Summary of Key Verilog Features (IEEE 1364)

Module

Encapsulates functionality; may be nested to any depth.

module module_name (list of ports);
Declarations
Port modes: **input**, **output**, **inout** identifier;
Nets (e.g., **wire** A[3:0];)
Register variable (e.g., **reg** B[31: 0];)
Constants: (e.g. **parameter** size = 8;)
Named events
Continuous assignments
(e.g. **assign** sum = A + B;)
Behaviors **always** (cyclic), **initial** (single-pass)
specify … **endspecify**
function … endfunction
task … endtask
Instantiations
primitives
modules
endmodule

Multi Input Primitives

(Each input is a scalar)

and (out, in_1, in_2, … in_N);
nand (out, in_1, in_2, … in_N);
or (out, in_1, in_2, … in_N);
nor (out, in_1, in_2, … in_N);
xor (out, in_1, in_2, … in_N);
xnor (out, in_1, in_2, … in_N);

Multi-Output Primitives

buf (out_1, out_2, …, out_N, in); // buffer
not (out_1, out_2, …, out_N, in); // inverter

Three-State Multi-Output Primitives

bufif0 (out, in, control); **bufif1** (out, in, control);
notif0 (out, in, control); **notif1** (out, in, control);

Pullups and Pulldowns

pullup (out_y); **pulldown** (out_y);

Propagation Delays

Single delay:	**and** #3 G1 (y, a, b, c);
Rise/fall:	**and** #(3, 6) G2 (y, a, b, c);
Rise/fall/turnoff:	**bufif0** #(3, 6, 5) (y, x_in, en);
Min:typ:Max:	**bufif1** #(3:4:5, 4:5:6, 7,8:9) (y, x_in, en);

Command line options for single delay value simulation: **+maxdelays**, **+typdelays**, **+mindelays**

Example: verilog +mindelays testbench.v

Concurrent Behavioral Statements

May execute a level-sensitive assignment of value to a net (keyword: **assign**), or may execute the statements of a cyclic (keyword: **always**) or single-pass (keyword: **initial**) behavior. The statements execute sequentially, subject to level-sensitive or edge-sensitive event control expressions.

Syntax:

assign net_name = [expression];
always begin [procedural statements] **end**
initial begin [procedural statements] **end**

Cyclic (**always**) and single-pass (**initial**) behaviors may be level sensitive and/or edge sensitive.

Edge sensitive:

always @(posedge clock)
q <= data;

Level sensitive:

always @ (enable or data)
if (enable) q = data

Data Types: Nets and Registers

Nets: Establish structural connectivity between instantiated primitives and/or modules; may be target of a continuous assignment; e.g., **wire**, **tri**, **wand**, **wor**.

Value is determined during simulation by the driver of the net; e.g., a primitive or a continuous assignment. (Example: **wire** Y = A + B.)

Registers: Store information and retain value until reassigned.

Value is determined by an assignment made by a procedural statement.

Value is retained until a new assignment is made; e.g., **reg**, **integer**, **real**, **realtime**, **time**.

Example:

always @ (posedge clock)
if (reset) q_out <=0;
else q_out <= data_in;

Procedural Statements

Describe logic abstractly; statements execute sequentially to assign value to variables.

if (expression_is_true) statement_1; **else**
statement_2;
case (case_expression)
case_item: statement;
…
default: statement;
endcase
for (conditions) statement;
repeat constant_expression statement;
while (expression_is_true) statement;
forever statement;
fork statements **join** // execute in parallel

Assignments

Continuous: Continuously assigns the value of an expression to a net.

Procedural (Blocked): Uses the = operator; executes statements sequentially; a statement cannot execute until the preceding statement completes execution. Value is assigned immediately.

Procedural (Nonblocking): Uses the <= operator; executes statements concurrently, independent of the order in which they are listed. Values are assigned concurrently.

Procedural (Continuous):

assign … **deassign** overrides procedural assignments to a net.

force … **release** overrides all other assignments to a net or a register.

Operators

{ }, {{ }}	concatenation
+ - * /	arithmetic
%	modulus
> >= < <=	relational
!	logical negation
&&	logical and
||	logical or
==	logical equality
!=	logical inequality
===	case equality
!==	case inequality
~	bitwise negation
&	bitwise and
|	bitwise or
^	bitwise exclusive or
^~ or ~^	bitwise equivalence
&	reduction and
~&	reduction nand
|	or
~|	reduction nor
^	reduction exclusive or
~^ or ^~	reduction xnor
<<	left shift
>>	right shift
?:	conditional
or	Event or

Specify Block

Example: Module Path Delays

specify
// specparam declarations (min: typ: max)
specparam t_r = 3:4:5, t_f = 4:5:6;
(A, B) *> Y) = (t_r, t_f); // full
(Bus_1 => Bus_1) = (t_r, t_f); // parallel
if (state == S0) (a, b *> y) = 2; // state dep
(**posedge** clk => (y <: d_in)) = (3, 4); // edge
endspecify

Example: Timing Checks

specify
specparam t_setup = 3:4:5, t_hold = 4:5:6;

$setup (data, posedge clock, t_setup);
$hold (posedge clock, data, t_hold);
endspecify

Memory

Declares an array of words.

Example: Memory declaration and readout

module memory_read_display();
reg [31: 0] mem_array [1: 1024];
integer k;

Advanced Digital Design with the Verilog HDL

Advanced Digital Design with the Verilog HDL

Michael D. Ciletti
Department of Electrical and Computer Engineering
University of Colorado at Colorado Springs

Prentice Hall

Pearson Education, Inc.
Upper Saddle River, New Jersey 07458

Preface

Simplify, Clarify, and Verify

Behavioral modeling with a hardware description language (HDL) is the key to modern design of application-specific integrated circuits (ASICs). Today, most designers use an HDL-based design method to create a high-level, language-based, abstract description of a circuit, synthesize a hardware realization in a selected technology, and verify its functionality and timing.

Students preparing to contribute to a productive design team must know how to use an HDL at key stages of the design flow. Thus, there is a need for a course that goes beyond the basic principles and methods learned in a first course in digital design. This book is written for such a course.

Many books discussing HDLs are now available, but most are oriented toward robust explanations of language syntax, and are not well-suited for classroom use. Our focus is on design methodology enabled by an HDL.

Our goal in this book is to build on a student's background from a first course in logic design by (1) reviewing basic principles of combinational and sequential logic, (2) introducing the use of HDLs in design, (3) emphasizing descriptive styles that will allow the reader to quickly design working circuits suitable for ASICs and/or field-programmable gate array (FPGA) implementation, and (4) providing in-depth design examples using modern design tools. Readers will be encouraged to simplify, clarify, and verify their designs.

The widely used Verilog hardware description language (IEEE Standard 1364) serves as a common framework supporting the design activities treated in this book, **but our focus is on developing, verifying, and synthesizing designs of digital circuits, not on the Verilog language**. Most students taking a second course in digital design will be familiar with at least one programming language and will be able to draw on that background in reading this textbook. We cover only the core and most widely used features of Verilog. In order to emphasize *using* the language in a synthesis-oriented design environment, we have purposely placed many details, features, and explanations of syntax in the Appendices for reference on an "as-needed" basis.

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

The Verilog language is introduced in an integrated, but selective manner, only as needed to support design examples. The text has a large set of examples illustrating how to address the key steps in a very large scale integrated (VLSI) circuit design methodology using the Verilog HDL. Examples are complete, and include source code that has been verified with the Silos-III simulator to be correct. Source code for all of the examples will be available (with important test suites) at our website.

The Intended Audience

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

Special Features of the Book

- Begins with a brief review of basic principles in combinational and sequential logic
- Focuses on modern digital design methodology
- Illustrates and promotes a synthesis-ready style of register transfer level (RTL) and algorithmic modeling with Verilog
- Demonstrates the utility of ASM charts for behavioral modeling
- In-depth treatment of algorithms and architectures for digital machines (e.g., an image processor, digital filters and circular buffers)
- In-depth treatment of synthesis for cell-based ASICs and FPGAs
- A practical treatment of timing analysis, fault simulation, testing, and design for testability, with examples
- Comprehensive treatment of behavioral modeling
- Comprehensive design examples, including a RISC machine and datapath controller
- Numerous graphical illustrations
- Provides several problems with a wide range of difficulty after each chapter
- Contains a worked example with JTAG and BIST for testing

- Contains over 250 fully verified examples
- An indexed list of all models developed in the examples
- A set of Xilinx FPGA-based laboratory-ready exercises linked to the book (e.g., arithmetic and logic unit [ALU], a programmable lock, a key pad scanner with a FIFO, a serial communications link with error correction, an SRAM controller, and first in, first out [FIFO] memory)
- Contains an up-to-date chapter on programmable logic device (PLDs) and FPGAs
- Contains a packaged CD-ROM with the popular Silos-III Verilog design environment and simulator and the Xilinx integrated synthesis environment (ISE) synthesis tool for FPGAs
- Contains an Appendix with full formal syntax of the Verilog HDL
- Covers major features of Verilog 2001, with examples
- Supported by an ongoing website containing:

1. Source files of models developed in the examples
2. Source files of testbenches for simulating examples
3. An Instructor's Classroom Kit containing transparency files for a course based on the subject matter
4. Solutions to selected problems
5. Jump-start tutorials helping students get immediate results with the Silos-III simulation environment, the Xilinx FPGA synthesis tool, the Synopsys synthesis tools, and the Synopsys Prime Time static timing analyzer
6. ASIC standard-cell library with synthesis and timing database
7. Answers to frequently asked questions (FAQs)
8. Clever examples submitted by readers
9. Revisions

Sequences for Course Presentation

The material in the text begins with a review of combinational and sequential logic design, but then progresses in the order dictated by the design flow for an ASIC or an FPGA. Chapters 1 to 6 treat design topics through synthesis, and should be covered in order, but Chapters 7 to 10 can be covered in any order. The homework exercises are challenging, and the laboratory-ready Xilinx-based exercises are suitable for a companion laboratory or for end-of-semester projects. Chapter 10 presents several architectures for arithmetic operations, affording a diversity of coverage. Chapter 11 treats postsynthesis design validation, timing analysis, fault simulation, and design for testability. The coverage of these topics can be omitted, depending on the level and focus of the course. Tools supporting Verilog 2001 are emerging, so an appendix discusses and illustrates the important new features of the language.

Chapter Descriptions

Chapter 1 briefly discusses the role of HDLs in design flows for cell-based ASICs and FPGAs. Chapters 2 and 3 review mainstream topics that would be covered in a first course in digital design, using classical methods (i.e. Karnaugh maps). This material will refresh the reader's background, and the examples will be used later to introduce HDL-based methods of design. Chapters 4 and 5 introduce modeling of combinational and sequential logic with the Verilog HDL, and place emphasis on coding styles that are used in behavioral modeling. Chapter 6 addresses cell-based synthesis of ASICs, and introduces synthesis of combinational and sequential logic. Here we pursue two main objectives: (1) present synthesis-friendly coding styles, and (2) form a foundation that will enable the reader to anticipate the results of synthesis, especially when synthesizing sequential machines. Many sequential machines are partitioned into a datapath and a controller. Chapter 7 covers examples that illustrate how to design a controller for a datapath. The designs of a simple RISC CPU and a UART[1] serve as platforms for the subject matter. Chapter 8 covers PLDs, complex PLDs (CPLDs), ROMs, and static random-access memories (SRAMs), then expands the synthesis target to include FPGAs. Verilog has been used extensively to design computers and signal processors. Chapter 9 treats the modeling and synthesis of computational units and algorithms found in computer architectures, digital filters, and other processors. Chapter 10 develops and refines algorithms and architectures for the arithmetic units of digital machines. In Chapter 11 we use the Verilog HDL in conjunction with fault simulators and timing analyzers to revisit a selection of previously designed machines and consider performance/timing issues and testability, to complete the treatment of design flow tasks that rely heavily on designer intervention. Chapter 11 models the test access port (TAP) controller defined by the IEEE 1149.1 standard (commonly known as the JTAG standard), and presents an example of its use. Another elaborate example covers built-in self test (BIST).

Acknowledgments

The author is grateful for the support of colleagues and students who expanded his vision of Verilog and contributed to this textbook. The reviewers of the original manuscript provided encouragement, critical judgment, and many helpful suggestions. Stu Sutherland helped the author gain a deeper appreciation for the issue of race conditions that can creep into the models of a digital system. These insights led to the disciplined style of adhering to nonblocking assignments for modeling edge-sensitive behavior and blocked assignments for modeling level-sensitive behavior. I owe a debt of gratitude to Dr. Jim Tracy and Dr. Rodger Ziemer, who supported my efforts to develop courses in VLSI circuit design; to Bill Fuchs, who introduced me to the Silos-III Verilog simulator from Simucad, Inc., and placed a user-friendly design environment in the hands of our students. Kirk Sprague and Scott Kukel were helpful in developing a Hamming encoder to work with the UART. Cris Hagan's thesis led to the models presented in Chapter 9 for decimators and other functional units found in digital signal processors. Rex Anderson proofread several chapters and scrubbed down my work. Terry Hansen and Lisa Horton provided the inspiration for the coffee vending machine example, and developed the assembler that supports the RISC CPU. Dr. Greg Sajdak developed material relating chip defects to test coverage and process yield. Dr. Bruce Harmon provided material for a FIR filter example. My editors, Tom Robbins and Eric Frank, have been a delight to work with. They supported the concept, encouraged my work and guided this book through the production process. My deep thanks to all of you.

[1]Universal asynchronous receiver and transmitter (UART), a circuit used in data transmission between systems.

Dedication

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