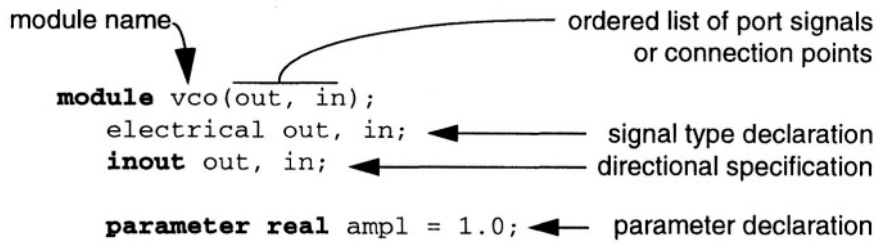
```
pd pd1(in, pdout, vcout);
lpf lpf1(pdout, out, gnd);
vco vco1(vcoout, out);
```

For example, we instantiate one `vco` instance named `vco1` within the `pll` module definition.

The type and direction of the signals, as well as parameters, used within the instantiation statements must be compatible with the respective declarations within the child module definition(s). The assignment of the connections and parameters of child modules is done via port and parameter association. The association can be done via *named association,* which assigns an expression or signal in the parent module with the parameter or signal *name* in the child module, or by *positional association,* which assigns an expression or signal to the *order of declaration* of a parameter or signal in the child module.

4.3 Module Interface Declarations

A module definition is enclosed between the keywords **module** and **endmodule**, in which the identifier following the keyword **module** is the name of the module being defined. For the vco module previously defined:



The optional list of ports specify an *ordered list* of the module's ports. The order specified will be significant when instantiating the module. The identifiers in this list must be declared with a discipline type defining the type of the signal and a directional specifier such as **input**, **output**, and **inout** in declaration statements within the module definition.

In addition to port declarations, module definitions can also optionally incorporate declarations of parameters incorporating default values and optional range checks. As with ports, the *order of declaration* can be significant when instantiating the module.

4.3.1 Port Signal Types and Directions

Ports provide a means of interconnecting instances of modules. For example, if a module X instantiates module Y, the ports of module Y are *associated* with either the ports or the internal signals of module X. Associating connections between modules requires that both the *type* and the *direction* of the signals are compatible. The Verilog-A language requires that both the type and direction attributes be declared for each of a modules' port signals.

The Verilog-A language uses the discipline definition for the type of the declaration for module ports (ports are also analog signals or nodes). Hence, the type of a modules' ports are declared by giving their **discipline** type, followed by the list of port identifiers as,

```
electrical out, in;
```

which declares two signals, `out` and `in`, of type `electrical`. The `electrical` discipline must have been defined prior to its use in the declaration. In essentially all cases, the definition of a **`discipline`** type comes from the standard include files,

```
`include "std.va"
```

which is read prior to any module definitions.

The direction of a port can be specified as **`input`**, **`output`**, or **`inout`** (bidirectional)[1]. If the direction is specified as being an **`input`** port, then the module will only monitor the signals at the port and not modify them. That is, within the module the port can only be passed into other modules as **`input`** ports and the signals on the ports can only be used in expressions. A signal declared as **`input`** cannot be used on the left side of a contribution statement (as a source).

If the direction is specified as being an **`output`** port, then the module will only affect the signals at the port, but not be affected by them. Thus, the port can be passed to instances of other modules as **`output`** ports and the signals on the ports cannot be used in expressions but can be used on the left side of a contribution statement. Finally, ports that are declared as being **`inout`** or bidirectional are not subject to these restrictions.The syntax for port directional declarations is illustrated by example below:

```
module vco(out, in);
    electrical out, in;
    inout out, in;
```

Two signals, `out` and `in`, in the port list of the `vco` are declared of type `electrical`. The syntax for the port direction specification follows that from the type of the port signals.

If the direction of the port is not specified, it is taken to be bidirectional (**`inout`**). In analog system modeling, a port, which is also a node, represents a point of physical connection between modules of continuous-time descriptions obeying conservation-law semantics. Thus, in most cases, the **`inout`** directional specifier is used. In mod-

1. The directional specifiers **`input`**, **`output`**, and **`inout`** are only relevant for signals within the modules port list. Internal signals only require (allow) the type specifier or discipline.

ule definitions in which signal-flow behavioral modeling is used, or when the directional specifiers are required as an documentation aid in a conservative description, the unidirectional specifiers **input** and **output** are appropriate.

In Verilog HDL, there is a close association of the directionality specified for the ports due to the activity-directed nature of digital simulation. In the Verilog-A language, for analog simulation, the same degree of association does not exist as all analog signals or unknowns in the analog system are solved for simultaneously.

Listing 4.2 and 4.3 provide examples of port declarations and their usage.

LISTING 4.2 Definition and usage of conservative signals

```
module conservative(p, n);
   electrical p, n;
   inout p, n;

   // ports p, n used on both sides of contribution
   analog
      V(p, n) <+ I(p, n)*R;

endmodule
```

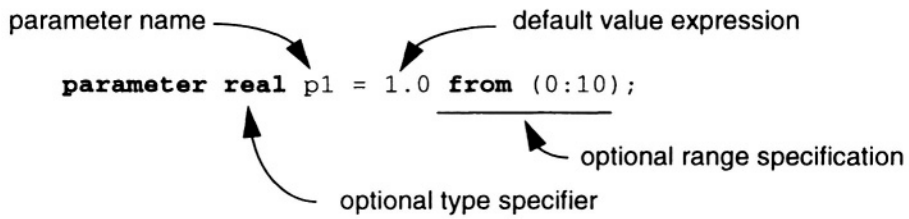
LISTING 4.3 Definition and usage of signal-flow signals.

```
module signal_flow(out, in);
   voltage out, in;
   output out;
   input  in;

   parameter real gain = 1.0;

   // out only used for source, in only for probe.
   analog
      V(out) <+ gain*V(in);

endmodule
```

4.3.2 Parameter Declarations

Parameter declarations are extensions of the basic **integer** and **real** type declarations supported by the Verilog-A language. Parameters differ in that parameter declarations must have initialization or default value expressions. Parameters can be modified via structural instantiation to have values that are different from those specified by the default value expression. In addition, parameters support an extended declaration syntax for range checking which allow the model developer to define acceptable ranges or values for the parameters. Specifying the valid range of values that a parameter can be assigned to during instantiation allows the model developer the ability to restrict the values for the parameters to insure proper and/or expected use of the model.

The basic syntax of parameter declarations is illustrated below:

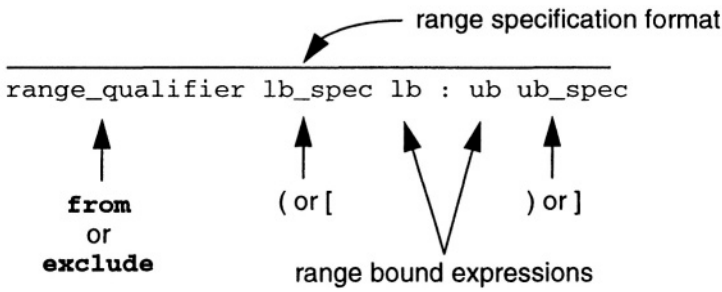


The **parameter** keyword can be followed by an optional type specification (**real** or **integer**) prior to the parameter name (`p1` above). If a type of a parameter is not specified, it is derived from the type of the value of the default value expression. If the type of the parameter is specified, and the value of the default expression conflicts with the declared type of the parameter, the value of the default expression is coerced to the type of the parameter.

As parameters are considered constants[1] during the simulation. As constants, a default value for the parameter must always be specified in its declaration. This also infers that the parameter value can only be set during the instantiation of the module, and hence it is illegal to modify their value at runtime.

1. Their values are know at elaboration or at the time of instantiation and do not change during the course of the simulation.

The parameter declaration can contain optional specifications of the permissible range of the values for that parameter. The range specification consists of a qualifier and a range. The value of a parameter is checked against the range during instantiation, and is not a runtime assertion check.



The qualifiers for the range specification include either **from** or **exclude**. If the keyword **from** is used, the value of the parameter must be within the following range. If the keyword **exclude** is used, the value of the parameter must be outside of the specified range. For the range, the use of brackets for the lower bound specifier (`lb_spec`) and upper-bound specifier (`ub_spec`), [and] respectively, indicate *inclusion* of the end point `lb` or `ub` in the range. The use of parenthesis for the `lb_spec` or `ub_spec`, (and) respectively, indicate *exclusion* of the end point `lb` or `ub` from the valid range. It is possible to include one end point and not the other by mixing inclusion/exclusion combinations of the range bound specifiers, such as [ ) or ( ]. In all cases, the first expression in the range must be numerically smaller than the second expression in the range (`lb < ub`).

More than one range may be specified for inclusion or exclusion of values as legal values for the parameter. The keyword **inf** may be used to indicate infinity for one or the other bound if there is none. Examples of legal parameter declarations are shown in Listing 4.4.

LISTING 4.4 Example parameter declarations

```
parameter real p1 = 1.0;
parameter real p2 = 1.0 from (0:inf);
parameter integer ip1 = 1 exclude 0;
parameter p3 = 1.0;
parameter real p4 = 1;
```

4.4 Local Declarations

Local declarations are variables and signals declared within the scope of a modules definition. For variables in the Verilog-A language, this includes variables of types **integer** and **real**. For signals, these can be of any defined **discipline** type.

Variable declarations in the Verilog-A language are similar to many programming languages in that the type keyword is followed by a list of one or more identifiers. The identifiers can be scalar or vector. Each variable identifier is initialized to zero, as initializer expressions for non-parameters is not allowed. For example,

LISTING 4.5 Illustration of local variable declarations.

```
module ex(out, in);
    inout out, in;
    electrical out, in;

    parameter integer width = 4;

    real x, y;
    real d[0:width - 1];

    electrical t1, t2;

    ...

endmodule
```

The first declaration, "**real** `x, y;`", declares two real-type variables named x and y. The second, "**real** `d[0:width - 1];`", declares a real-type vector with a left bound of 0, and an upper bound of `width - 1`. The size of the vector `d` is parameterized by the parameter `width` and includes both the left-bound and right-bound elements.

Signal declarations are similar in that the **discipline** is treated as a user-defined type. The declaration,

```
electrical t1, t2 ;
```

declares two signals of type electrical named t1 and t2. These signals can be used in the instantiation of components of a structural definition for the module (Section 4.5), in defining internal structure for the module, or higher-order behavioral definitions.

4.5 Module Instantiations

A structural description in Verilog-A is any description in which a module instantiates another module within the scope of its definition. A structural definition for the system will define an explicit hierarchy in the design.

Instantiation allows one module to incorporate a copy of another module into itself by instantiating it[1]. The module instantiation statement creates one or more named instances of a defined module.

When one module instantiates another module, it can alter the values of any parameters declared within the instantiated module. The common ways to alter parameter values are:

- Module instance parameter value assignment by order, which allows values to be assigned in-line during module instantiation in the order of their declaration. This is known as *positional association* of module parameters.
- Module instance parameter value assignment by name, which allows values to be assigned in-line during module instantiation by explicitly associating parameter names with the overriding values. This is known as *named association* of module parameters.

Similarly, for connections to the model, there are two ways to assign ports of the instantiated module to the local connection points:

- Module instance parameter value assignment by order, which allows values to be assigned in-line during module instantiation in the order of their declaration (positional association).
- Module instance parameter value assignment by name, which allows values to be assigned in-line during module instantiation by explicitly associating parameter names with the overriding values (named association).

1. A module definition does not contain the text of another module definition within its **`module-endmodule`** keyword pair.

The general syntax for module instantiation are:

```
mod_name #(param_assigns) inst_name(port_assigns)
```

Where `param_assigns` and `port_assigns` can be either by positional or named association. Within an individual `param_assigns` or `port_assigns` of a module instantiation, positional and named association cannot be mixed.

4.5.1 Positional and Named Association Example

The example of Figure 4.2 of a sub-ranging 8-bit A/D will be used to illustrate the parametric specification and port connection of the module instantiation process. The sub-ranging A/D consists of two 4-bit A/Ds, a 4-bit D/A, summing, and gain stages.

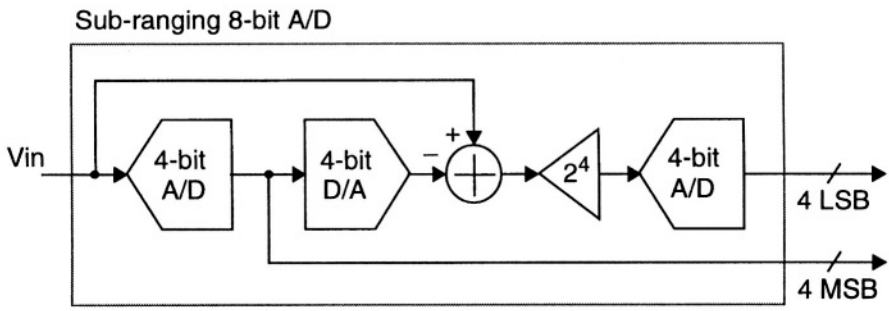


FIGURE 4.2 Architectural schematic of a 8-bit sub-ranging A/D (no error correction).

The Verilog-A language definition of the interfaces (parameters and ports) of the relevant portions of the instantiated components, as well as the structural definition of the sub-ranging N-bit A/D system are shown in Listing 4.6.

LISTING 4.6 Verilog-A definition of the 8-bit sub-ranging A/D structure.

```
// module interface declarations for a2d definition
module a2d(d0, d1, d2, d3, in, clk);
    input in, clk;
    output d0, d1, d2, d3;
    electrical in, clk;
    electrical d0, d1, d2 , d3 ;
```

```
    parameter real vrange = 1.0;
    parameter real tdel = 10n;
    parameter real trise = 10n;
    parameter real tfall = 10n;

    ....

endmodule

// module interface declarations for d2a definition
module d2a(out, d0, d1, d2, d3, clk);
    output out;
    input d0, d1, d2, d3, clk;
    electrical out;
    electrical d0, d1, d2, d3, clk;

    parameter real vthresh = 0.5;
    parameter real tdel = 10n;
    parameter real trise = 10n;
    parameter real tfall = 10n;

    ...

endmodule

module sum(out, posin, negin);
    inout out, posin, negin;
    electrical out, posin, negin;

    ...

endmodule

module gain(out, in);
    inout out, in;
    electrical out, in;

    ...
```

```
endmodule

// structural instantiations of all child modules
module subranging_a2d(bit0, bit1, bit2, bit3,
   bit4, bit5, bit6, bit7, in, clock);
   output bit0, bit1, bit2, bit3,
      bit4, bit5, bit6, bit7;
   input in, clock;
   electrical bit0, bit1, bit2, bit3,
      bit4, bit5, bit6, bit7;
   electrical in, clock;

   // internal signals
   electrical aout, rem_out, gain_out;

   // structure
   a2d #(.vrange(5.0))
      msb_a2d(bit4, bit5, bit6, bit7, in, clock),
      lsb_a2d(bit0, bit1, bit2, bit3, gain_out, clock);

   d2a  #(.vthresh(2.5))
      convrtr(aout, bit4, bit5, bit6, bit7, clock);

   sum sum1(rem_out, aout);
   gain gain1(gain_out, rem_out);

endmodule
```

This example will be used in the following sections as an example of the variations for assigning parameters and connecting ports in structural definitions.

4.5.2 Assignment of Parameters

Parameter value assignment by position is a method for assigning values to parameters within module instances of a module to any parameters that have been specified in the definition of that module.

The order of the assignments in module instance parameter value assignment must follow the order of declaration of the parameters within the module. For example, in the `a2d` module defined previously:

```
module a2d(d0, d1, d2, d3, in, clk);
   input in, clk;
   input d0, d1, d2, d3;
   electrical in, clk;
   electrical d0, d1, d2, d3;
   parameter real vrange = 1.0;   <-- parameter #1
   parameter real tdel = 10n;     <-- parameter #2
   parameter real trise = 10n;    <-- parameter #3
   parameter real tfall = 10n;    <-- parameter #4
```

Parameter value assignment by name, or *named association,* is a method for assigning values to parameters within a module instance to any parameters that have been specified in the definition of that module. Parameters are assigned within the declaration or instantiation statement (within the "#(...)") in the parent module definition. The name of the parameter to be assigned must be preceded by a period (.) and must be the name of a parameter in the definition of the module being instantiated. The overriding value for each parameter must be a constant expression and must be enclosed in parenthesis `( )`.

Named association is used for the two `a2d` module instances within the `subranging_a2d` module. The following lines instantiated the two components `msb_a2d` and `lsb_a2d` of the `a2d` module:

```
a2d #(.vrange(5.0))
   msb_a2d(bit4, bit5, bit6, bit7, in, clock),
   lsb_a2d(bit0, bit1, bit2, bit3, gain_out, clock);
```

For both instances, the `vrange` parameter is set to 5.0. Note that the parameter `vrange` belongs to the module `a2d`. The expression that is used to initialize the parameter, 5.0, is evaluated in the context of the instantiating module (`subranging_a2d`).

Positional association is the other way to assign values to parameters. The order in which parameters are passed in the declaration or instantiation statement (within the "#(...)"), correspond to the order in the definition of the module. It is not necessary to assign values to all of the parameters within a module using this method, only up to

the last parameter in the module definition that should be assigned a value different from its default value. For example, using the a2d instances of subranging_a2d:

```
a2d #(5.0)
   msb_a2d(bit4, bit5, bit6, bit7, in, clock),
   lsb_a2d(bit0, bit1, bit2, bit3, gain_out, clock);
```

also assigns the value 5.0 to the parameter vrange of both instances as vrange is the first parameter defined in the a2d module. Note that it is not possible to skip over a parameter assignment using this method. An alternative is to assign values to all of the parameters, but use the default value (the same value assigned in the declaration of the parameter within the module definition) for those parameters that do not need new values. For instance,.

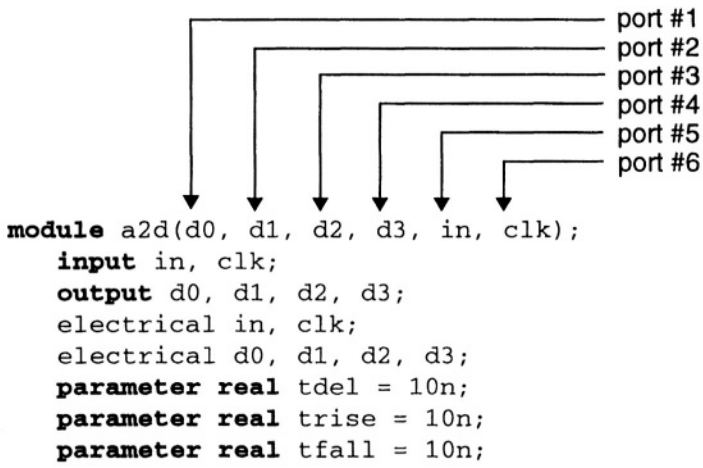
```
a2d #(5.0, 10n, 15n, 15n)
   msb_a2d(bit4, bit5, bit6, bit7, in, clock),
   lsb_a2d(bit0, bit1, bit2, bit3, gain_out, clock);
```

The above instantiates the components (msb_a2d and lsb_a2d) of the subranging_a2d module with specified parameter values of vrange = 5.0 as before, and trise = 15n, and tfall = 15n. Parameter tdel is assigned its default value of 10n as specified in the definition of the a2d module. Only those parameters whose value is being overridden from the modules default value need specification within the declaration statement.

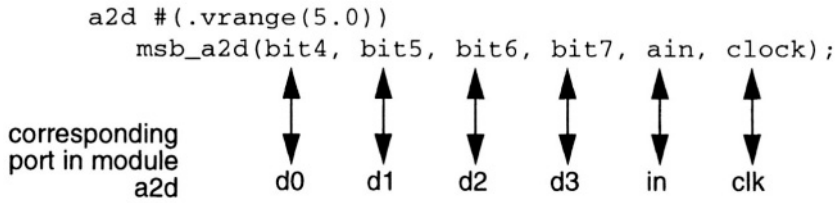
4.5.3 Connection of Ports

One method of making the connection between the ports listed in a module instantiation and the ports defined by the instantiated module is the ordered list - that is, the

ports listed for the module instance must be in the same order as the ports listed in the module definition. For example, in the `a2d` module defined previously:



The following instantiates a components (`msb_a2d`) of the `a2d` module defined above:



The second way to connect module ports consists of explicitly linking the two names for each side of the connection - the name used in the module definition, followed by the name used in the instantiating module. This compound name is then placed in the list of module connections. The name of the port must be the name specified in the module definition (same as for parameters).

Using connection by name, the previous example can be rewritten:

```
a2d #(.vrange(5.0))
   msb_a2d(.d0(bit4), .d1(bit5), .d2(bit6),
       .d3(bit7), .in(in), .clk(clock)),
    lsb_a2d(.d0(bit0), .d1(bit1), .d2(bit2),
       .d3(bit3), .in(gain_out), .clk(clock));
```

The two types of module port connections can not be mixed; connections to the ports of a particular module instance must be all by order or all by name. The are rules governing the way module ports are declared and the way they are interconnected. The most important of which is that all ports connected to a node must be compatible with each other as well as to the discipline of the node[1].

1. The node of any discipline type is compatible in a connection to the ground or reference node.

CHAPTER 5

Applications

5.1 Introduction

The Verilog-A language can be used to describe the analog behaviors of both electrical and non-electrical systems at different levels of abstractions. To illustrate this, a number of examples are given in this chapter using different modeling objectives and techniques. The examples illustrated in this chapter include modeling of:

- Common emitter amplifier
- Voltage regulator
- Operational amplifier
- QPSK modulator and demodulator
- Frequency synthesizer
- Position control system

The modeling and characterization of a common emitter amplifier is used to illustrate three levels of models for the amplifier. The first model is only applicable for midband operation, where gain is constant over a given frequency range. The other two examples of the common emitter amplifier, show different styles to include gain behavior outside the midband range. Spice simulation results are provided for reference.

An operational amplifier example includes the model of the op amp, including a test bench model to measure the settling time characteristics of the amplifier. The effects of poles in the gain/frequency plot is modeled using two techniques. One model uses a resistor and a capacitor in the transfer function to provide the dominant low frequency pole effect. The other example models a higher frequency second order pole using a Laplace transform function.

The voltage regulator example includes a bandgap reference circuit which uses a curve-fitting equation to define the output voltage. The equation includes the effect of supply voltage and temperature variation. The equation was derived from extrapolated data obtained from transistor-level Spice simulations, traceable to actual silicon.

Three system level examples are also given. The QPSK modulator and demodulator show high-level modeling of analog behaviors in which nonlinearities are present. A fractional N-loop frequency synthesizer illustrates analog and digital modeling in a mixed-signal system. An antenna position-control system is used to illustrate the use of the Verilog-A language in modeling and optimization of electro-mechanical systems.

5.2 Behavioral Modeling of a Common Emitter Amplifier

A single transistor common emitter amplifier is used to illustrate the concept of developing a model. This classic example provides a good review of basic principals of circuit design and analysis. It includes DC biasing requirements, transistor parameter considerations, and AC constraints due to the transistor parasitics and discrete capacitors used in the design.

Results from the simulation of the Verilog-A common emitter amplifier model can be compared to Spice transistor-level simulations and with laboratory measurements, if desired.

This section explains a bottom-up methodology of step-wise refinement in analog behavioral model development consisting of the following:

- spice transistor model
- functional model
- structural model for the behavior .

- behavioral model

It is common to define different levels of abstractions in the model of a component for either top-down or bottom-up design methodologies. Functional models can be used to verify connectivity of the component within the system of a larger design, while more detailed behavioral and transistor-level models can be utilized to investigate higher-order effects on performance.

The common emitter amplifier circuit to be modeled is shown in Figure 5.1. It contains a generic npn transistor, biasing resistors, and coupling capacitors, designed to provide a small signal gain of around 25 in the midband frequency range between a low frequency zero less than f_L= 1.0 kHz, and high frequency pole greater than f_H = 400 kHz.

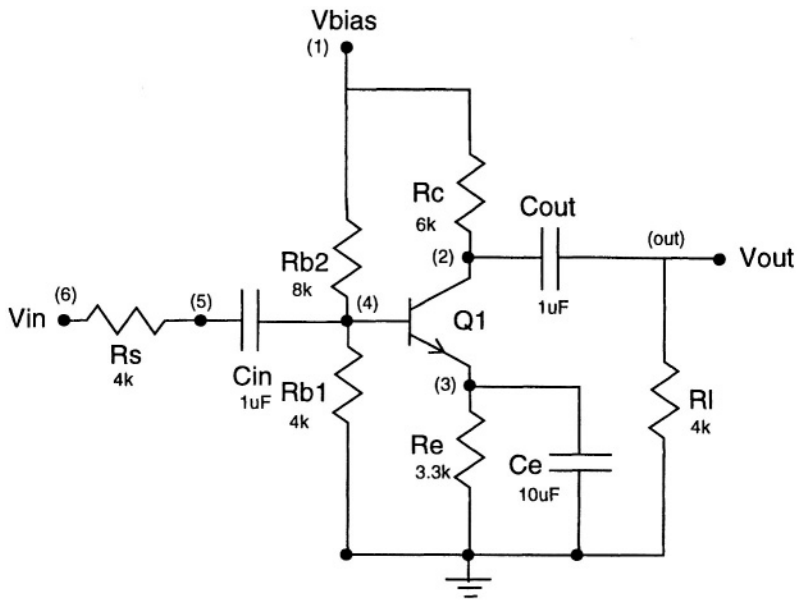


FIGURE 5.1 Spice schematic of the common emitter amplifier.

Typically, Spice circuit simulation results are used as a reference for integrated circuit model development. The Spice transistor model parameters can be characterized to agree with silicon test structures, and provide a path to link the simulation results to the manufacturing process.

For the purpose of the Verilog-A model development, a test structure, as shown in Figure 5.2, is utilized for encapsulating the different representations of the amplifier.

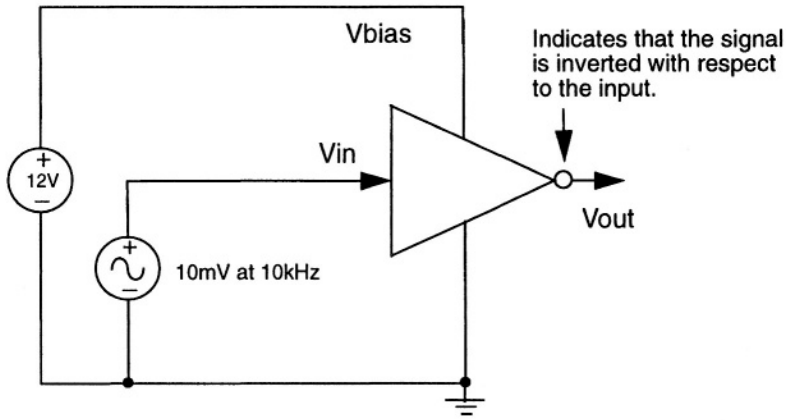


FIGURE 5.2 Test bench configuration for common emitter amplifier circuit.

The gain block is an inverting amplifier, that is, the output is inverted with respect to the input. To develop an understanding for the design requirements and constraints, the amplifier circuit is analyzed in detail, within and outside the midband range.

The Spice input file for the common emitter amplifier is shown in Listing 5.1.[1]

LISTING 5.1 Spice listing of common emitter amplifier test bench for representations of all models in this section.

```
* title: test bench for models

* Verilog-A input files
.verilog "ceamp_fm.va"
.verilog "ceamp_rc.va"
.verilog "ceamp_lp.va"

* sources
Vcc 1 0 dc 12
```

1. The Spice test bench file contains references to all the behavioral models being developed in this section.

```
Vin 6 0 dc 0 ac 1 sin(0 10m 10k)

* biasing resistors
Rs  5 6 4k
Rb1 4 0 4k
Rb2 1 4 8k
Rc 1 2 6k
Re 3 0 3.3k

* load resistors
Rsp out_sp 0 4k
Rfm out_fm 0 4k
Rrc out_rc 0 4k
Rlp out_lp 0 4k

* coupling capacitors
Cin 5 4 1uf
Ce 3 0 10uf
Cout 2 out_sp 1uf

* transistors
Q1 2 4 3 Qnpn
.model Qnpn npn (Is=48.718fA BF=200 BR=100.1m Rb=0
+ Re=0 Rc=0 Cjs=0F Cje=4.5pF Cjc=3.5pF Vje=750mV
+ Vjc=750mV Tf=461.95ps Tr=10ns mje=333.33m
+ mjc=333.33m VA=200V ISE=0A IKF=10mA Ne=1.5)

* Verilog-A behavioral models
xa1 6 out_fm ceamp_fm gain=25
xa2 6 out_rc ceamp_rc gain=25
xa3 6 out_lp ceamp_lp gain=25

.op
.ac dec 1k 10 100Meg
.tran 1n 200u

.end
```

The common emitter amplifier is first simulated at the transistor level with Spice for the amplifier biased in the midband frequency range. The results of running a Spice

small-signal transient analysis on the common emitter amplifier is shown in Figure 5.3. The transient simulation results of Figure 5.3(a) are used to verify the connectiv-

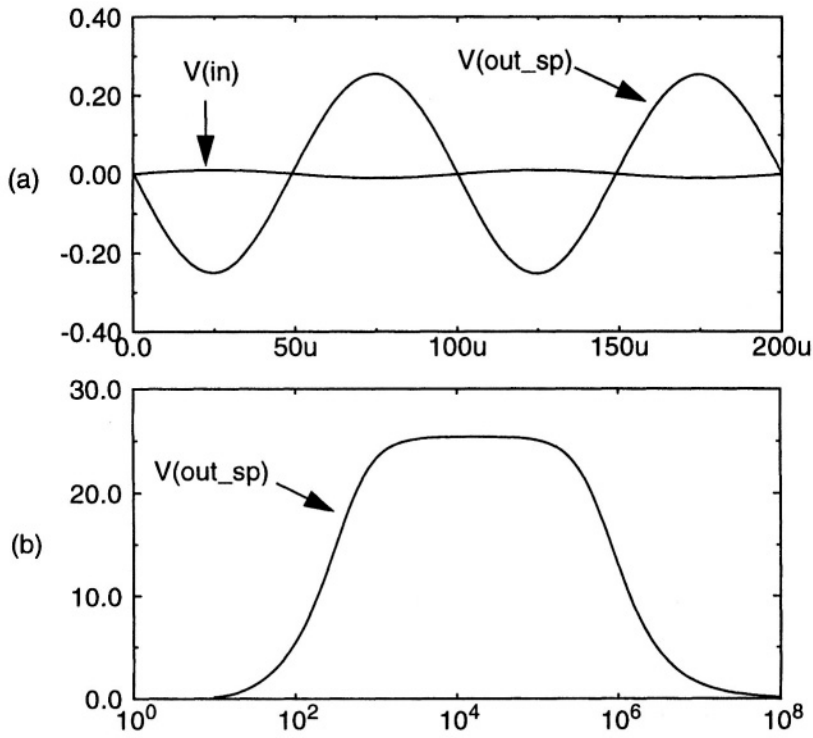


FIGURE 5.3 Time domain analysis (a) and results for the Spice transistor-level circuit of the common emitter amplifier for amplifier input and output. Small-signal AC magnitude response (b) shows constant gain from 1kHz to 400kHz.

ity and proper operation of the amplifier. In addition, the small-signal AC response of Figure 5.3(b) of the amplifier offers some insight into the model requirements.

5.2.1 Functional Model

When the amplifier is used within the midband frequency range, a simple gain model without frequency effects is adequate for system evaluation purposes. Functional models are useful for top-level system architectural design and analysis. From the

previous transistor-level Spice analysis, a simple functional model can be valid for the midband frequency range between 1kHz to 400kHz. With this simplification, we only need to focus on modeling the gain characteristics.

With reference to the schematic of Figure 5.1, the gain of the amplifier at midband is determined using the following equation:

$$A_o = \left((R_{in} / (R_{in} + R_s)) \times g_m \times R'_L \right)$$

The resistance at the base (R_{in}) of the npn is related to the following parameters:

$$R_{in} = R_{b1} \| R_{b2} \| \langle r_\pi + r_x \rangle$$

where r_π is the internal resistance of the npn between the base and the emitter, and r_x is resistance from the external base to the internal intrinsic base of the npn. The effective load resistance R'_L is a parallel combination of resistances at the output node.

$$R'_L = R_c \| R_l \| r_o$$

The output resistance of the npn r_o is a function of a constant V_A, called the Early voltage, and the collector current I_C. The transconductance of the npn g_m is I_C/V_T = 3.85mA/V at T = 300K and I_C = 1mA. In this example the current is 1mA. With r_π = 4k ohms, r_x =10 ohms, V_A = 200V, and using the values from the schematic, and a npn transistor with a gain equal to 200, the amplifier gain is calculated.

$$r_o = (V_A / I_C) = \frac{200}{1\times10^{-3}} = 200\times10^3$$

$$R'_L = 6k \| 4k \| 200k = 2.37\times10^3$$

$$R_{in} = 8k \| 4k \| \langle 4k + 10 \rangle = 1.6k$$

$$A_o = \left(((1.6k) / (1.6k + 4k)) \times 0.0385 \times 2.37\times10^3 \right) = 26$$

The derived value of A_o is used as the gain value in the simple functional model of Listing 5.2.

LISTING 5.2 Verilog-A module definition for common emitter amp

```
`include "std.va"

module ceamp_fm(in, out);
    inout in, out;
    electrical in, out;

    parameter real gain = 1.0;

    analog begin
        V(out) <+ V(in)*(-gain);
    end

endmodule
```

The gain of the amplifier can be selected with parameters passed into the model from the test bench (Spice circuit file) or from another Verilog-A module. If a parameter is not specified during instantiation, the default value declared in the behavioral model file is used. In this example the parameter `gain` is specified in the Spice circuit file as,

```
xa1 in out ceamp_fm gain = 25
```

and the default value declared in the Verilog-A model file,

```
parameter real gain = 1.0;
```

System performance can be easily studied with various amplifier gain values by choosing the value in the Spice circuit file, without having to rewrite and test the model. The final transistor level circuit design can then be completed and characterized with a final gain selected for optimum system level performance. In this example the gain of the behavioral model was selected to be 25 to match results with Spice simulations. The results from the transistor-level Spice and functional model simulations are shown in Figure 5.4.

5.2.2 Modeling Higher-Order Effects

Modeling higher-order effects in the common emitter amplifier to account for the frequency response, requires developing an intuitive understanding of the circuits general behavior. For example, in the amplifier, the input and output capacitors will appear as near open circuits at 0 Hz, and as near short circuits at high frequencies

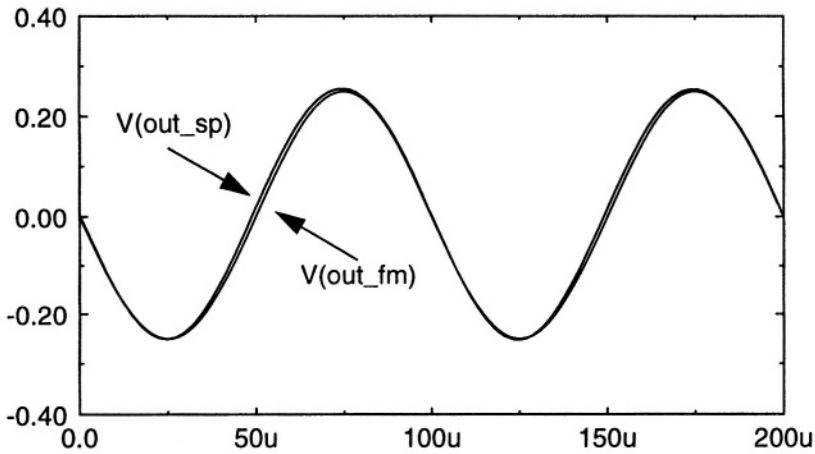


FIGURE 5.4 Time domain analysis and results for the transistor-level Spice `V(out_sp)` and functional Verilog-A model `V(out_fm)` for the common emitter amplifier.

(although the capacitors do have leakage and resistive components which is not factored in). The first order basic equation for the gain of the common emitter amplifier, as a function of frequency is:

$$A(s) = A_M \left(\frac{1}{1 + \frac{s}{\omega_H}} \right) \left(\frac{s^2 (s + \omega_Z)}{(s + \omega_{P1})(s + \omega_{P2})(s + \omega_{P3})} \right)$$

Fortunately, there is a dominant low frequency zero which allows the equation to be simplified, yielding one easier to use while adequately representing the behavior:

$$A(s) = A_M \left(\frac{1}{1 + \frac{s}{\omega_H}} \right) \left(\frac{s}{s + \omega_L} \right)$$

The effective emitter resistance, as a function of circuit and npn transistor parameters, is required in the low frequency zero calculation. It is dependent upon the following relationship:

$$R_E' = ((R_{B1} \| R_{B2} \| R_s) + r_\pi)/\beta_{ac}$$

The dominant low frequency zero, as a result of the effective resistance between the emitter and ground, is given by the following equation:

$$\omega_L = 1/\left(C_E \cdot R_E'\right)$$

In a similar fashion, the dominant high frequency pole is approximated by the following relationship:

$$\omega_H = 1/((R_s \| R_{in})\left(C_\pi + C_\mu\left(1 + g_m R_L'\right)\right))$$

And, as discussed during midband model development, the term g_m is the transistor transconductance, the parameters r_π and r_x are the dominant intrinsic npn resistances, and the values C_π and C_μ are the dominant npn capacitance for the design. Discrete values from the schematic and parameters from the midband gain analysis are used in the calculation of ω_L and ω_H with the following results:

$$R_E' = (((4k \| 8k \| 4k) + 4k)/200) = 28\Omega$$

$$\omega_L = \frac{1}{10uf \times 28} = 568Hz$$

$$\omega_H = 1/((4k \| 1.6k)\left(4.5pf + 3.5pf\left(1 + 38.5\times10^{-3} \times 2.37\times10^{3}\right)\right) = 430kHz$$

These approximations were verified using the transistor-level Spice small-signal AC analysis as shown previously in Figure 5.3 (b).

5.2.3 Structural Model of Behavior

As previously discussed, the gain of the amplifier is not constant with respect to frequency because of parasitic capacitances of the transistor, AC coupling capacitors, and the bypass capacitors. A structural/behavioral model, based upon a classical RC network using the Verilog-A language, can be used to model the amplifier as a function of frequency. The simple RC network, followed by the gain stage, is used to model gain and frequency response characteristics of common emitter amplifier (Figure 5.5). The gain of the amplifier has been modified to 25 to match the characteristics

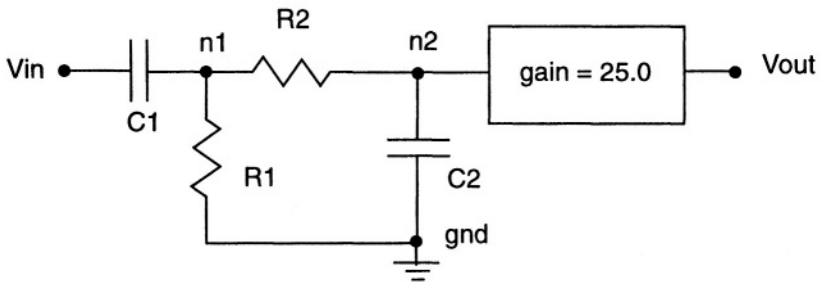


FIGURE 5.5 RC network for modeling the frequency domain response of the behavioral model.

of the Spice model. The other values for the structural RC network were selected by first setting resistor R1 to 4000 ohms, calculating the capacitance C1 at ω_L and then adjusting the value until the simulation results of the structural model matched the transistor-level Spice simulation. The calculated value of the capacitance C1 is 68nF.

$$C1 = \frac{1}{2\pi f R1} = \frac{1}{2\pi \times 568 \times 4k}$$

The value of Cl was tuned to l00nF for use in the simplified behavioral model. The same procedure is use to determine R2 and C2. Resistor R2 is selected to be 100k, capacitor C2 was calculated at ω_H, and tuned to match Spice results. The final value for C2 is 2.8pF.

The resulting Verilog-A model file is shown in Listing 5.3. Internal nodes are declared within the module for the RC network. The **analog** block is used to implement the behavior.

LISTING 5.3 Verilog-A module of ce-amp w/RC bandpass filter.

```
`include "std.va"

module mbce_amp_rcn(in, out, gnd);
    inout in, out, gnd;
    electrical in, out, gnd;

    parameter real gain = 1.0;
    parameter real r1 = 4k;
    parameter real c1 = 100n;
```

```
    parameter real r2 = 100k;
    parameter real c2 = 2.8p;

    electrical n1;
    electrical n2;

    analog begin
        I(in, n1) <+ c1*ddt (V(in, n1));
        V(n1, gnd) <+ r1*I(in, n1);
        I(n1, n2) <+ V(n1, n2)/r2;
        I(n2, gnd) <+ c2*ddt(V(n2, gnd));
        V(out, gnd) <+ V(n2, gnd)*(-gain);
    end

endmodule
```

Transient analysis of the common emitter amplifier based on the RC bandpass network for a sinusoidal input is shown in Figure 5.6 (a). Figure 5.6 (b) shows the magnitude of the frequency response. Both are compared to the transistor-level simulations and exhibiting the expected behavior.

5.2.4 Behavioral Model

The Verilog-A language includes built-in Laplace transform functions that implement lumped linear continuous-time filters. This transform is used to model the frequency effects of the amplifier by treating the behavior as a simple bandpass filter,

The **laplace_nd** analog operator is used in this behavioral model to provide the behavior of the gain with respect to frequency. The simplified form of the frequency response equation for the amplifier is expanded and coefficients are calculated for use in the **laplace_nd** analog operator.

$$A(s) = A_M \left(\frac{1}{1 + \frac{s}{\omega_H}} \right) \left(\frac{s}{s + \omega_L} \right)$$

$$(A(s))/A_M = \frac{s}{\omega_L + \langle 1 + \frac{\omega_L}{\omega_H} \rangle s + \langle \frac{1}{\omega_H} \rangle s^2}$$

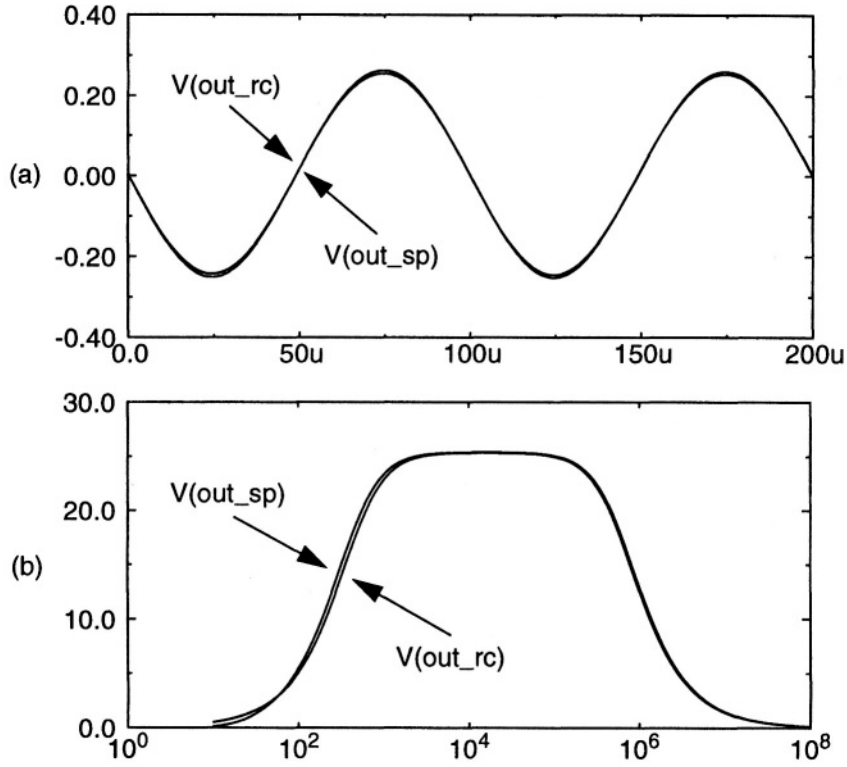


FIGURE 5.6 Time domain analysis (a) and small-signal AC magnitude response (b) results for Verilog-A RC model, `V(out_rc)` of Listing 5.3 and the Spice transistor-level model, `V(out_sp)` of the common emitter amplifier circuit.

where,

$$\omega_L = 2\pi \times 568 \times 3k = 3.6k$$

$$\langle 1 + \frac{\omega_L}{\omega_H} \rangle = \langle 1 + \frac{3.6k}{2.7meg} \rangle = 1.001$$

$$\frac{1}{\omega_H} = \frac{1}{2\pi \times 430k} = 3.7 \times 10^{-7}$$

are used as the starting point values for the coefficients in the denominator expression. The coefficients were then tuned to match Spice transistor-level simulations.

LISTING 5.4 Verilog-A model of ce-amp using laplace analog operators

```
`include "std.va"

module com_emtr_amp_lp(in, out, gnd);
    inout in, out;
    electrical in, out;

    parameter real gain = 1.0;

    analog begin
        V(out, gnd) <+ -gain*laplace_nd( V(in),
            { 0.0, 1.0 }, // numerator zeros
            { 3.6k, 1.001, 3.7e-7 } ) ; // denomenator poles
    end

endmodule
```

After curve-fitting to the transistor-level Spice reference simulation results, the coefficients for the denominator of the **laplace_nd** analog operator of the model in Listing 5.4 become 2K, 1.001, and 2.7e-7 respectively. The transient and small-signal AC

analysis for the resulting behavioral model is shown in Figures 5.8 (a) and (b) respectively.

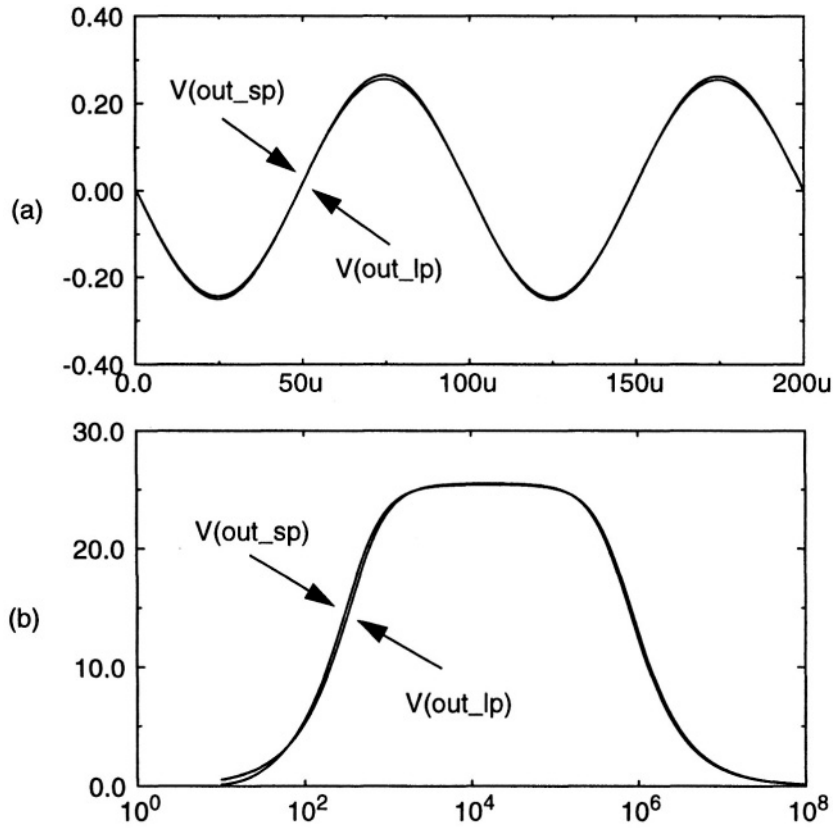


FIGURE 5.7 Time domain analysis (a) and small-signal AC magnitude response (b) results for Verilog-A behavioral model, `V(out_lp)` of Listing 5.4 and the Spice transistor-level model, `V(out_sp)` of the common emitter amplifier circuit.

5.3 A Basic Operational Amplifier

Operational amplifiers are key building blocks for the analog functions used within signal processing systems. The basic configuration of this op amp as shown in Figure 5.8, is a voltage-to-current converter followed by an inverting voltage amplifier which drives a current output buffer. The amplifier gain is provide by the first and second stages. The first stage converts a differential input voltage to a single ended output current which drives the second gain stage. The bypass capacitor C_c, around the second stage, ensures stable operation within the intended frequency range of operation by bypassing higher frequencies around the gain stage, reducing the gain to zero at some high frequency value. The bypass capacitor C_c will also set the slew rate, or the maximum rate of change reflected in the output for any given step at the input of the amplifier, since it must be charged and discharged with current from the input amplifier stage.

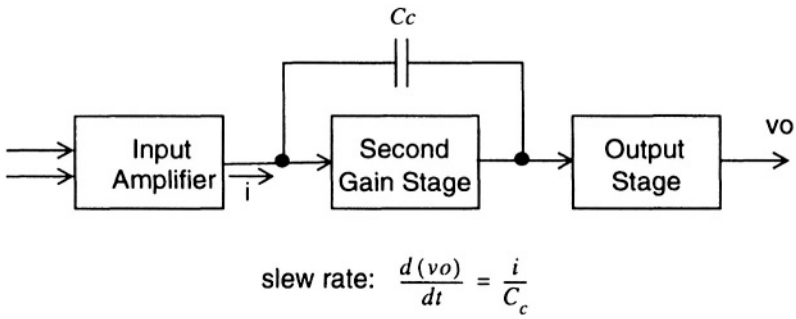


FIGURE 5.8 Basic three stage architecture of an op amp. A differential input amplifier feeds a second gain stage that drives a current output stage.

5.3.1 Model Development

A variety of modeling levels can be used to describe the operational amplifier. These range from a simple functional model with a gain equation to sophisticated models with pole and zero effects, as well as noise behavior, offset and drift effects.

The first example uses a simple model useful for top-level architectural studies. The symbols to represent the behavior of the basic stages of the op amp are shown in Figure 5.9.

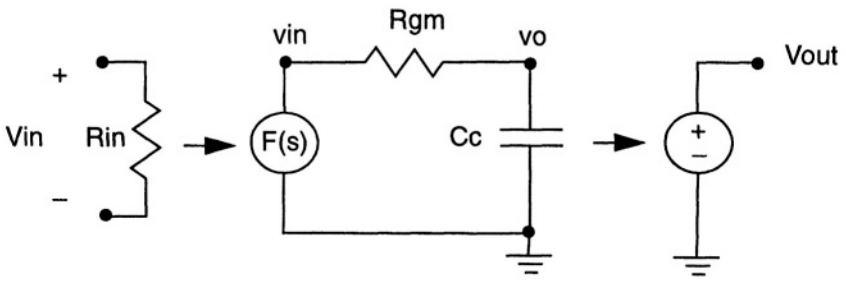


FIGURE 5.9 Conceptual model of the stages in the behavioral model for the operational amplifier.

The input impedance for the op amp is modeled with a simple resistor across the differential input. The frequency behavior of the first gain stage is represented by passing the signal through a laplace transform function filter. The dominant pole introduced in the second gain stage is modeled using the analog operator in conjunction with a resistor and a capacitor $R_{gm} \cdot C_c$. Note the slew rate of the model is the rate at which the capacitor can be charged and discharged in this RC low pass filter. The voltage-controlled voltage source is used to create a zero-impedance output stage with infinite sourcing capability. In higher-level models the output stage usually contains output impedance and output voltage swing limitation characteristics. These effects are not included in this model.

The Verilog-A module definition of the op amp using the conceptual model is shown in Listing 5.5.

LISTING 5.5 Verilog-A model of the operational amplifier

```
`include "std.va"
`include "const.va"

module opamp(inm, inp, out);
    inout inm, inp, out;
    electrical inm, inp, out;

    parameter real gain = 250k;
    parameter real rgm = 2.3k;
    parameter real cc = 30p;
    parameter real rin = 2Meg;
```

```
    electrical vin, vo;

    analog begin
       I(inp, inm) <+ V(inp, inm)/rgm;
       V(vin) <+ laplace_nd(gain*V(inp, inm),
          { 1.0 },{ 1.0, 5.0e-7 });
       I(vin, vo) <+ V(vin, vo)/rgm;
       I(vo) <+ ddt(cc*V(vo));
       V(out) <+ V(vo);
    end

endmodule
```

For the purpose of the Verilog-A model development, a test structure symbol, as shown in Figure 5.10, is utilized to encapsulate the amplifier. In addition to develop-

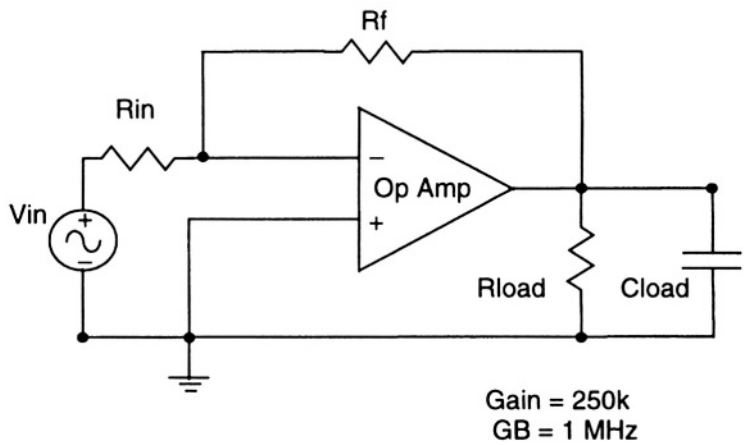


FIGURE 5.10 Spice schematic of the operational amplifier test bench used to develop and characterize the model.

ing the behavioral model, the Verilog-A language is used in the characterization of the

operational amplifier. Here, we will develop a module that sets up the step input and measures the settling time at the output of the op amp.

Listing 5.6 shows the circuit file for the operational amplifier circuit used to test the model:

LISTING 5.6 Spice netlist of the operational amplifier test bench

```
* basic operational amplifier

.verilog  "op_amp.va"

Vb inp 0 dc 0
Vin 1 0 dc 0 ac 1 sin(0 10m 1k 0 0)

xamp1 inm inp out 0 opamp

Rin 1 inm 10k
Rf inm out 100k
Rload out 0 100k
Cload out 0 20p

.op
.ac dec 100 0.1 10Meg
.tran 10u 3m

.end
```

The magnitude response of the op amp for AC small-signal simulation results of the operational amplifier are shown in Figure 5.11. The bode plot shows both the low and high-frequency poles of the op amp.

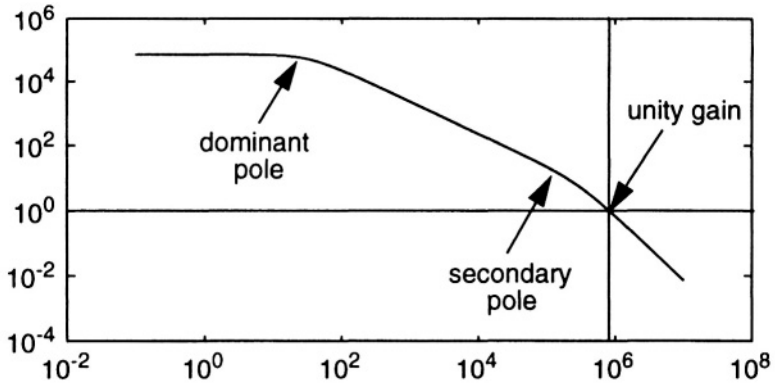


FIGURE 5.11 AC small-signal magnitude response of the behavioral model of the operational amplifier.

The op amp transient response to a sinusoidal input verifying the functionality of the model is shown in Figure 5.12.

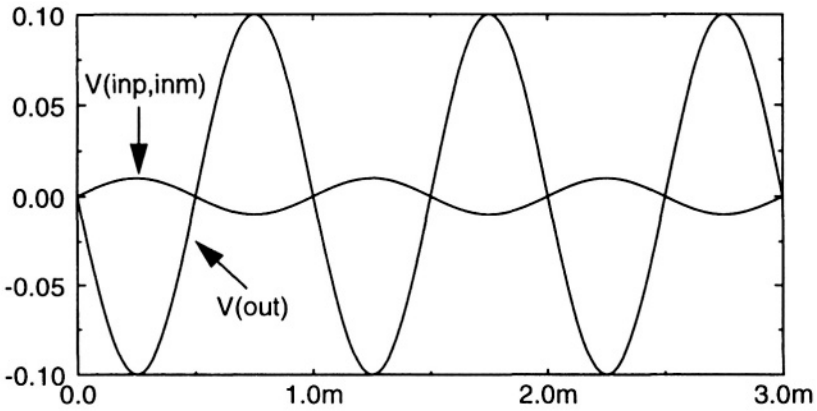


FIGURE 5.12 Time domain response of the behavioral model of the operational amplifier.

5.3.2 Settling Time Measurement

Measuring the settling time of an operational amplifier can be automated by development of a Verilog-A module that acts as a test bench for the device under test. The conceptual approach is illustrated in Figure 5.13. A measurement module sets up a

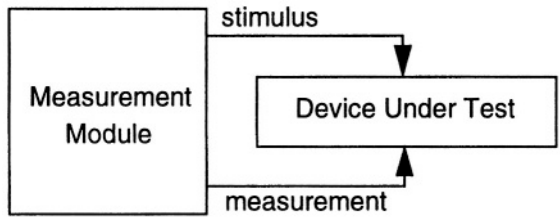


FIGURE 5.13 Set up for measuring the settling time of an op amp.

stimulus to the Device Under Test (or DUT) under known conditions and records the results. At the end of the simulation, the measure results are summarized.

The module (Listing 5.7) performs the measurement using a **`timer`** analog operator to initialize a step on the stimulus signal that provides the input to the operational amplifier. The measurement module then senses the crossings of the output of the module when they are within +/- 5% of the final state value. The times of the crossings are recorded, the settling time being the difference between the latest crossing time and the start of the stimulus step input.

LISTING 5.7 Verilog-A model of settling-time test bench measurement

```
module settling_test(stim, meas);
    inout stim, meas;
    electrical stim, meas;

    parameter real vstep = 5.0;
    parameter real tstart = 1.0u;
    parameter real interval = 10.0u;

    real vstim;
    real last;

    analog begin
    // generate stimulus
```

```
        @(timer(tstart)) begin
           vstim = vstep;
           last = tstart;
        end
        V(stim) <+ transition(vstim, 0.0, 1.0n, 1.0n);

        // measure results - op amp is in inverting
        @(cross(V(meas) - 1.05*vstep, -1.0)) begin
           last = $realtime();
        end
        @(cross(V(meas) - 0.95*vstep, +1.0)) begin
           last = $realtime();
        end

        // report at end of measurement interval.
        @(timer(interval)) begin
           $strobe("settling time = %g s.",
              last - tstart);
        end
     end

endmodule
```

The result of the measurement is printed to the standard output using the $**strobe**

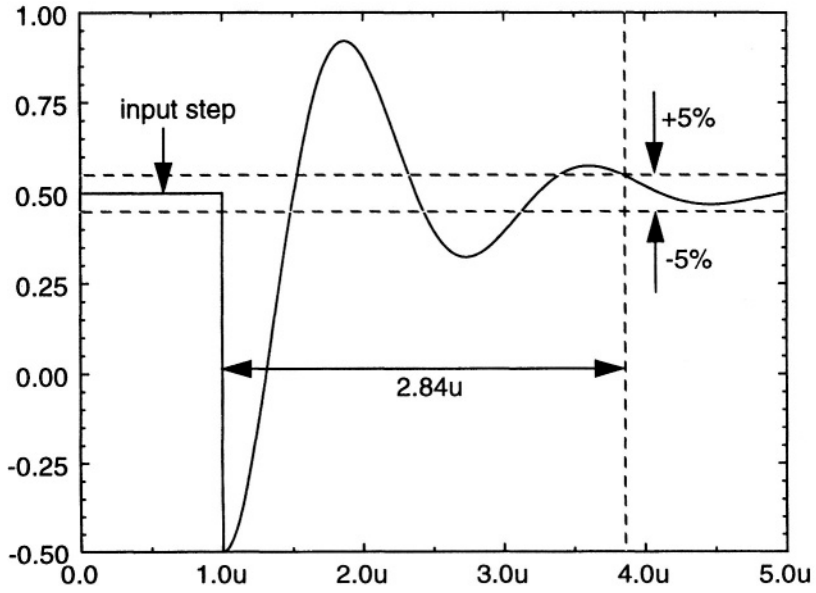


FIGURE 5.14 Schematic and waveforms showing the measurement process for the settling time measurement of the op amp.

system task. The results of the measurement simulation are shown in Figure 5.14. The results for the simulation are recorded to the standard output as:

```
settling time = 2.84e-06 s.
```

5.4 Voltage Regulator

The architecture of the voltage regulator (Figure 5.15), is composed of a bandgap reference model, the operational amplifier model from Section 5.3, a module to represent the current of the op amp, and a switch model. The bandgap reference circuit which a curve-fitting equation to define the output voltage. The equation includes the effect of supply voltage and temperature variation. The equation was derived from extrapolated data obtained from transistor-level Spice simulations, traceable to actual

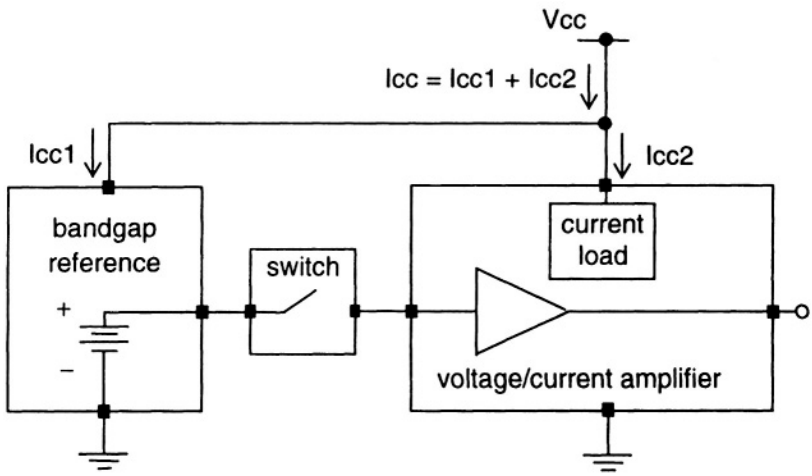


FIGURE 5.15 Voltage regulator composed of modules for a bandgap reference, current switch, op amp, and op amp current monitor.

silicon behavior. The reference is connected to an amplifier module that boosts the voltage to 3.3 volts and provides the current necessary to drive the load. A switch model connects the reference voltage to the amplifier so that the voltage can be switched to zero for power down applications. The switch model was reused in the current load model that represents the current required by the amplifier. This illustrates how another module can be added to include the current loading requirements of behavioral models. Separate modules can be used, or the behavior can be included in the functional definition of the module, depending upon design requirements and modeling methodology.

At the heart of most voltage regulators in integrated circuit design is a bandgap voltage reference (Listing 5.8). The voltage is proportional to V_{GO}, the band-gap voltage of silicon, and a thermal voltage V_{TO} evaluated at a given temperature. With careful design a low voltage drift with respect to temperature can be maintained throughout a given temperature range. Theoretical calculations can predict the first order behavior of the generated voltage, but secondary effects due to differing manufacturing processes must be considered, and the model tuned to match performance.

LISTING 5.8 Verilog-A model of bandgap reference

```
// bandgap reference
`include "std.va"
```

```
`include "const.va"

module bandgap(vcc, vbg, temp);
   inout vcc, vbg, temp;
   electrical vcc, vbg, temp;

   parameter real vbg_nom = 1.0;
   parameter real icc_nom = 10.0e-6;
   real tempC_swp;
   real vcc_appl;

   analog begin
      tempC_swp =(V(temp) - 27.0)/27.0;
      vcc_appl =(V(vcc) - 5.0)/5.0;
      I(vcc, gnd) <+ (icc_nom - 2.78e-8*tempC)*
          (1.0 + (0.01*(V(vcc) - 2.0)));
      V(vbg) <+ vbg_nom - 0.0008 +
          1.1m*vcc_appl - 0.5m*tempC_swp*tempC_swp;
   end

endmodule
```

The Verilog-A equation for the voltage and the cell current are:

```
I(vcc, gnd) <+ (icc_nom - 2.78e-8*tempC)*
    (1.0 + (0.01*(V(vcc) - 2.0)));
V(vbg) <+ vbg_nom - 0.0008 +
    1.1m*vcc_appl  -  0.0005*tempC_swp*tempC_swp;
```

The switch model in the voltage regulator uses a control signal to change the characteristics of its output branch (Listing 5.9). The model monitors changes in the control signal with the use of the cross analog operator. The control signal is then compared to its threshold value to determine the state of the switch. The variable that is set is then used to set either the voltage or current condition at the output branch, `I(vp, vn)` or `V(vp, vn)` depending upon the switch state. Note the use of preprocessor defines (`` `OPEN `` and `` `CLOSED ``) to help document the module description.

LISTING 5.9 Verilog-A simple switch model.

```
`define OPEN 1
`define CLOSED 0
```

```
module sw(vp, vn, vctrlp, vctrln);
    inout vp, vn, vctrlp, vctrln;
    electrical vp, vn, vctrlp, vctrln;

    parameter real vth = 2.0;

    integer sw_state;

    analog begin
        @(cross(V(vctrlp, vctrln) - vth, 0.0, 1u)) ;

        if ((V(vctrlp, vctrln) - vth) > 0.0)
            sw_state = `CLOSED;
        else
            sw_state = `OPEN;

        if (sw_state == `OPEN)
            V(vp, vn) <+ 0.0;
        else
            I(vp, vn) <+ 0.0;
    end

endmodule
```

The op amp current load model is used to generate a current, `Icc2`, to represent the cell current of the amplifier. The current load module, `icc`, is derived from the switch module (Listing 5.10), excepting in this case the objective is to sink 20.0u of current to represent the load of the op amp during the switching on condition.

LISTING 5.10 Current load module.

```
`define OPEN 1
`define CLOSED 0

module icc(vcc, vctrlp, vctrln);
    inout vcc, vctrlp, vctrln;
    electrical vcc, vctrlp, vctrln;

    parameter real vth = 2.0;
```

```
    integer sw_state;

    analog begin
       @(cross(V(vctrlp, vctrln) - vth, 0.0, 1.0u)) ;

       if ((V(vctrlp, vctrln) - vth) > 0.0)
          sw_state = `CLOSED;
       else
          sw_state = `OPEN;

       if (sw_state == `OPEN)
          I(vcc) <+ 0.0;
       else
          I(vcc) <+ 20.0u;
    end

endmodule
```

The `icc` module can be part of the amplifier module or maintained separately, depending upon the design style, reuse considerations, and model methodology. By including the cell current, we can accurately capture the total amount of current used within the system. For example, during architectural studies the current can be monitored to help select various configurations for the system. The current for the bandgap reference, `Icc1`, is included in the bandgap model and is constant because the cell is not switched off in this application.

5.4.1 Test Bench and Results

Circuit file for the bandgap reference circuit used to test the model is shown in Listing 5.11. The circuit file includes tests for temperature, supply voltage, and the switching characteristics.

LISTING 5.11 Spice netlist of bandgap reference test bench

```
* voltage regulator circuit.

.verilog "bandgap.va"
.verilog "sw.va"
.verilog "icc.va"
```

```
.verilog "op_amp.va"

vcc1 vcc 0 dc 3.0
vc cntrl 0 dc 2.5 pulse(0 2.5 0.1m 50u 50u 2.5m 5m)
vtemp temp 0 dc 25.0

xbg vcc vbg temp bandgap vbg_nom=1.295 icc_nom=19u
xicc vcc cntrl 0 icc vth=1.0
xsw vbg inp cntrl 0 sw vth=1.0
xamp1 inm inp vout opamp

* amplifier biasing
Rf2 inm vout 100k
Cf inm vout 150p
Rf1 inm 0 61k
Rinp inp 0 30k
Rload vout 0 5k

.op
.dc vtemp -40 140 1
.dc vcc 4 6.1
.tran 1u 6m

.end
```

The output voltage is designed for a nominal bandgap voltage of 1.259 volts. The results of simulation using the Verilog-A model over a temperature range of -40 to

180 degrees Celsius is shown in Figure 5.16. The "bow" in the voltage with respect to temperature is a typical characteristic for bandgap based voltage reference.

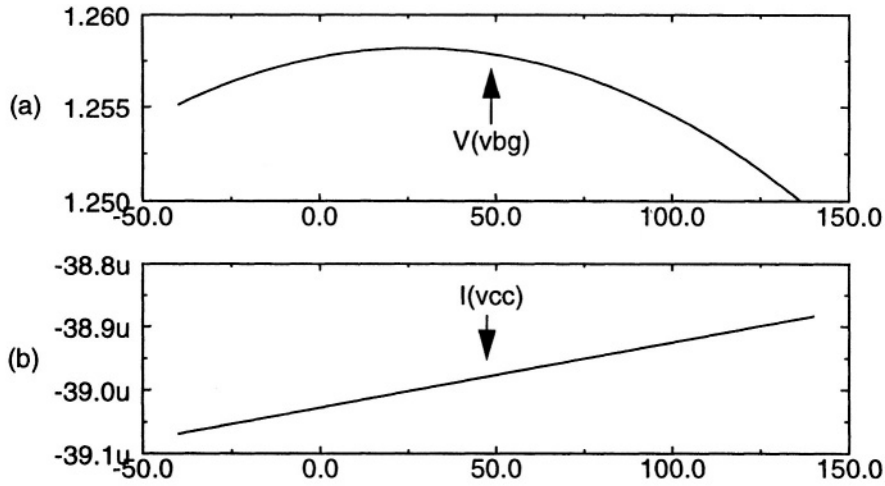


FIGURE 5.16 Bandgap reference voltage (a) and total current (b) over temperature for the behavioral model.

Figure 5.17, shows the output voltage and the cell current as the supply voltage is varied over the expected range of usage from 4 to 6 volts.

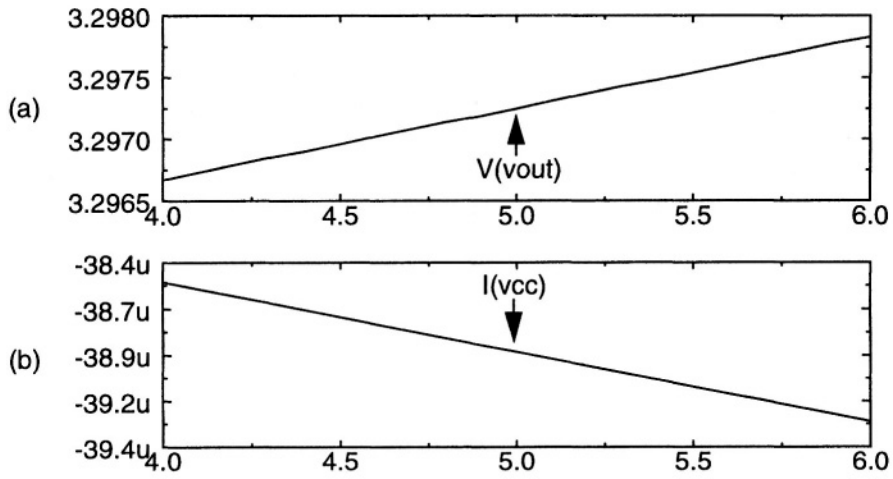


FIGURE 5.17 Regulator output voltage (a) and total current (b) over the operating supply range.

Figure 5.18, testing the dynamic response of models, verifies the output voltage as a function of the switch current between the reference and the buffer amplifier.

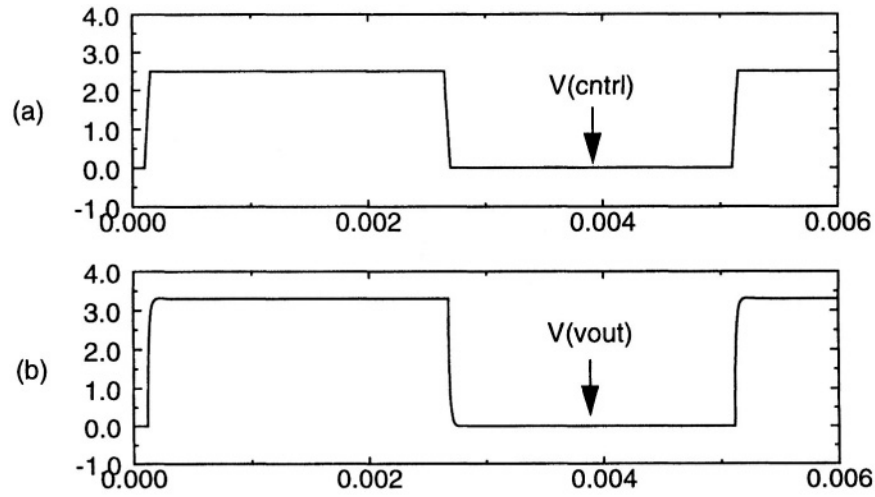


FIGURE 5.18 Voltage regulator control signal (a) and output voltage (b).

5.5 QPSK Modulator/Demodulator

Quadrature phase-shift keying, or QPSK, is an example of a modulation technique in which the information carried within the signal is contained in the phase. The phase of the signal can take on one of four values, such as shown in the constellation diagram of Figure 5.19.

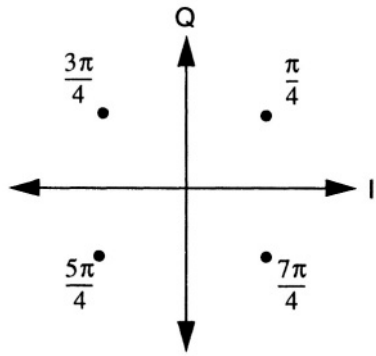


FIGURE 5.19 QPSK modulator constellation diagram showing the allowable states between the in-phase and quadrature components.

5.5.1 Modulator

As shown in the QPSK modulator schematic (Figure 5.20), the incoming binary sequence is transformed into polar form by a nonreturn-to-zero (NRZ) encoder. Here, the binary 1 and 0 symbols are transformed into +1 and -1 respectively. The NRZ data stream is de-multiplexed into two separate binary sequences consisting of the odd- and even-numbered input bits. These binary sequences are used to modulate a pair of quadrature carriers, which are added together to produce the QPSK signal.

The QPSK modulator module consists of two primary components for the polarization of the input data sequence and the modulation of the quadrature components to produce the QPSK signal. The modulator samples the input data stream (0s and 1s) and converts it to the corresponding -1 or +1 every `period` seconds using the **`timer`** operator. An integer variable `state` is toggled to convert the serial data stream into two parallel streams for modulating the quadrature carriers.

LISTING 5.12 Verilog-A module definition of QPSK modulator

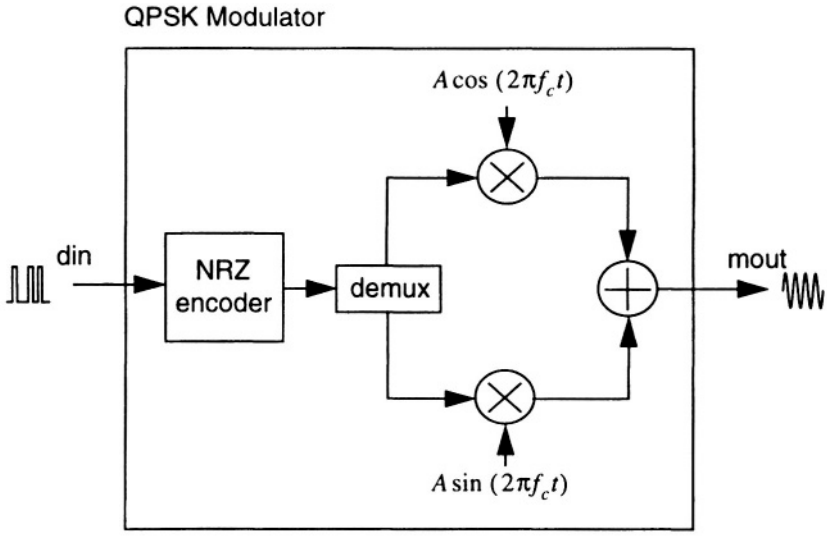


FIGURE 5.20 Schematic representation of the QPSK modulator. The input data stream is non-return to zero (NRZ) encoded, de-multiplexed between the I and Q channels, modulated and summed for the output.

```
module qpsk_mod(out, in);
    inout out, in;
    electrical out, in;

    parameter real offset = 0.0n;
    parameter real period = 100.0n;
    parameter real oscfreq = 2.0e7;

    real an, bn, bnm1;
    integer state;

    analog begin
        @(timer(offset, period)) begin
            if (state == 0) begin
                an = (V(in) > 2.5) ? 1.0 : -1.0;
                bn = bnm1;
            end else begin
```

```
            bnm1 = (V(in) > 2.5) ? 1.0 : -1.0;
        end
        state = !state;
    end

    V(out) <+ (1.0/sqrt(2.0))*
        (an*cos(2.0*`M_PI*oscfreq*$realtime()) +
        bn*sin(2.0*`M_PI*oscfreq*$realtime()));

    bound_step(0.05/oscfreq);
end

endmodule
```

To insure that an accurate representation of the QPSK signal is generated, the simulation timestep is bounded to require a minimum of 20 points per oscillator period using the **bound_step** function. The **bound_step** function acts to limit the timestep utilized during the simulation. Its primary use is for the accurate generation of independent sources such as the modulator. In this case,

```
bound_step(0.05/oscfreq);
```

limits the timestep used in the representation of the modulated signal to a minimum of 20 points per the period of the oscillator (Figure 5.21).

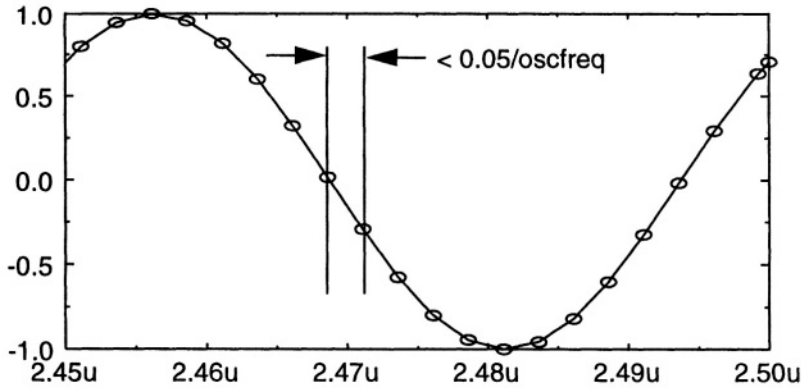


FIGURE 5.21 Use of **bound_step** for insuring accurate representation of custom sources.

The output of the modulator (shown in Figure 5.22), shows the constant-envelope modulator output with the phase transitions at the changes in the input data sequence.

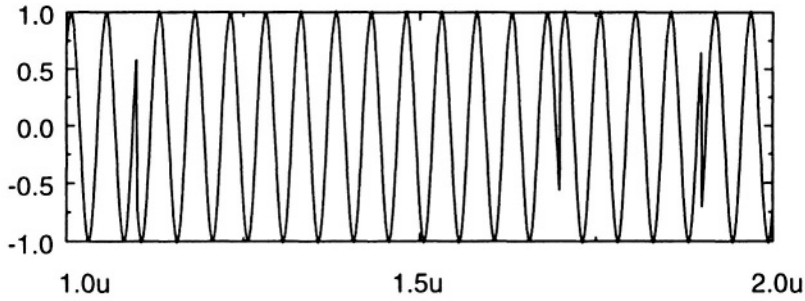


FIGURE 5.22 Time domain output of the QPSK module for some random bit input sequence.

5.5.2 Demodulator

The QPSK demodulator, shown in Figure 5.23, consists of a pair of correlators supplied with a locally generated pairs of reference signals. The outputs of the correlators, x_i and x_q, are compared to a threshold of zero for their respective in-phase and quadrature channel outputs. For the in-phase channel, if $x_i > 0$, then a decision is made in favor of symbol 1. Likewise, if $x_i < 0$, then a decision is made in favor of the symbol 0. A similar process occurs for the quadrature channel. The two binary

sequences are combined in a parallel-to-serial converter to produce the original binary input sequence.

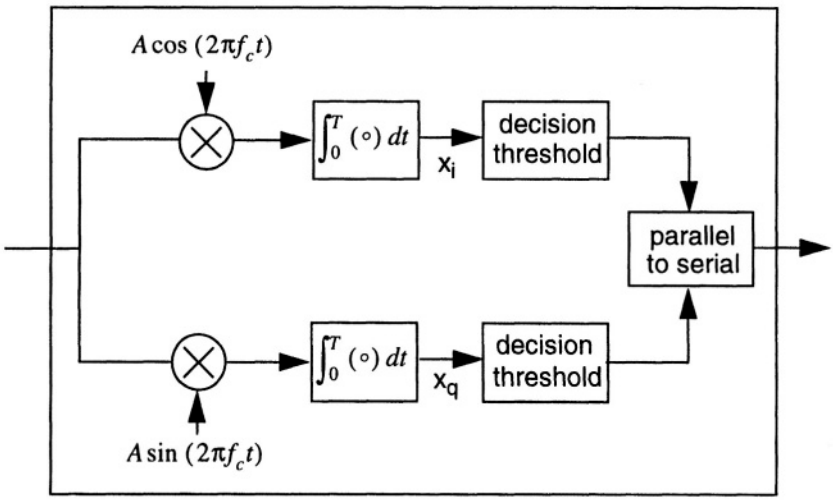


FIGURE 5.23 Schematic representation of the QPSK demodulator. The incoming signal is mixed with two locally-generated orthogonal signals, integrated to determine the symbol.

LISTING 5.13 Verilog-A definition of QPSK demodulator.

```
module qpsk_demod(out, in);
    inout out, in;
    electrical out, in;

    parameter real offset = 0.0n;
    parameter real period = 100.0n;
    parameter real oscfreq = 2.0e7;

    real x_i, x_q;
    real v_i, v_q;
    real d_i, d_q;
    real bout;
    integer integreset;
    integer state;
```

```
    analog begin
       v_i  =  V(in)*cos(2.0*`M_PI*oscfreq*$realtime());
       v_q  =  V(in)*sin(2.0*`M_PI*oscfreq*$realtime());

       integreset = 0;
       @(timer(offset, period)) begin
          integreset = 1;
          d_i = (x_i > 0.0) ? 5.0 : 0.0;
          d_q = (x_q > 0.0) ? 5.0 : 0.0;
       end

       x_i = idt(v_i, 0.0, integreset);
       x_q = idt(v_q, 0.0, integreset);

       @(timer(offset, period)) begin
          if (state == 0) begin
             bout = d_i;
          end else begin
             bout = d_q;
          end
          state = !state;
       end

      V(out) <+ transition(bout, 1n, 1n, 1n);
    end

endmodule
```

The timer analog operator is used to sample the output of the quadrature correlators at the symbol period rate. The real variables `x_i` and `x_q` are used to store the correlator outputs from the previous evaluation time. At the same time the correlator outputs are sampled, the variable `integreset` is set to `1`, causing the correlators to be reset to the specified initial condition (`0.0`) .

```
       integreset = 0;
       @(timer(offset, period)) begin
          integreset = 1;
          d_i = (x_i > 0.0) ? 5.0 : 0.0;
          d_q = (x_q > 0.0) ? 5.0 : 0.0;
       end
```

```
x_i = idt(v_i, 0.0, integreset);
x_q = idt(v_q, 0.0, integreset);
```

A more detailed model of the demodulator would extract the timing information from the incoming signal and use that to synchronize the symbol extraction.

Resetting the integrators at the symbol period implements an integrate-and-dump algorithm for determining the symbol thresholds as shown in Figure 5.24.

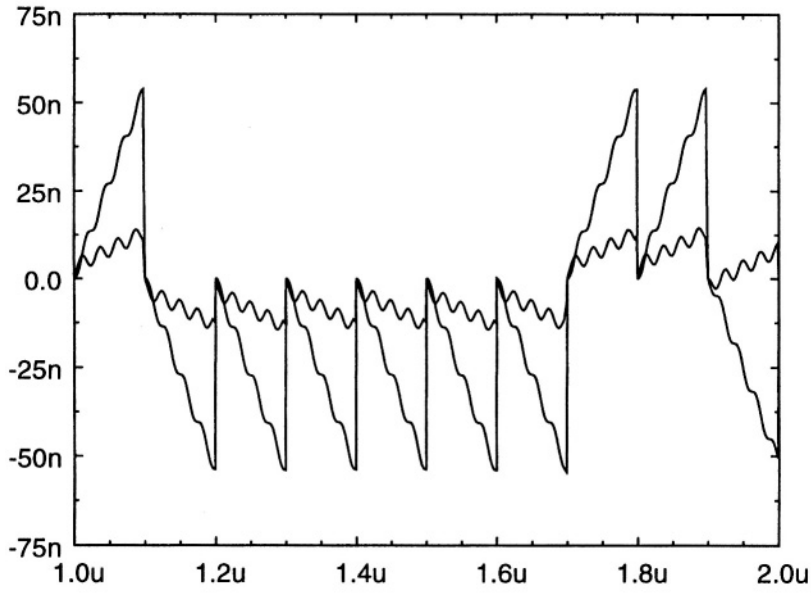


FIGURE 5.24 Output of demodulator integrators for some random bit sequence.

5.6 Fractional N-Loop Frequency Synthesizer

This example illustrates design and analysis of a N.F frequency synthesizer, where N is the integer multiple of the number and F is the fractional portion that the synthesizer multiplies its input signal by.

The architecture, shown in Figure 5.25, consists of a divide-by-N frequency synthesizer, augmented to provide fractional loop division. The fractional loop division is carried out by removing pulses (module PR) prior to the divide-by-N counter which feeds the phase detector. A pulse is removed whenever the accumulator (module ACCUM) detects that the number of reference clock pulses times the fractional part exceed one. To adjust for the phase error that occurs due to the missing pulses, the accumulator generates an offset term that is summed in with the VCO control signal.

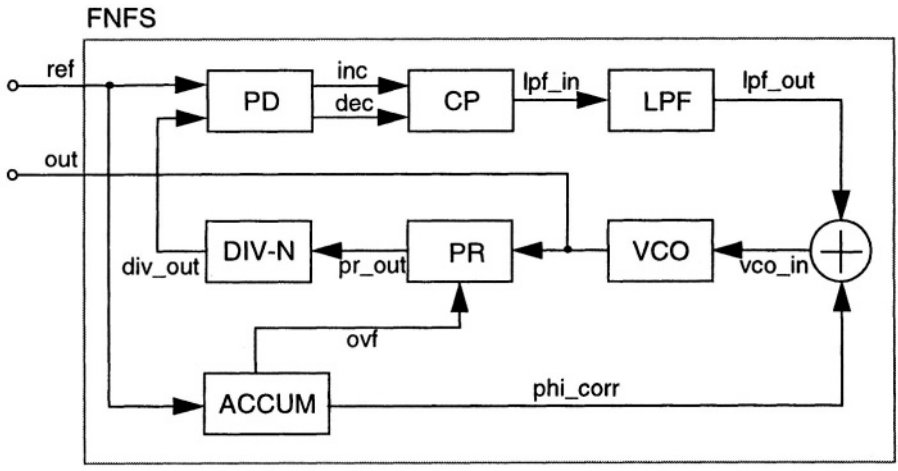


FIGURE 5.25 Schematic representation of the fractional-N frequency synthesizer.

The structural definition of the fractional n-loop frequency synthesizer is shown below. The `resistor` and `capacitor` instantiations that constitute the low-pass filter use simulator built-in primitives (see the test bench Listing 5.14 for their definitions).

LISTING 5.14 Verilog-A definition for the structural module of the frequency synthesizer.

```
`include  "std.va"
`include  "const.va"

module fnfs(out, in, gnd);
    inout out, in, gnd;
    electrical out, in, gnd;
```

```
    parameter integer n = 1;
    parameter integer f = 1;

    accu #(.fract(f), .tdel(2n), .trise(2n), .tfall(2n))
       xaccu(phase_offset, overflow, in);
    fpd #(.tdel(2n), .trise(2n), .tfall(2n))
       xpd(in, ndiv_out, inc, dec);
    cp #(.cmag(.2m), .tdel(1n), .trise(5n), .tfall(5n))
       xcp(inc, dec, filt_in, gnd) ;
    sum
       xsum(vco_in, filt_in, phase_offset);
    dvco #(.fc(20e6), .gain(2e6), .tdel(10n), .
       trise(2n), .tfall(2n)) xvco(vco_in, out);
    pulrem #(.tdel(10n), .trise(2n), .tfall(2n))
       xpulrem(rem_out, out, overflow);
    divbyn #(.ratio(n))
       xdivbyn(ndiv_out, rem_out);

    capacitor #(.cap(50p)) c1(filt_in, comm);
    resistor #(.resis(10k)) r1(comm, gnd);
    capacitor #(.cap(5p)) c2(comm, gnd) ;
endmodule
```

5.6.1 Digital VCO

The digital vco defines a relationship between its input voltage and output frequency as follows:

$$T^{-1} = \int (f_c + k_v V(in))\, dt$$

The algorithm used must have two discernible states that can be used to drive the vco output. Consider the algorithm represented graphically in Figure 5.26.

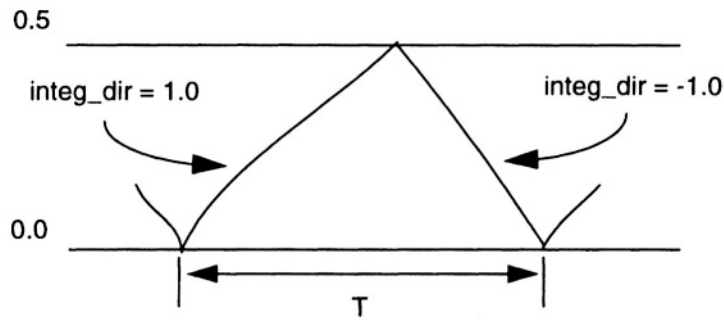


FIGURE 5.26 The algorithm used to define the integration period of the voltage-controlled oscillator.

The period, `T`, is defined by a full cycle of the integration - integrating the characteristic equation from `0.0` to `0.5` and then back to `0.0` again. The direction of the integration is set by the variable `integ_dir`, which is also used to define the output (either `0` or `1`). The implementation of this algorithm for the `VCO` is shown in Listing 5.15.

LISTING 5.15 Verilog-A definition of the digital vco.

```
`include "std.va"
`include "const.va"
`include "logic.va"

module dvco(in, out);
    inout in, out;
    electrical in, out;

    parameter real fc = 200.0;
    parameter real gain = 1.0;

    parameter real tdel = `LF_GATE_PROP_DELAY;
    parameter real trise = `LF_GATE_01_DRIVE_DELAY;
    parameter real tfall = `LF_GATE_10_DRIVE_DELAY;
    parameter real vhigh = `LF_GATE_LOGIC_HIGH;
```

```
    real  vout;
    real period;
    real integ_dir;

    initial  begin
       integ_dir = 1.0;
    end

    analog begin
       period = idt(integ_dir*(fc + kv*V(in));

// catch rising transition.
       @(cross(period - 0.5, +1.0)) begin
          integ_dir = -1.0;
       end
// catch falling transition.
       @(cross(period, -1.0)) begin
          integ_dir = 1.0;
       end

       vout = 0.5*vhigh*(integ_dir + 1.0);
       V(out) <+ transition(vout, tdel, trise, tfall);
    end

endmodule
```

The variable `period` is used to store the value of the integral. Analog events are generated whenever the value of period crosses `0.5` in the positive or upward direction, or `0.0` in the negative or downward direction. At the generation of these events, the output is toggled via the `integ_dir` variable, and the direction of the integration is reversed.

5.6.2 Pulse Remover

The pulse-removing module needs to monitor the overflow signal from the accumulator in order to determine when to remove a pulse from the `vco` output prior to the counter. A flag, `rn`, is used to determine when to signal that an overflow condition has been received. This flag is checked on the next input transition - if set, that transition is effectively ignored. The use of a flag (versus direct clearing of the output

value) allows that an entire pulse will be removed, and not partial pulses. The implementation is shown in Listing 5.16.

LISTING 5.16 Verilog-A definition of pulse-remover

```
`include "std.va"
`include "const.va"
`include "logic.va"

module pulrem(out, in, remove);
   inout out, in, remove;
   electrical out, in, remove;

   parameter real tdel = `LF_GATE_PROP_DELAY;
   parameter real trise = `LF_GATE_01_DRIVE_DELAY;
   parameter real tfall = `LF_GATE_10_DRIVE_DELAY;
   parameter real vthresh = `LF_MID_THRESH;

   real vout_val = 0.0;
   integer rn = 0;

   analog begin
      @(cross(V(in) - vthresh, +1.0)) begin
         vout_val = (rn) ? 0.0 : 5.0;
         rn = 0;
      end
      @(cross(V(in) - vthresh, -1.0)) begin
         vout_val = 0.0;
      end
// set the rn (remove_next) flag on positive
// transitions of the remove signal.
      @(cross(V(remove) - vthresh, +1.0)) begin
         rn = 1;
      end
      V(out) <+ transition(vout_val, tdel, trise,
         tfall);
   end

endmodule
```

5.6.3 Phase-Error Adjustment

The accumulator module is used to both determine the removal of pulses from the vco output for the control loop and to provide the phase-error correction voltage that is required to offset the missing pulse. At each edge of the reference input, a summation register `vsum` is incremented with the fractional loop value. When this value exceeds the equivalent value of `1.0`, the overflow bit is set and `vsum` is set to the remainder. The overflow bit is reset on the next clock cycle of the reference input.

LISTING 5.17 Verilog-A definition of phase adjustment accumulator

```
`include "std.va"
`include "const.va"
`include "logic.va"

module accu(sum, ovf, ref);
   inout sum, ovf, ref;
   electrical sum, ovf, ref;

   parameter real tdel = `LF_GATE_PROP_DELAY;
   parameter real trise = `LF_GATE_01_DRIVE_DELAY;
   parameter real tfall = `LF_GATE_10_DRIVE_DELAY;
   parameter real vthresh = `LF_MID_THRESH;

   parameter real fract = 1.0;
   parameter real scale = 1.0;

   real vsum = 0.0;
   real vovf = 0.0;

   analog begin
      @(cross(V(ref) - vthresh, +1.0)) begin
         vsum = vsum + fract;
         if (vovf > 0.0) begin
            vovf = 0.0;
         end
         if (vsum > 10.0) begin
            vsum = vsum - 10.0;
            vovf = 5.0;
         end
      end
```

```
        V(ovf) <+ transition(vovf, tdel, trise, tfall);
        V(sum) <+ transition(0.1*scale*vsum, tdel,
           trise, tfall);
    end

endmodule
```

5.6.4 Test Bench and Results

Listing 5.18 is the test bench designed for evaluating the system performance. The input reference clock is 4MHz. The loop multiplication factor is set to 5.4 (N=5, F=4) and thus the vco output frequency should be at 21.6MHz.

LISTING 5.18 Spice netlist of frequency synthesizer test bench.

```
* Fractional N-loop frequency synthesizer
.verilog "accu.va"
.verilog "cp.va"
.verilog "dvco.va"
.verilog "divbyn.va"
.verilog "fnfs.va"
.verilog "fpd.va"
.verilog "pulrem.va"
.verilog "sum.va"

vref ref 0 dc 0 pulse(0 5 10n 2n 2n 100n 250n)

xfnfs out ref 0 fnfs n=5 f=4

.model capacitor c
.model resistor r

.tran .02u 10. 0u

.end
```

The test bench setup defines two model definitions (for `resistor` and `capacitor`) which are simulator primitives instantiated from within the `fnfs` structural description.

Figure 5.27 shows the dynamic characteristics of the vco input signal. The phase-locked loop achieves lock after approximately five microseconds.

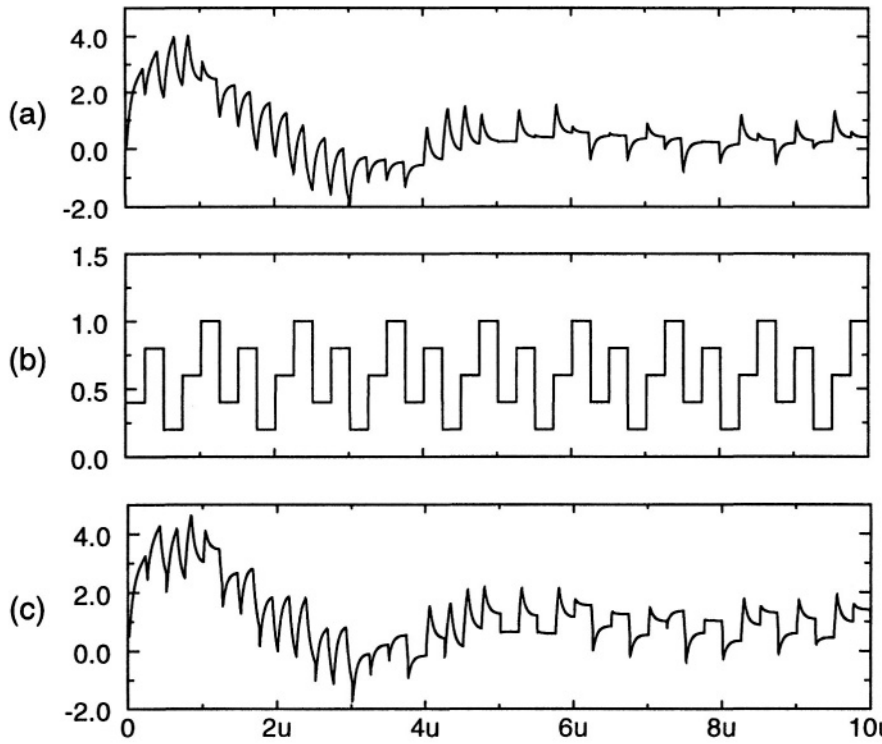


FIGURE 5.27 Time domain settling response of frequency synthesizer. (a) is output of the low-pass filter, (b) the phase-error correction signal, and (c) the input control signal to the voltage-controlled oscillator.

Figuire 5.28 shows the output signal after the `vco` acquires lock. Note, that for five clock cycles of the reference signal, the output goes through 27 cycles (5.4 * 5).

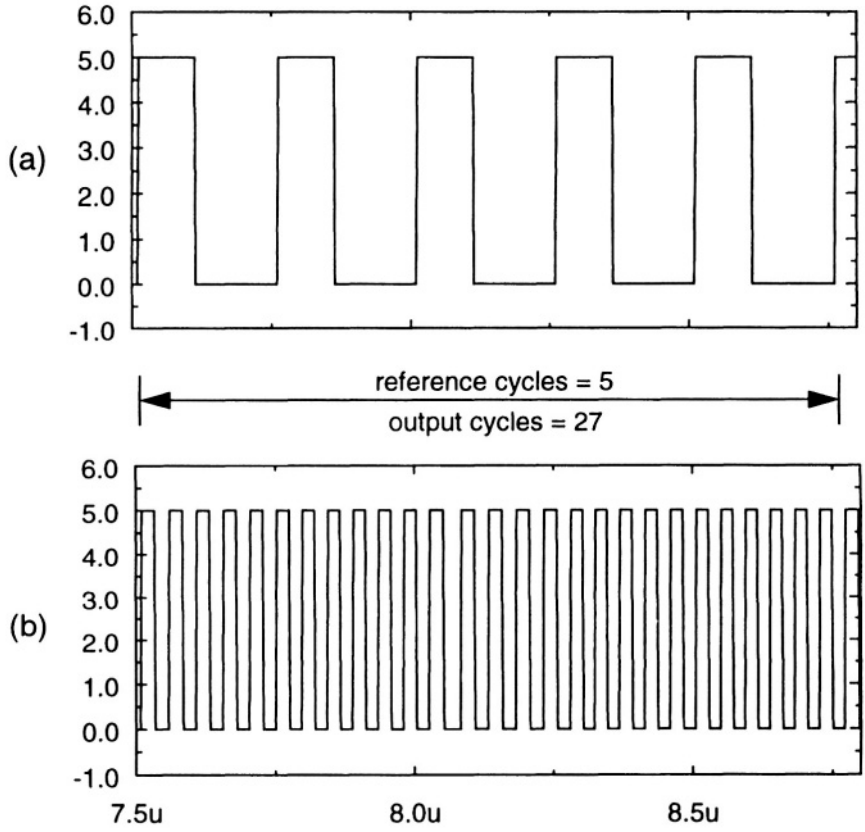


FIGURE 5.28 Measurement of the reference (a) and output (b) signals of the frequency synthesizer. Note the ratio of the cycles is exactly 5.4:1.

5.7 Antenna Position Control System

This example illustrates some of the multi-disciplinary modeling capabilities of the Verilog-A language. The antenna position control system, shown in Figure 5.29, consists of both electrical and mechanical (rotational) components.

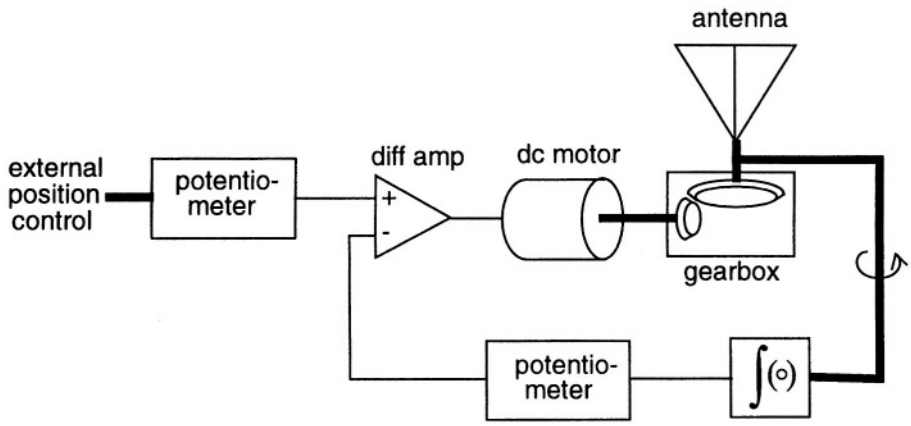


FIGURE 5.29 Schematic representation of the antenna position control system.

The position control system employs two potentiometers, one for converting the external position control into a voltage and another for sensing the current position of the antenna. The outputs of the potentiometers feed into a differential amplifier which drives the motor. The antenna is driven by the output of the motor via a gearbox.

The potentiometers are defined in terms of `electrical` and `rotational` disciplines. The `rotational` discipline relates an angle to a torque and is used to sense the position of the input shaft of the potentiometer. The motor, gearbox, and antenna mechanical components are defined in terms of the `rotational_omega` discipline which relates angular velocity to torque. Hence, we integrate the angular velocity of the antenna to determine its position.

5.7.1 Potentiometer

The potentiometer model converts a rotational position into a voltage. The module is parameterized in terms of the minimum and maximum values of the potentiometer control shaft. The corresponding output voltage scale is controlled by the value of the voltage across the input pins, `inPos` and `inNeg`.

LISTING 5.19 Verilog-A potentiometer definition.

```
module potentiometer(out, shaft, inPos, inNeg);
   output out;
   input shaft, inPos, inNeg;
   electrical out;
   rotational shaft;
   electrical inPos, inNeg;

   parameter real min_angle = -`M_PI;
   parameter real max_angle = `M_PI;

   real scale, shaft_angle;

   analog begin
      if (Theta(shaft) > max_theta)
         shaft_angle = max_theta;
      else if (Theta(shaft) < min_theta)
         shaft_angle = min_theta;
      else
         shaft_angle = Theta(shaft);

      scale = V(inPos, inNeg)/(max_cntrl - min_cntrl);
      V(out) <+ V(inNeg) + scale*ctrl_val;
   end

endmodule
```

5.7.2 DC Motor

The core of any DC motor is an electrical armature which converts between electrical and mechanical power without any loss. The electrical properties of the motor include its resistance, R_m and inductance, L_m. The mechanical properties are the motors iner-

tia, J_m and rotational friction, B_m. The back voltage generated by the motor is K_m times the angular frequency of the motor, θ_m, and the torque is K_t times the current through the motor, I_m. This is shown diagrammatically in Figure 5.30.

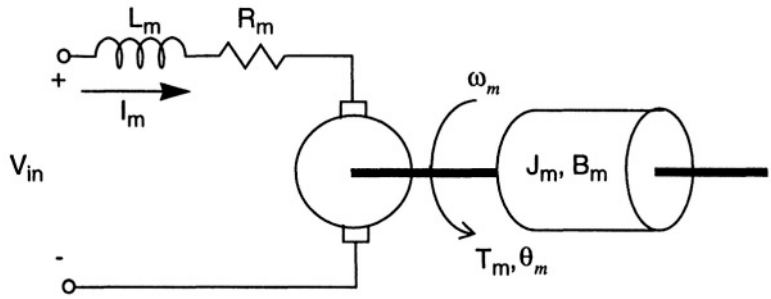


FIGURE 5.30 Schematic representation of the dc motor model.

The equations describing the terminal and output characteristics of the motor become:

$$T_m = K_t \cdot I_m - B_m \cdot \theta_m - \frac{d}{dt}(J_m \cdot \theta_m)$$

$$V_{in} = R_m \cdot I_m + \frac{d}{dt}(L_m \cdot I_m) + K_m \cdot \theta_m$$

Within the DC motor module, these equations representing the constitutive behavior of the component are:

```
Tau(shaft) <+ Kt*I(in) - Bm*Omega(shaft) -
   ddt(Jm*Omega(shaft));
V(in) <+ Rm*I(in) + ddt(Lm*I(in)) + Km*Omega(shaft);
```

5.7.3 Gearbox

The motor translates torque to the antenna via a gearbox. In addition to the translational affects of the gear ratios between the two shafts, the model for the gearbox must be bidirectional in that the torque from the motor must affect the antenna, and the inertial load of the antenna must be expressed on the load of the motor.

If we assume that the gears do not slip, then equating translational distance for the two gears in terms of their angular position yields:

$$r_1 \cdot \theta_1 = r_2 \cdot \theta_2$$

The relationship between the torque on the two shafts is related by the force at the point of contact, $F_1 = F_2$, where $\tau = r \cdot F$. The total torque on the shaft is the externally applied torque less the inertia of the gear.

LISTING 5.20 Verilog-A gearbox model.

```
module gearbox(shaft1, shaft2);
    inout shaft1, shaft2;
    rotational_omega shaft1, shaft2;

    parameter real r1 = 1 from (0:inf);
    parameter real i1 = 1m from [0:inf);
    parameter real r2 = 1 from (0:inf);
    parameter real i2 = 1m from [0:inf);

    analog begin
        Omega(shaft1) <+ Omega(shaft2)*(r2/r1);
        Tau(shaft2) <+ i2*ddt(Omega(shaft2)) +
            (Tau(shaft1) - i1*ddt(Omega(shaft1)))*r2/r1;
    end

endmodule
```

5.7.4 Antenna

The antenna represents a rotational load on the shaft of the gearbox which is characterized in terms of the inertia.

LISTING 5.21 Verilog-A antenna model.

```
module antenna(shaft);
    inout shaft;
    rotational_omega shaft;

    parameter real i = 1;
```

```
    analog begin
       Tau(shaft) <+ i*ddt(Omega(shaft));
    end

endmodule
```

5.7.5 Test Bench and Results

The modules are assembled in the Listing 5.22 as per the schematic of Figure 5.31.

LISTING 5.22 Spice netlist of antenna position controller test bench

```
* position controller

.verilog "servo.va"

vsupply supply 0 5.0
vctrl inpos 0 pwl(0 0 1 0 2 -1.0472
+ 10 -1.0472 11 0.7854 20 0.7854)

xinpot supply 0 inpos diffplus potentiometer
+ min_ctrl=-1.5708 max_ctrl=1.5708

xdiffamp inmotor diffplus diffminus diff_amp k = 24

xmotor inmotor outangle motor_dc

xgearbox outangle gearangle gearbox
+ r2=10 i1=0 i2=0

xantenna gearangle antenna inertia=1

xintgr8 igearpos gearangle intgr8 pos_ic = 0

xoutpot supply 0 igearpos diffminus potentiometer
+ min_ctrl = -1.5708 max_ctrl = 1.5708

.tran 0.01 20

.end
```

In Figure 5.31, we show the applied position to the control system and the response for both light and heavy antennas. The position input to the system is in radians.

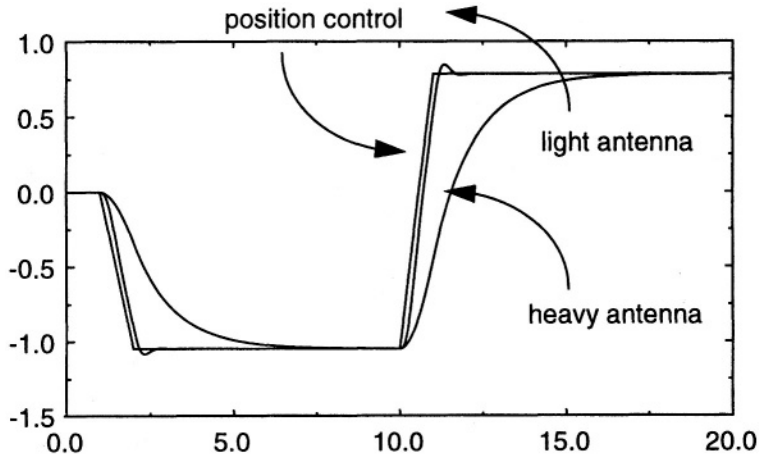


FIGURE 5.31 Time-domain trajectory of antenna position in response to a position control signal for both light and heavy antennas.

The applied voltage to the motor is shown in Figure 5.32

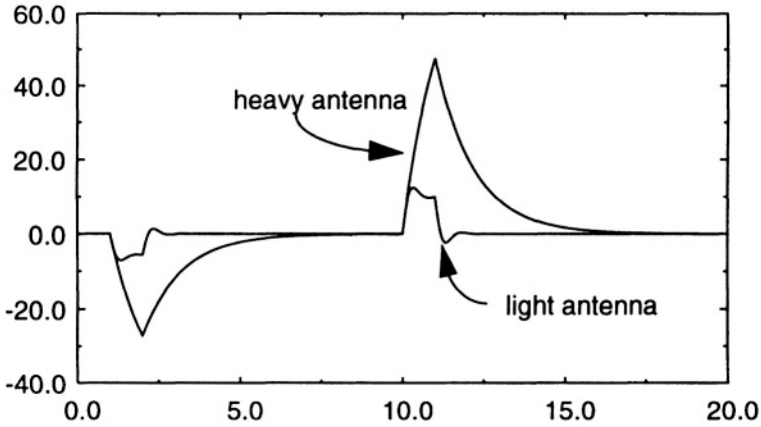


FIGURE 5.32 Applied motor drive voltage in response to position control signal for both light and heavy antennas.

Appendix A

Lexical Conventions and Compiler Directives

A.1 Verilog-A Language Tokens

Verilog-A source files are a stream of lexical tokens. A lexical token consists of one or more characters. The layout of tokens in a source file is free format - spaces and newlines are not syntactically significant other than being token separators, except escaped identifiers.

The types of lexical tokens in the language are as follows:

- white space
- comment
- operators
- number
- string
- identifier
- keyword

A.1.1 White Space

White space contains the characters for spaces, tabs, newlines, and form feeds. These characters are ignored except when they serve to separate other lexical tokens.

A.1.2 Comments

The Verilog-A language has two forms to introduce comments. A one-line comments starts with two characters // and ends with a newline. A block comment starts with a /* and ends with a */. Block comments can not be nested. The one-line comment token // does not have any meaning within a block comment.

A.1.3 Operators

Operators are single, double, or triple character sequences and are used in expressions. Unary operators appear to the left of their operand. Binary operators appear between their operands. A conditional operator has two operator characters that separate three operands. The following table lists the Verilog-A operators and their descriptions.

+	plus
-	minus
*	times
/	divide
%	modulus
||	logical or
&&	logical and
|	bit or
&	bit and
^	bit xor
~^	bit xnor
~	neg
<<	left shift
>>	right shift
<	less than
>	greater than
<=	less equal
>=	greater equal
!=	not equal

==	equal
=	assignment
<+	contrib
?:	ternary

A.1.4 Numbers

Constant numbers can be specified as integer or real constants.

Integer constants are specified in decimal format as a sequence of digits 0 through 9, optionally starting with a plus or minus unary operator. The underscore character (_) is legal anywhere in a decimal number except as the first character. The underscore character is ignored. This feature can be used to break up long numbers for readability purposes.

Real constant numbers are represented as described by IEEE STD-754-1985, an IEEE standard for double precision floating point numbers.

Real numbers can be specified in either decimal notation (for example, 14.82) or in scientific notation (for example 1.6e8, which indicates 1.6 multiplied by 10 raised to the 8th power). Real numbers expressed with a decimal point must have at least one digit on each side of the decimal point.

Real numbers can be specified with the following letter symbols for the scale factors indicated:

T	10e12
G	10e9
Meg	10e6
K	10e3
m	10e-3
u	10e-6
n	10e-9
p	10e-12
f	10e-15
a	10e-18

No space is permitted between the number and the symbol. This form of floating-point number specification is provided in the Verilog-A language in addition to the other methods for writing floating point numbers.

A.1.5 Conversion

Real numbers are converted to integers by rounding the real number to the nearest integer, rather than truncating it. Implicit conversion takes place when a real number is assigned to an integer. The ties are rounded away from zero.

A.1.6 Identifiers, Keywords and System Names

An identifier is used to give an object an unique name so that it can be referenced. An identifier can be any sequence of letters, digits, and the underscore characters (_).

The first character of an identifier can not be a digit; it can be a letter. Identifiers are case sensitive.

A.1.7 Escaped Identifiers

Escaped identifiers start with the backslash character (\) and end with white space (space, tab, newline). They provide a means of including any of the printable ASCII characters in an identifier (the decimal values 33 through 126 or hexadecimal values 21 through 7E).

Neither the leading back-slash character nor the terminating white space is considered part of the identifier.

A.1.8 Keywords

Keywords are predefined non-escaped identifiers that are used to define language constructs. All keywords are defined in lowercase only.

A.1.9 Verilog-A Keywords

abstol	**access**	**analog**	**begin**
case	**ddt**	**ddt_nature**	**default**
discipline	**else**	**end**	**endcase**
enddiscipline	**endmodule**	**endnature**	**exclude**
flow	**for**	**from**	**idt**
idt_nature	**if**	**inf**	**initial**
inout	**input**	**integer**	**module**
nature	**output**	**or**	**parameter**
potential	**real**	**repeat**	**units**
while			

A.1.10 Math Function Keywords

abs	**acos**	**acosh**	**asin**
asinh	**atan**	**atanh**	**cos**
cosh	**exp**	**ln**	**log**
max	**min**	**pow**	**sin**
sinh	**sqrt**	**tan**	**tanh**

The following table contains the standard mathematical functions supported by the Verilog-A language and their regions of validity. The operands must be numeric (integer or real). For `min`, `max`, and `abs` , if either operand is real, both are converted to real as is the result. Arguments to all other functions are converted to real.

Function	Description	Domain
ln(x)	Natural logarithm	x > 0
log(x)	Decimal logarithm	x > 0
exp(x)	Exponential	x < 80
sqrt(x)	Square root	x > 0
min(x, y)	Minimum	all x, all y
max(x, y)	Maximum	all x, all y
abs(x)	Absolute	all x

Function	Description	Domain
pow(x, y)	Power. x^y	all x, all y

The following table defines the trigonometric and hyperbolic functions supported by the Verilog-A language. All operands must be of the numeric type (integer and real) and are converted to real if necessary. All arguments to the trigonometric and hyperbolic functions are specified in radians.

Function	Description	Domain
sin(x)	Sine	all x
cos(x)	Cosine	all x
tan(x)	Tangent	$x \neq n\left(\frac{\pi}{2}\right)$, n is odd
asin(x)	Arc-sine	-1 <= x <= 1
acos(x)	Arc-cosine	-1 <= x <= 1
atan(x)	Arc-tangent	all x
sinh(x)	Hyperbolic sine	all x
cosh(x)	Hyperbolic cosine	all x
tanh(x)	Hyperbolic tangent	all x
asinh(x)	Arc-hyperbolic sine	all x
acosh(x)	Arc-hyperbolic cosine	x >= 1
atanh(x)	Arc-hyperbolic tangent	-1 <= x <= 1

As with any mathematical description language, the Verilog-A language requires that the model developer (and user of the model) understand the usage conditions. Almost all the mathematical and trigonometric functions, by their definition, have some restrictions on their inputs. Consideration of these properties require that the modeler understand the types and ranges of signals that the model will be used under and develop accordingly.

A.1.11 Analog Operator Keywords

analysis	**bound_step**	**cross**	**delay**
laplace_nd	**laplace_np**	**laplace_zd**	**laplace_zp**
slew	**timer**	**transition**	**zi_nd**
zi_np	**zi_zd**	**zi_zp**	

Analog operators are described in Chapter 3.

A.1.12 System Tasks and Functions

The ($) character introduces a language construct that enables development of user-defined tasks and functions. A name following the ($) is interpreted as a system task or a system function.

See Appendix B. for Verilog-A system tasks and their descriptions

A.2 Compiler Directives

All Verilog-A language compiler directives are preceded by the (`) character. This character is called accent grave. It is different from the character ('), which is the single quote character. The scope of compiler directives extends from the point where it is processed, across all files processed, to the point where another compiler directive supersedes it or the processing completes.

The following compiler directives are available:

`define

`else

`endif

`ifdef

`include

`resetall

`undef

A.2.1 ‘define and ‘undef

The directive **`define** creates a macro for text substitution. This directive can be used both inside and outside module definitions. After a text macro is defined, it can be used in the source description by using the (`) character, followed by the macro name. The compiler substitutes the text of the macro for the string `macro_name.

All compiler directives are considered predefined macro names; it is illegal to redefine a compiler directive as a macro name. A text macro can be defined with arguments. This allows the macro to be customized for each individual use. An example of the definition and use is illustrated below:

```
`define threshold 0.5
`define pp_max(a,b)  ((a > b)  ? a  :  b)

@(cross(V(thr)  -`threshold,  0.0))

V(out)  <+ `pp_max(V(in1), V(in2));
```

The macro text can be any arbitrary text specified on the same line as the text macro name. If more than one line is necessary to specify the text, the newline must be preceded by a backslash (\). The first newline not preceded by a backslash will end the macro text The newline preceded by a backslash is replaced in the expanded macro with a newline (but without the preceding backslash character).

For an argument-less macro, the text is substituted “as-is” for every occurrence of the `text_macro. However, a text macro with one or more arguments must be expanded by substituting each formal argument with the expression used as the actual argument in the macro usage.

The directive **`undef** undefines a previously defined text macro. An attempt to undefine a text macro that was not previously defined using a **`define** compiler directive can result in a warning.

An undefined text macro has no value.

A.2.2 `ifdef, `else, `endif

These conditional compilation compiler directives are used to optionally include lines of a Verilog-A language source description during compilation. The **`ifdef** compiler directive checks for the definition of a variable name. If the variable name is defined, then the lines following the **`ifdef** directive are included. If the variable name is not defined and a **`else** directive exists then this source is compiled.

These directives may appear anywhere in the Verilog-A source description.

The **`ifdef, `else, `endif** compiler directives work in the following manner:

- When an **`ifdef** is encountered, the text macro name is test to see if it is defined as a text macro name using **`define** within the Verilog-A language source description
- If the text macro name is defined, the first group of lines is compiled as a part of the description. If there is an **`else** compiler directive, the second group of lines is ignored.
- If the text macro name has not been defined, the first group of lines is ignored. If there is an **`else** compiler directive the second group of lines is compiled.

Any group of lines that the compiler ignore still must follow the Verilog-A language lexical conventions for white space, comments, numbers, strings, identifiers, keywords, and operators. The following are some examples of using the **`ifdef** and **`endif** compiler directives:

```
`define debug

`ifdef debug
      $strobe("module %m: input signal = %g at time %g",
         V(in), $realtime());
`endif
```

These compiler directives may be nested.

A.2.3 ‘include

The **`include** compiler directive is used to insert the entire contents of a source file in another file during compilation. The result is as thought the contents of the included source file appear in place of the **`include** compiler directive. The **`include** compiler directive can be used to include global or commonly used definitions and tasks without encapsulating repeated code within module boundaries. For example:

```
`include "std.va"
```

Includes the standard definitions for **discipline** and **nature** definitions into the scope of the current file. The ‘include mechanism permits configuration management and organization of Verilog-A source files.

The compiler directive **`include** can be specified anywhere within the Verilog-A language description. The filename is the name of the file to be included in the source file. The filename can be a full or relative path name.

Only white space or a comment may appear on the same line as the **`include** compiler directive.

A file included in the source using the **`include** compiler directive may contain other **`include** compiler directives. The number of nesting levels for included may be limited, but the limit shall be at least 15.

A.2.4 ‘resetall

When the **`reset_all** compiler directive is encountered during compilation, all compiler directives are set to the default values. This is useful for insuring that only those directives that are desired in compiling a particular source file are active.

The recommend usage is to place **`resetall** at the beginning of each source text file, followed immediately by the directives desired in the file.

Appendix B

System Tasks and Functions

B.1 Introduction

The Verilog-A language supports a variety of system tasks and functions. These tasks and functions are useful for querying and controlling the current simulation as well as displaying the results of the simulation as it progresses.

B.2 Strobe Task

The **$strobe** task is the main task for printing information during a simulation. `$strobe` always prints a newline character at the end of its execution. The typical form of the parameters to `$`**strobe** is:

```
$strobe("format specification", parameters)
```

The format control specification string defines how the following arguments in the **$strobe** task are to be printed. The syntax is a percent character (%) followed by a

format specifier letter. Allowable letters in the format control specification are shown in the table below:

letter	display format	comments
h or H	hexadecimal	
d or D	decimal	
o or O	octal	
b or B	binary	
c or C	ASCII character	
s or S	string	
m or M	hierarchical name	takes no arguments

Other special characters may be used with escape sequences.

escape sequence	display
\n	new line character
\t	tab character
\\	the \ character
\"	the " character
\ddd	character specified in up to three octal digits

B.2.1 Examples

The following are examples on the use of the `$strobe` task:

```
$strobe("input = %g", V(in));
$strobe("result = %b", ~flag & bits);
$strobe("%m: event triggered at t = %g", $realtime());
```

B.3 File Output

The **`$strobe`** task has a version for writing to files, **`$fstrobe`**. **`$fstrobe`** requires an extra parameter, called the file descriptor, as shown below:

```
$fstrobe(descriptor,  "format specification",
   parameters)
```

The descriptor is an integer value returned from the **$fopen** function. **$fopen** takes the form:

```
integer  fdescriptor;

fdescriptor = $fopen("file_name");
```

$fopen will return a 0 if it was unable to open the file for writing. When finished, the file can be closed with the **$fclose** function call:

```
$fclose(fdescriptor) ;
```

The file descriptors are set up so that each bit of the descriptor indicates a different channel. Thus, multiple calls to **$fopen** will return a different bit set. The least significant bit indicates the "standard output". By passing the bit-wise or of two or more file descriptors to **$fstrobe**, the same message will be printed into all of the files (as well as standard output) indicated by the or-ed file descriptors.

B.4 Simulation Time

$realtime is a system task that returns the current simulation time as a real number.

```
$realtime()
```

B.5 Probabilistic Distribution

The probabilistic distribution functions return pseudo-random numbers whose characteristics are described by the task name.

```
$dist_uniform(seed,  start,  end)
$dist_normal(seed,  mean,  standard_deviation)
$dist_exponential(seed,  mean)
$dist_poisson(seed,  mean)
```

```
$dist_chi_square (seed, degree_of_freedom)
```

All of the parameter are integer values and all return real values. For each system task, the `seed` parameter must also be an integer variable that is initialized by the user and only updated by the system task.

The **$dist_uniform** returns random numbers uniformly distributed in the interval specified by its parameters.

For the **$dist_normal** and **$dist_chi_square** functions, the `standard_deviation` and `degree_of_freedom` parameter respectively are used to determine the shape of the density functions. With a mean of 0 and `standard_deviation` of 1, **$dist_normal** generates a gaussian distribution. For **$dist_chi_square,** larger numbers will spread the returned values over a wider range.

For **$dist_exponential** and **$dist_poisson,** the mean parameter is an integer which causes the average value returned by the function to approach the value specified.

B.6 Random

The **$random** system function provides a random number mechanism, returning a new 32-bit random number each time the function is called. The returned value is a signed integer; it can be positive or negative. The function may be called with or without a **seed** parameter.

```
$random
$random(seed)
```

The `seed` parameter is used to initialize the stream of numbers that **$random** returns. The `seed` parameter must be a integer variable and assigned a value before calling **$random.**

B. 7 Simulation Environment

These functions return information about the current simulation environment parameters. All return a real number.

Function	Returned value	Description
`$temperature()`	real	Ambient temperature
`$vt()`	real	Thermal voltage at the ambient temperature.
`$vt(temp)`	real	Thermal voltage at temperature of `temp`.

Appendix C

Laplace and Discrete Filters

C.1 Introduction

The laplace and discrete transform analog operators in the Verilog-A language allow for specification of the filter coefficients in four different combinations of either polynomial or zeros/poles for the numerator and denominator of the filter.

This appendix describes these forms in detail, as well as presenting some MATLAB scripts for generating the required coefficients from filter specifications. (MATLAB is a registered trademark of the MathWorks, Inc., 24 Prime Park Way, Natick, MA 01760-1500: telephone (508) 653-1415, Fax (508) 653-2997, e-mail: info@mathworks.com). These scripts may require the Simulink Toolkit.

C.2 Laplace Filters

C.2.1 laplace_zp

`laplace_zp` implements the zero-pole form of the Laplace transform filter.

`laplace_zp`(expr, ζ, ρ)

where ζ is a vector of M paris of real numbers. Each pair represents a zero, the first number in the pair is the real part of the zero, and the second is the imaginary part. Similarly, ρ is the vector of N real pairs, one for each pole. The poles are given in the same manner as the zeros. The transfer function is:

$$H(s) = \frac{\prod_{k=0}^{M-1}\left(1 - \frac{s}{\zeta_k^{\,r} + \zeta_k^{\,i}}\right)}{\prod_{k=0}^{N-1}\left(1 - \frac{s}{\rho_k^{\,r} + \rho_k^{\,i}}\right)}$$

where $\zeta_k^{\,r}$ and $\zeta_k^{\,i}$ are the real and imaginary parts of the k^{th} zero, while $\rho_k^{\,r}$ and $\rho_k^{\,i}$ are the real and imaginary parts of the k^{th} pole. If a root (a pole or a zero) is real, the imaginary part must be specified as 0. If a root is complex, its conjugate must also be present. If a root is zero, then the term associated with it is implemented as s rather than $(1 - s/r)$ where r is the root.

C.2.2 laplace_zd

laplace_zd implements the zero-denominator form of the Laplace transform filter.

laplace_zd(expr, ζ, d)

where ζ is a vector of M paris of real numbers. Each pair represents a zero, the first number in the pair is the real part of the zero, and the second is the imaginary part. Similarly, d is the vector of N real coefficients of the denominator. The transfer function is:

$$H(s) = \frac{\prod_{k=0}^{M-1}\left(1 - \frac{s}{\zeta_k^{\,r} + \zeta_k^{\,i}}\right)}{\sum_{k=0}^{N-1} d_k s^k}$$

where $\zeta_k^{\,r}$ and $\zeta_k^{\,i}$ are the real and imaginary parts of the k^{th} zero, while d^k is the k^{th} power of s in the denominator. If a zero is real, the imaginary part must be spec-

ified as 0. If a zero is complex, its conjugate must also be present. If a zero is zero, then the term associated with it is implemented as s rather than $(1 - s/\zeta)$.

C.2.3 laplace_np

laplace_np implements the numerator-pole form of the Laplace transform filter.

laplace_np(expr, ζ, ρ)

where ζ is a vector of M real numbers that contains the coefficients of the numerator. For the denominator, ρ is the vector of N real pairs, one for each pole where the first number in the pair is the real part of the zero, and the second is the imaginary part. The transfer function is:

$$H(s) = \frac{\sum_{k=0}^{M-1} d_k s^k}{\prod_{k=0}^{N-1} \left(1 - \frac{s}{\rho_k^r + \rho_k^i}\right)}$$

where n_k is the coefficient of the k^{th} power of s in the numerator, while ρ_k^r and ρ_k^i are the real and imaginary parts of the k^{th} pole. If the pole is real, the imaginary part must be specified as 0. If a pole is complex, its conjugate must also be present. If a pole is zero, then the term associated with it is implemented as s rather than $(1 - s/\rho)$.

C.2.4 laplace_nd

laplace_nd implements the numerator-denominator form of the Laplace transform filter.

laplaca_nd(expr, *n, d*)

where n is a vector of M real numbers that contains the coefficients of the numerator, and d is a vector of N real numbers that contains the coefficients of the denominator. The transfer function is:

$$H(s) = \frac{\sum_{k=0}^{M} n_k s^k}{\sum_{k=0}^{N} d_k s^k}$$

where n_k is the coefficient of the k^{th} power of s in the numerator, and d_k is the coefficient of the k^{th} power of s in the denominator.

C.3 Discrete Filters

C.3.1 zi_zp

zi_zp implements the zero-pole form of the Z transform filter.

zi_zp(expr, ζ, ρ, *T*)

where ζ is a vector of M paris of real numbers. Each pair represents a zero, the first number in the pair is the real part of the zero, and the second is the imaginary part. Similarly, ρ is the vector of N real pairs, one for each pole. The poles are given in the same manner as the zeros. The transfer function is:

$$H(z) = \frac{\prod_{k=0}^{M-1}\left(1 - \frac{z}{\zeta_k^{\,r} + \zeta_k^{\,i}}\right)}{\prod_{k=0}^{N-1}\left(1 - \frac{z}{\rho_k^{\,r} + \rho_k^{\,i}}\right)}$$

where $\zeta_k^{\,r}$ and $\zeta_k^{\,i}$ are the real and imaginary parts of the k^{th} zero, while $\rho_k^{\,r}$ and $\rho_k^{\,i}$ are the real and imaginary parts of the k^{th} pole. If a root (a pole or a zero) is real, the imaginary part must be specified as 0. If a root is complex, its conjugate must also be present. If a root is zero, then the term associated with it is implemented as z rather than $(1 - z/r)$ where r is the root

C.3.2 zi_zd

zi_zd implements the zero-denominator form of the Z transform filter.

zi_zd(expr, ζ, *d*, *T*)

where ζ is a vector of M paris of real numbers. Each pair represents a zero, the first number in the pair is the real part of the zero, and the second is the imaginary part. Similarly, *d* is the vector of N real coefficients of the denominator. The transfer function is:

$$H(z) = \frac{\prod_{k=0}^{M-1}\left(1 - \frac{z}{\zeta_k^{\,r} + \zeta_k^{\,i}}\right)}{\sum_{k=0}^{N-1} d_k z^k}$$

where $\zeta_k^{\,r}$ and $\zeta_k^{\,i}$ are the real and imaginary parts of the k^{th} zero, while d^k is the k^{th} power of *s* in the denominator. If a zero is real, the imaginary part must be specified as 0. If a zero is complex, its conjugate must also be present. If a zero is zero, then the term associated with it is implemented as z rather than $(1 - z/\zeta)$.

C.3.3 zi_np

zi_np implements the numerator-pole form of the Z transform filter.

zi_np(expr, *n*, ρ, *T*)

where *n* is a vector of M real numbers that contains the coefficients of the numerator. For the denominator, ρ is the vector of N real pairs, one for each pole where the first number in the pair is the real part of the zero, and the second is the imaginary part. The transfer function is:

where n_k is the coefficient of the k^{th} power of *s* in the numerator, while $\rho_k^{\,r}$ and $\rho_k^{\,i}$ are the real and imaginary parts of the k^{th} pole. If the pole is real, the imaginary part must be specified as 0. If a pole is complex, its conjugate must also be present. If

$$H(z) = \frac{\sum_{k=0}^{M-1} d_k z^k}{\prod_{k=0}^{N-1}\left(1 - \frac{z}{\rho_k^{\,r} + \rho_k^{\,i}}\right)}$$

a pole is zero, then the term associated with it is implemented as z rather than $(1 - z/\rho)$

C.3.4 zi_nd

zi_nd implements the numerator-denominator form of the Z transform filter.

zi_nd(expr, *n, d, T*)

where *n* is a vector of M real numbers that contains the coefficients of the numerator, and *d* is a vector of N real numbers that contains the coefficients of the denominator. The transfer function is:

$$H(z) = \frac{\sum_{k=0}^{M} n_k z^k}{\sum_{k=0}^{N} d_k z^k}$$

where n_k is the coefficient of the k^{th} power of z in the numerator, and d_k is the coefficient of the k^{th} power of z in the denominator.

C.4 Verilog-A MATLAB Filter Specification Scripts

The following MATLAB scripts for generating Verilog-A laplace and discrete filter coefficients from the filter characteristics. Scripts are provided for Butterworth, Chebyshev Type I and II, and Elliptic filters for both continuous and discrete time filters. The Verilog-A MATLAB filter scripts are found in the MATLAB subdirectory on the Verilog-A Explorer distribution diskette.

For laplace filters, there are four nnMATLAB scripts:

```
laplace_butter(fname, fpass, fstop, apass, astop);
laplace_cheby1(fname, fpass, fstop, apass, astop)
laplace_cheby2(fname, fpass, fstop, apass, astop)
laplace_ellip(fname, fpass, fstop, apass, astop)
```

Discrete filters in Verilog-A are specified:

```
zi_butter(fname, samp_freq, fpass, fstop, apass, astop,
   tau, t0)
zi_cheby1(fname, samp_freq, fpass, fstop, apass, astop,
   tau, t0)
zi_cheby2(fname, samp_freq, fpass, fstop, apass, astop,
   tau, t0)
zi_ellip(fname, samp_freq, fpass, fstop, apass, astop,
   tau, t0)
```

Both laplace and discrete filters accept the following arguments:

`filter_name` - name of the Verilog-A filter module.

`fpass` - passband corner frequency or cutoff frequency.

`fstop` - stopband corner frequency.

`apass` - passband attenuation in dBs. Maximum passband ripple or loss.

`astop` - stopband attenuation in dBs. Amount the stopband is down from the passband.

In addition, the discrete filters take the following arguments:

`samp_freq` - sampling frequency of the filter.

`tau` - filter output transition time.

`t0` - initial delay to the filter output.

The type of the filter is specified by the relationship of the passband to corner frequencies. For lowpass filters, `fpass` is less than `fstop`:

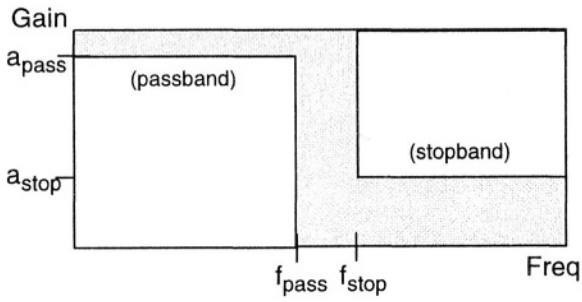


FIGURE C.1 Lowpass filter specifications.

A lowpass example:

```
laplace_cheby1('filter_1p', 18000, 22000, 10, 60);
```

For highpass filters, `fpass` is greater than `fstop`

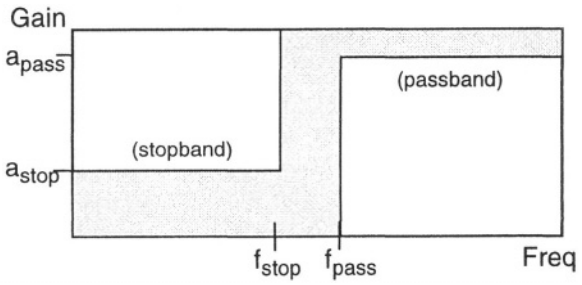


FIGURE C.2 Highpass filter specifications.

A highpass example:

```
laplace_cheby1('filter_hp', 22000, 18000, 10, 60);
```

For bandpass and bandstop filters, `fpass` and `fstop` are two-element vectors that specify the corner frequencies at both edges of the filter, lower frequency edge first.

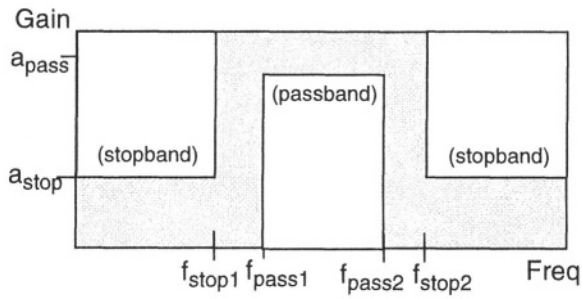


FIGURE C.3 Bandpass filter specifications.

A bandpass example:

```
laplace_cheby1('filter_bp', [17750 18250],
   [17500 18500], 3, 90);
```

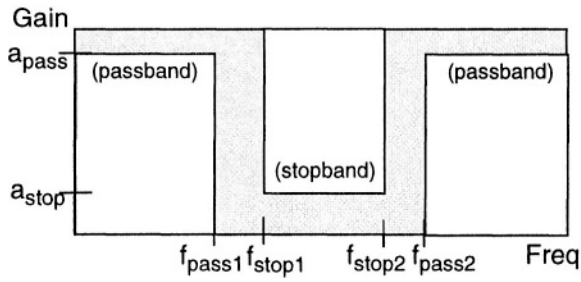


FIGURE C.4 Bandstop filter specifications.

For bandstop:

```
laplace_cheby1('filter_bs', [17500 18500],
    [17750 18250], 3, 90);
```

Appendix D

Verilog-A Explorer IDE

D.1 Introduction

The Verilog-A Explorer IDE (Integrated Development Environment) is a Windows '95/NT application targeted for designers and modelers wishing to learn analog behavioral modeling with the Verilog-A language. The application includes:

- Graphical user interface for language-based design entry and analysis
- Spice-SL, a Spice3f5-based[1] simulator with the Verilog-A language compiler integrated
- Example circuits and sample Verilog-A modules

The Verilog-A Explorer user interface (Figure D.1) consists of thee major sections:

- Project Navigator which allows interactive navigation of the design via the simulation output results database
- Workspace for editing of both circuit (`* .ckt`) and Verilog-A (`* .va`) files

1. The simulator is a limited capability demonstration version in terms of capacity, components, and analysis types supported.

- Simulation Output Window for viewing the output results of the simulation.

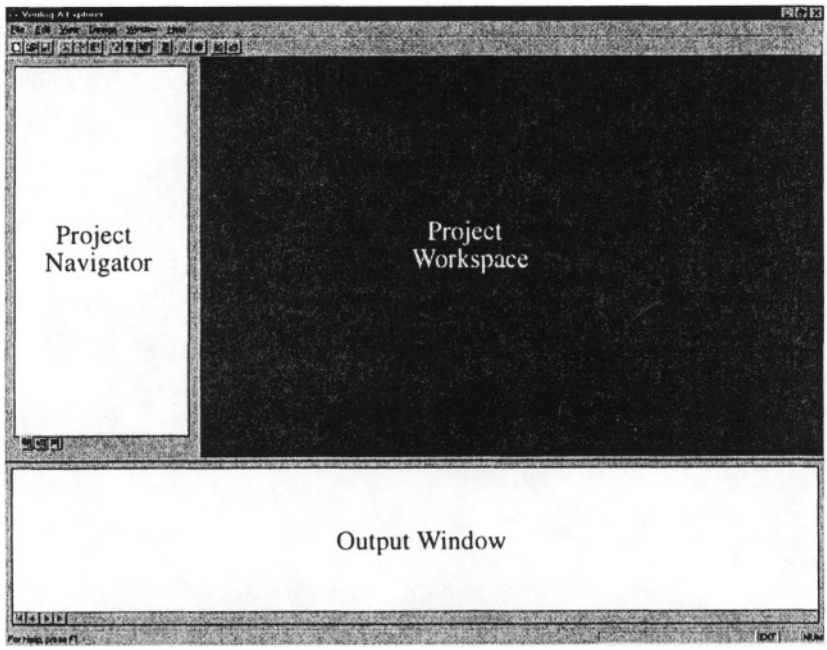


FIGURE D.1 Screenshot of the Verilog-A Explorer IDE including the three major components.

All evaluation and analysis of Verilog-A designs within the Verilog-A Explorer environment is centered around standard Spice circuit files. The circuit files include features of:

- Standard Spice syntax for design description and simulation control.
- **`.verilog`** extension for incorporating Verilog-A module descriptions within the circuit or test-bench description.
- Instantiation of Verilog-A modules via standard subcircuit instantiations (standard Spice Xxxxx device card).

D.2 Installation and Setup

Running `a:\setup` from the distribution media results in the following setup dialog. Choose the installation directory (`<install_dir>`) if you would like to change the path from the default (`c:\veriloga`)

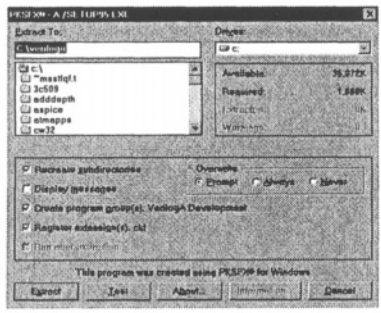


FIGURE D.2 Installation dialog for the Verilog-A Explorer IDE.

D.2.1 Overview of the Distribution

After successful installation, under `<install_dir>` you will find the following directory structure and files:

- File `license.txt`.
- Directory `bin` executables for the IDE and simulator.
- Directory `book` contains selected examples from the book.
- Directory `examples` contains miscellaneous examples.
- Directory `include` contains the Verilog-A standard definitions for disciplines and physical constants.
- Directory `lib` is organized in subdirectories for behavioral models of analog, communications, data acquisition, and digital. Circuit test bench files are also included.
- Directory `matlab` includes the MATLAB scripts referenced in Appendix C.
- Directory `template` has the template files for new `*.ckt` and `*.va` files created from the Explorer IDE.
- Directory `tutorial` contains the behavioral models and circuit used for illustration in this Appendix.

D.2.2 Executable and Include Path Setup

From the program group that is created, you can check the installation by starting the Verilog-A Explorer IDE. From the *Design->Settings* menu (from the main menu bar), raise the *Settings* dialog:

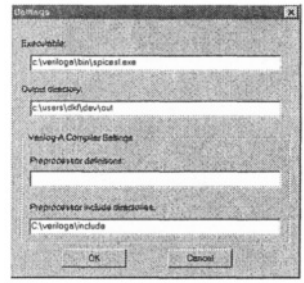


FIGURE D.3 *Settings* dialog for defining Spice-SL executable, output directory, and include paths.

The *Settings* dialog includes information regarding paths to the Verilog-A Explorer executables and include directories. These properties are defined as follows:

- Executable: Path to the Spice simulator (should be: `<install_dir>\bin\spicesl.exe`)
- Output Directory: Path to top-level directory where the results directory will be created. The default output directory is the path of the input circuit file. Change this to point to an area where you would like all the simulation results to be stored. Maintaining a common output directory can help with the organization (and deletion) of unneeded results directories.
- Verilog-A Preprocessor Definitions: A comma-separated list of Verilog-A preprocessor directives to be passed to the Verilog-A language compiler. An identifier, var, on this line is passed to the Verilog-A compiler with the same effect as **`` `define``** var in the Verilog-A source.
- Verilog-A Preprocessor Include Directory: Path to a comma-separated list of standard (for "`std.va`" and "`const.va`" definition files) and user-defined include directories (should at least have `<install_dir>\include` for the standard include files). If you maintain your own Verilog-A module libraries in a separate directory, add a path to that directory here.

D.2.3 Overview of the IDE Organization

The organization of the Verilog-A Explorer IDE is centered around the input circuit file (`*.ckt`). The files containing the Verilog-A module definitions (`*.va` for the environment) are referenced from the circuit file with the **`.verilog`** statement. For

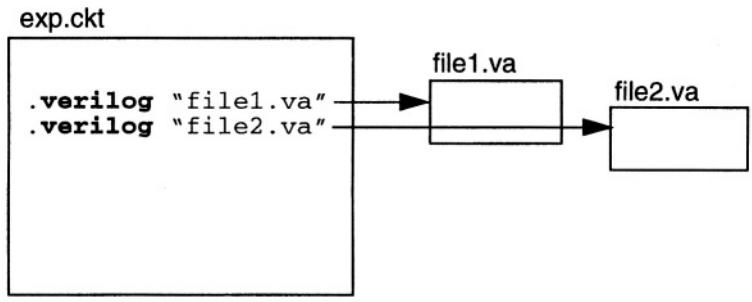


FIGURE D.4 Circuit file, `exp.ckt`, referencing two Verilog-A module files, `file1.va` and `file2.va`.

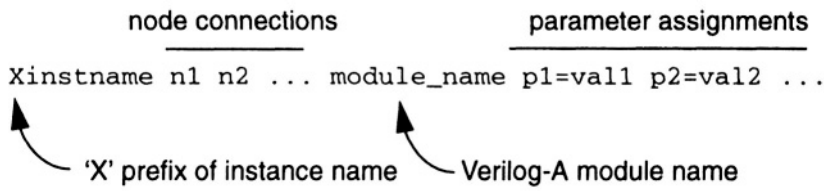
example, the circuit file `exp.ckt` of Figure D.4 references two Verilog-A files, `file1.va` and `file2.va`. When the Spice-SL simulator processes the input circuit file, it will pass these files off to the Verilog-A compiler.

A Verilog-A file can contain one or more definitions of Verilog-A modules. In essentially all cases, module files include at least the standard discipline definitions file, "`std.va`", as well as a file of pre-defined physical constants, "`const.va`", as shown in Listing D.1.

LISTING D.1 Inclusion of standard discipline and constants definitions

```
`include "std.va"
`include "const.va"

...
```

Within the circuit file, instantiations of Verilog-A modules is done via an extension of the Spice subcircuit instantiation mechanism. The subcircuit is instantiated as a 'X'



device, as part of the instance name. Followed are the circuit nodes attached to the module in the order as defined in the modules' port list. The Verilog-A module name identifies the type of the module, followed by an optional list of parameter-value pairs.

Simulation of a circuit file creates a results directory that stores the output results for all the analyses specified within the circuit file. The name of the results directory created is the same as the circuit file but with a **`.res`** extension. When you are asked to specify a results directory, it is this name. For example, in Figure D.5, `exp.res` is the name of the results directory.

Within the results directory, each analysis type creates a results file of the circuit name

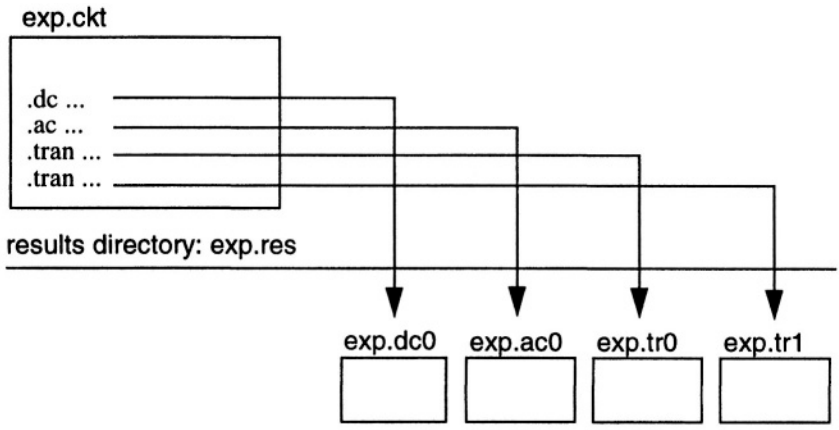


FIGURE D.5 Example circuit file, `exp.ckt`, with correspond results directory, `exp.res` (at the same level in the filesystem hierarchy). Within the results directory, files are created for each of the analyses specified in the circuit file.

with a corresponding extension indicating the type of analysis. An index identifier is used in the suffix of the filename for differentiating the results files of different analysis of the same type.

You can change the destination of the results directory via the *Design->Settings* menu to be another location such as a common temporary directory. This can be used to simplify data management (see Section D.2, *Installation and Setup*) as unwanted results directories can be easily identified and removed.

D.3 Using the Explorer IDE

This section provides simple walk-through examples of the Verilog-A Explorer IDE. The first example will include opening and running an existing design and plotting results. The second example will create a circuit file and Verilog-A module from scratch.

D.3.1 Opening and Running an Existing Design

First start the Explorer IDE and from the main menu, select *File->Open.* From the *Open File* dialog, select the file **`pll.ckt`** from the `<install_dir>\tutorial` directory as in Figure D.6.

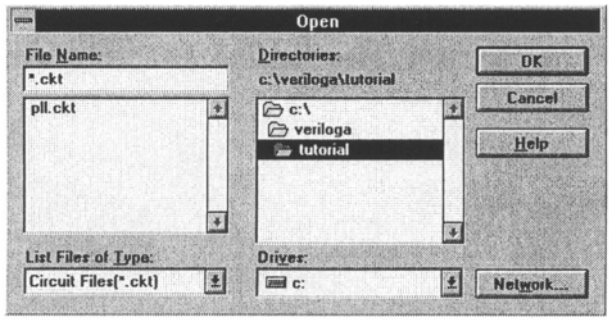


FIGURE D.6 Opening the `pll.ckt` circuit file.

This will load the circuit file into the Explorer IDE workspace. As everything is already defined for this example circuit, you can begin simulation. From the Explorer IDE toolbar, Figure D.7, press the start simulation button, or from the *Design->Start Simulation* menu entry.

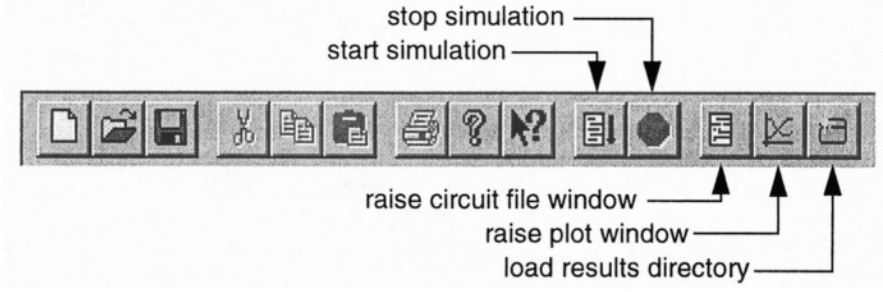


FIGURE D.7 Explorer IDE toolbar extensions.

Output from the Spice-SL simulation will be displayed in the Output Window (Figure D.1) as the simulation progresses. If there are errors in the circuit, all information will be output to the Output Window.

When the simulation completes successfully, either

- The output results of the simulation will be auto-loaded into the workspace if there was only one analysis in the circuit file.
- Or, a *Load Results* dialog will be raised asking you to select the results you would like to look at if there were multiple analysis (**`.dc`**, **`.tran`**, **`.ac`**) performed (Figure D.8):

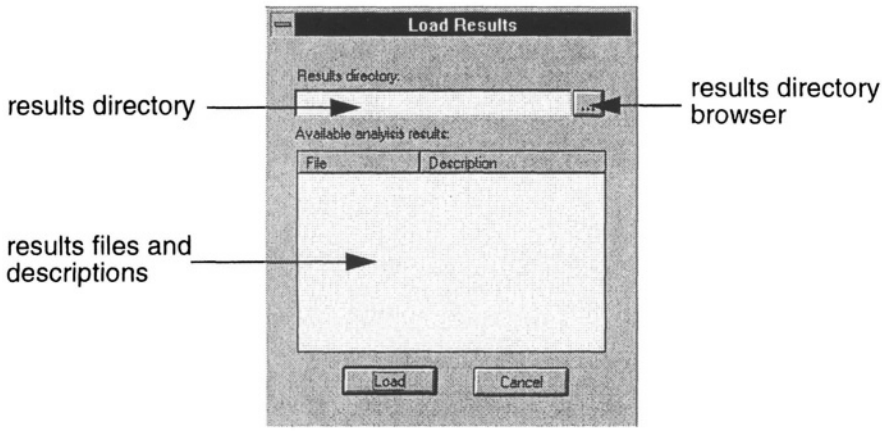


FIGURE D.8 Load Results dialog and description.

The *Load Results* dialog displays the available result files that can be loaded into the workspace. If the results directory entry is empty, to the right of the results directory name is a button which raises a directory browser (Figure D.9) which allows you to specify the directory containing the output results files of interest.

After the results directory is defined, you can choose one of the results files (Figure D. 10). A results file will be identified by name (`pll.tr0`) and description of the type of corresponding analysis.

When a results file is read into the environment, the Explorer IDE will use the information in the results file to fill out the Project Navigator (Figure D.1). The Project Navigator allows you to plot signals (via the hierarchy browser) and manage the files associated with the design.

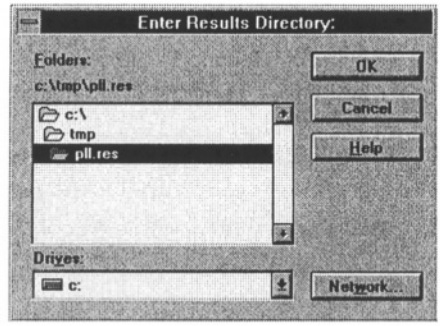


FIGURE D.9 Results Directory Browser dialog. Navigate the filesystem hierarchy until the appropriate directory is high-lighted.

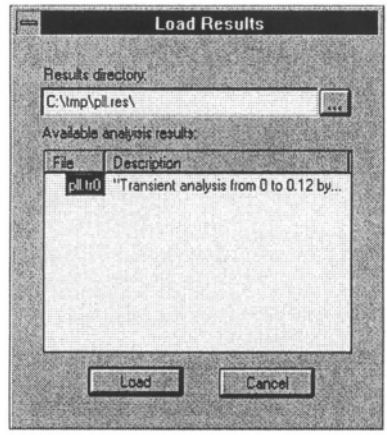


FIGURE D.10 Load Results dialog with results directory specified and all results files enumerated.

For `pll. ckt`, loading the results file `pll.tr0` results in the Project Navigator of Figure D. 11. The Project Navigator shows a hierarchical view of both signals and components within the design. If there is a plus (+) box to the left of an icon, that indicates that there is another level of hierarchy in the design below that component.

The Project Navigator is used for both traversing the hierarchy as well as specifying the output signals to be plotted. Once a results file has been loaded into the Explorer

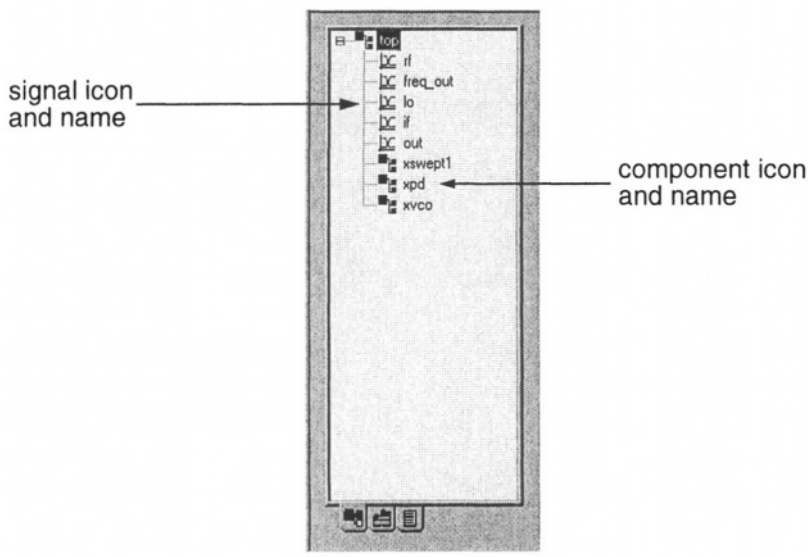


FIGURE D.11 Project Navigator showing a hierarchical view of signals and components.

workspace, you can display simulation results with the waveform viewer. From the main menu, select *View->Plot Window* to raise the plot window (or use the toolbar shortcut - Figure D.7). To add signals to the plot window, simply double-click on a

signal icon within the hierarchy view of the Project Navigator. For example, double-clicking on signal icon `out` results in shown in Figure D.12.

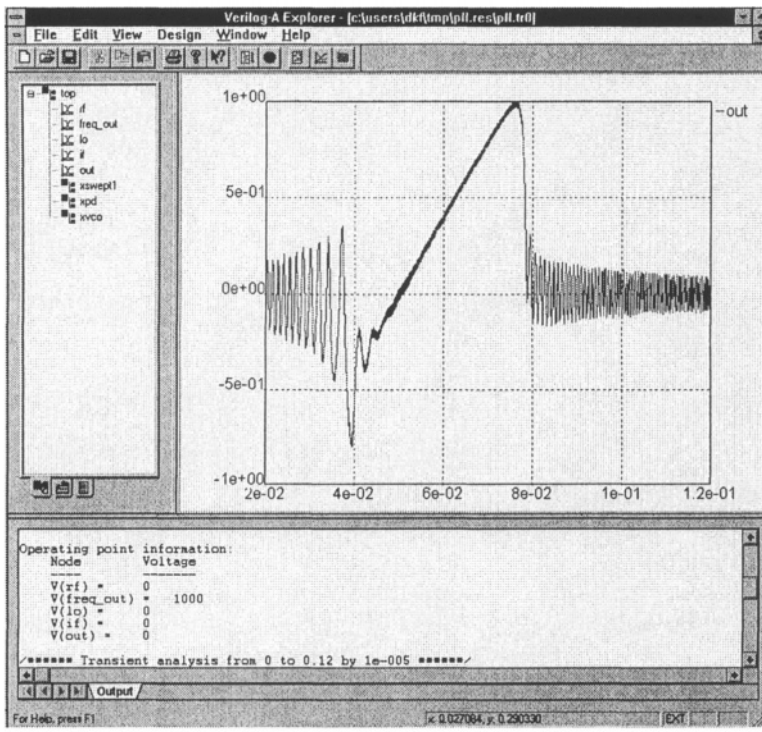


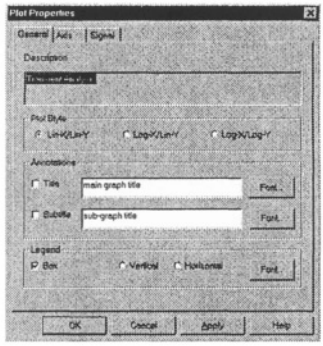
FIGURE D.12 Plotting signals within the IDE. To plot a signal, raise the plot window and double-click on the signal icon of interest in the hierarchy navigator. To delete a trace, click on a signal name in the plot legend to select. Delete with the <DEL> key.

Signals can be deleted from the plot view by clicking on the signal within the plot legend. This will select the signal by placing a box around the name. If a signal is selected, it can be deleted simply by pressing the <DEL> key.

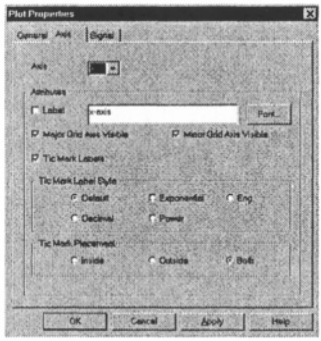
Zooming in on a specific area of the plot view is accomplished by left-mouse button drag operations. To zoom back out, press the right-mouse button in the plot window and choose either *Zoom To Fit* or *Zoom Out.*

After a plot has been selected, you can change its properties via the *Plot Properties* dialog accessible from the right-mouse button within the plot window. The *Plot Properties* dialog allows you to set generic, axis, and signal attributes.

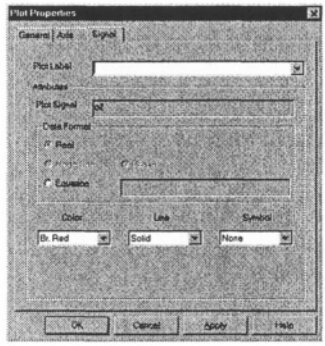
Generic plot attributes include the plot type as well as display of titles and/or subtitles.



Axis properties of the plot allow you to set the axis styles for both the X- and Y-axis, including labels and tic-mark styles.

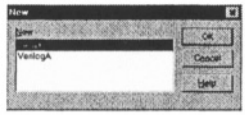


Signal properties allow you to edit the description of the signals displayed in the legend box, as well as the data format and drawing attributes.



D.3.2 Creating a New Designs

Starting a new design follows essentially the same procedure as previously outlined, but with the addition to creating a new circuit and/or Verilog-A file(s). From the main Explorer menu, select *File->New,* which raises the following dialog box:



If you select a circuit file, the workspace will be cleared of any open files. If you select a Verilog-A file, it is assumed that it is associated with any existing circuit design open within the workspace. In both cases, a new file is created and initialized with a template file of the appropriate type. If you prefer your own template files, change the path of the template via the respective *Editor Properties* dialog accessible via the right mouse button.

Appendix E

Spice Quick Reference

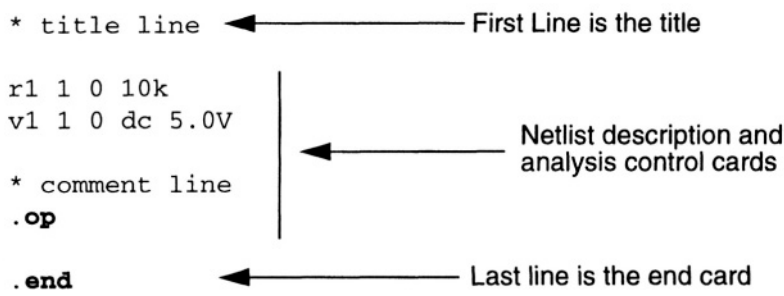
E.1 Introduction

Spice is a general-purpose circuit simulation program for nonlinear DC, nonlinear transient, and linear AC analysis. Originating from the University of California at Berkeley, is by far the best known and most widely used circuit simulator. It is available in for a wide variety of computer platforms, in both commercial and proprietary derivatives of the original version.

Newer versions of Spice offer many extensions, but the input format for circuit descriptions reflect the original batch-oriented program architecture. This appendix overviews the Spice input format, or netlist files including the fundamental types and analyses supported. Omitted for brevity are details regarding semiconductor device models and the various Spice options.

E.2 Circuit Netlist Description

The netlist (also referred to as the input deck) consists of element lines which describes both the circuit topology and element values and control lines which describe analyses to be performed for Spice. The first card in the input deck must be a title card, and the last card must be the **.END** control line. The order of the remaining element and control lines is arbitrary.



The input format is free format. Fields on an element or control line are separated by one or more blanks, commas, equal (=) sign, or a left or right parenthesis. A element or control line may be continued by placing a (+) in column 1 on the following line. Spice will continue reading beginning with column 2.

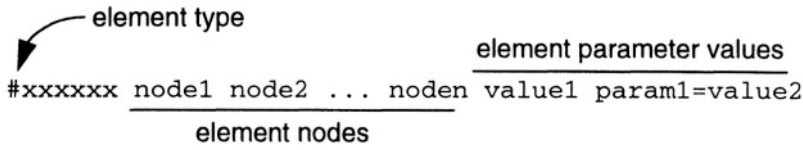
Name fields must begin with a letter [a-z] and cannot contain any delimeters. Names within Spice netlists are considered case-insensitive[1]. An integer or a floating point number can be followed by one of the following scale factors:

G = 1.0e9
MEG = 1.0e6
K = 1.0e3
MIL = 25.4e-4
M = 1.0e-3
U = 1.0e-6
N = 1.0e-9
P = 1.0e-12

1. Names in Verilog are case-sensitive requiring a certain level of awareness for modelers in developing Verilog-A models that are case-independent for use within Spice netlists.

Letters immediately following a scale factor are ignored.

Each element in the circuit is specified by an element line that contains the element name, the circuit nodes to which the element is connected, and the values of the parameters that determine the electrical characteristics of the element.



The first letter of the element name specifies the element type. The nodes following the element name must be non-negative integers but need not be numbered sequentially and where node 0 is the ground or reference node.

A control line within the input deck is specified by a line containing a (.) in the first column, followed by the name of the control and its parameters. Examples include all the analysis cards (described later) and the **.END** control line signifying the end of input.

E.3 Components

Circuits in Spice may contain resistors, capacitors, inductors, mutual inductors, independent voltage and current sources, dependent sources, transmission lines and the four most common semiconductor devices: diodes, bipolar junction transistors, junction field-effect transistors, and mosfets. The general input formats for each of these types is described below. Arguments specified within [] are optional.

E.3.1 Elements

Passive elements in Spice such as resistors (R), capacitors (C), and inductors (L):

```
Rxxxxxxx NP NN value
Cxxxxxxx NP NN value
Lxxxxxxx NP NN value
```

Linear dependent sources including voltage-controlled current sources (G), voltage-controlled voltage sources (E), current-controlled current sources (F), and current-controlled voltage sources (H):

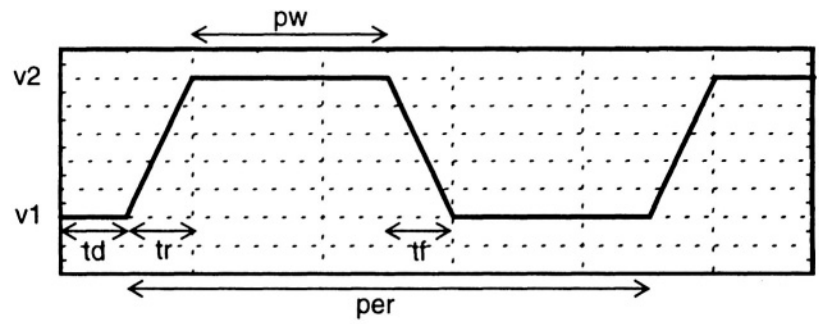
```
Gxxxxxxx NP NN NCP NGN value
Exxxxxxx NP NN NCP NCN value
Fxxxxxxx NP NN vname value
Hxxxxxxx NP NN vname value
```

where <vname> is the source through which the controlling current is measured. Independent voltage and current sources are specified in Spice as:

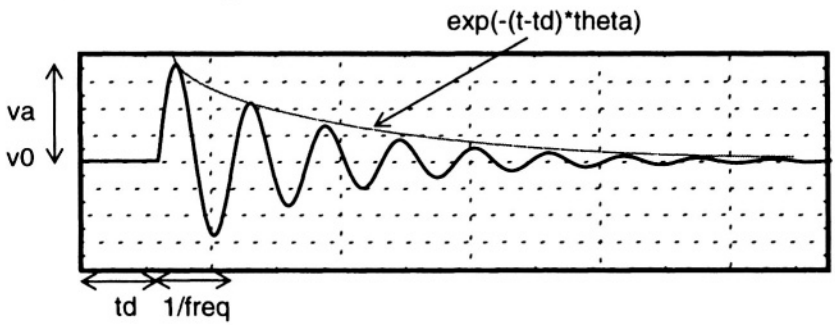
```
Vxxxxxxx NP NN [[DC] dctr_value] [AC [acmag [acphase]]
Ixxxxxxx NP NN [[DC] dctr value] [AC [acmag [acphase]]
```

where dctran_val is a constant value for time-independent sources, and one of the following for time-dependent sources:

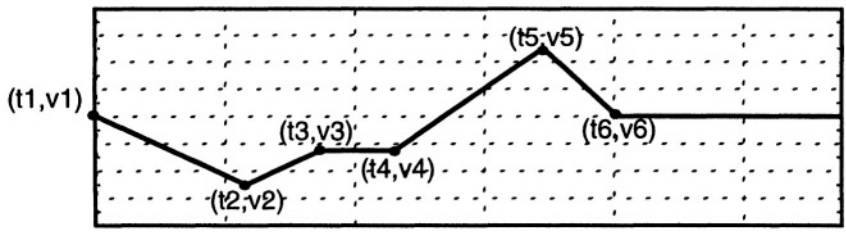
pulse(v1 v2 td tr tf pw per)



sin(v0 va freq td theta)



pwl(t1 v1 [t2 v2 [t3 v3 [....]]])



For AC small-signal analyses, at least one AC source must be defined in the circuit.

E.3.2 Semiconductor Devices and Models

For semiconductor devices, the large number of parameters require that the device model parameters be specified on a separate **.MODEL** definition and assigned a unique model name. The device element cards in Spice then reference the model name.

Each device element card contains the device name, the nodes to which the device is connected to, and the device model name. The standard semiconductor devices supported by Spice include diodes (D), bipolar junction transistors (Q), junction field-effect transistors (J), and mosfets (M) devices.

```
Dxxxxxxx NP NN MNAME [area]
Qxxxxxxx NC NB NE MNAME [area]
Jxxxxxxx ND NG NS MNAME [area]
Mxxxxxxx ND NG NS NB MNAME [w=value] [l=value]
```

Where MNAME is the model name. The model name is defined using a **.MODEL** card, assigning parameters by appending the parameter name for the given model type with an equal sign and the parameter value. Model parameters not given are assigned the default values for the model. The general format of **.MODEL** cards is:

```
.MODEL MNAME TYPE ( P1=VAL1 P2=VAL2 ... )
```

and TYPE is one of the following:

```
NPN       NPN bjt model
PNP       PNP bjt model
D         Diode model
```

```
NJF       N-channel jfet model
PJF       P-channel jfet model
NMOS      N-channel mosfet model
PMOS      P-channel mosfet model
```

For information on the specific model types and associated parameters, refer to more complete documentation.

E.4 Analysis Types

E.4.1 Operating Point Analysis

The DC analysis portion of Spice determines the operating point of the circuit with inductors shorted and capacitors opened. An operating point analysis is specified using:

```
.OP
```

In addition, an operating point analysis is performed automatically prior to a transient analysis to determine the transient initial conditions, and prior to an AC small-signal analysis to determine the linearized, small-signal models for nonlinear devices.

E.4.2 DC Transfer Curve Analysis

A DC transfer curve analysis can be used to examine the response of the circuit to a range of input conditions. A transfer curve analysis is specified using:

```
.DC srcname <srcstart> <srcstop> <srcincr>
```

Where `srcname` is the name of an independent voltage or current source. `<srcstart>`, `<srcstop>`, and `<srcincr>` are the starting, final, and incrementing values of the transfer curve analysis respectively.

E.4.3 Transient Analysis

The transient analysis command of Spice computes the transient output variables as a function of time over a user-specified time interval. The initial conditions are automatically computed by an operating point analysis. A transient analysis is specified using:

```
.TRAN <tstep> <tstop> [<tstart> [<tmaxstep>] ]
```

Where <tstep> is the printing increment, <tstop> is the final time, and <tstart> is the initial time. If <tstart> is omitted, it is assumed to be zero. <tmaxstep> is the maximum stepsize that Spice will use (defaults to <tstop>/ 50.0).

E.4.4 AC Small-signal Analysis

The AC small-signal portion of spice computes the AC output variables as a function of frequency. Spice first computes the operating point of the circuit and determines linearized small-signal models for all the nonlinear devices in the circuit. The resultant linear circuit is then analysed over a user-specified range of frequencies. An AC small-signal analysis is specified using:

```
.AC DEC <numdec> <fstart> <fstop>
.AC OCT <numoct> <fstart> <fstop>
.AC LIN <numlin> <fstart> <fstop>
```

Where DEC stands for decade variation, and <numdec> is the number of points per decade, OCT stands for octave variation and <numoct> is the number of points per octave, and LIN stands for linear variation and <numlin> is the number of points. Note, that for AC small-signal analysis to be meaningful, at least one independent source must have been specified with an AC value.

Index

D

E

F

I

K

L

M

N

This package contains a diskette that includes software described in this book. See the applicable appendix for descriptions of this program and instructions for its use. By opening this package you are agreeing to be bounded by the following:

> The software contained on this diskette is copyrighted and all rights are reserved by Apteq Design Systems, Inc.
>
> THIS SOFTWARE IS PROVIDED FREE OF CHARGE, AS IS, AND WITHOUT WARRANTY OF ANY KIND, EITHER OR EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANT-ABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Neither Apteq Design Systems, Inc., Kluwer Academic Publishers, Inc., its dealers and distributors assumes any liability for any alleged or actual damages arising from the use of this software.