



# Introduction to Verilog

by

Bob Zeidman

# Table of Contents

Introduction.....	1
Chapter 1: About Hardware Description Languages.....	3
Chapter 2: Basic Verilog Syntax .....	Error! Bookmark not defined.
Chapter 3: Register, Net, and Parameter Data Types .....	Error! Bookmark not defined.
Chapter 4: Modules .....	Error! Bookmark not defined.
Chapter 5: Operators and Expressions.....	Error! Bookmark not defined.
Chapter 6: Assignments .....	Error! Bookmark not defined.
Chapter 7: Execution Control Statements.....	Error! Bookmark not defined.
Chapter 8: Functions and Tasks.....	Error! Bookmark not defined.
Chapter 9: Procedural Blocks .....	Error! Bookmark not defined.
Chapter 10: Compiler Directives, System Tasks and Functions .....	Error! Bookmark not defined.
Chapter 11: Coding Guidelines.....	Error! Bookmark not defined.
Final Exam.....	Error! Bookmark not defined.
Resources for Further Study.....	Error! Bookmark not defined.
Answers to Problems.....	Error! Bookmark not defined.

# Introduction

This book is based on the popular Verilog seminar that I give at conferences around the world. Over the years I've used the feedback from students to try to make it the best introductory Verilog seminar available. I hope I've succeeded, and I hope I've succeeded in turning that seminar into a useful book. If you want to comment, either to congratulate me on the excellent job I've done, to ask a question, to point out a mistake or misconception, or to suggest improvements for the future, or simply to complain, please do so. I welcome all feedback.

## Course Objectives

Hardware Description Languages (HDLs) use statements, like programming language statements, in order to define, simulate, synthesize, and layout hardware. One of the main HDLs is Verilog, a widely used and standardized language. Verilog can be used to design anything from the most complex ASIC to the least complex PAL. As ASICs and FPGAs become more complex, HDLs become a necessity for their design.

This book teaches how to use Verilog to design and simulate hardware. It begins by explaining the benefits of HDLs over other design entry methods, including its ability to model different levels of abstraction, its reusability, and documentability. Next, the syntax of the Verilog language is explained in detail. By the end of the course, the student should be able to design and simulate real hardware using Verilog.

## Intended Audience

The book is aimed at electrical engineering students and practicing electrical engineers who are not yet familiar with Verilog. It is intended for engineers who wish to cover a lot of ground toward understanding and using Verilog to create real designs. The prerequisite is a good knowledge of digital logic design. Knowledge of programming languages, such as C or C++, and knowledge of other Hardware Description Languages (HDLs) such as VHDL, is beneficial but not mandatory.

## How To Use This Book

This book is divided into chapters. Each chapter is divided into sections. Each section has an icon to give some idea about it. The sections and their corresponding icons are described below.



### Objectives

At the beginning of each chapter in this book, the *Objectives* section will describe the goals of the particular chapter. This enables you to know the concepts that will be discussed and examined in the section.



### Comments

This is where I get to shoot my mouth off. I will give my opinions, anecdotes, and real-world examples relating to the subject. Many students find these tangents to be very interesting, breaking up the otherwise dry material. Others, however, think it's a great waste of time. If you fall into the latter category, feel free to skip these sections. You won't miss any crucial material (you'll simply hurt my feelings). Comments are in grey boxes to separate them from critical subject matter.



## **Problems**

At the end of most chapters in this book, there will be a number of problems to solve based on the material in that chapter. These problems are designed not only to test you, but to reinforce the subject. You learn better by applying your knowledge to solve problems.



## **Solutions**

In this section you will find all of the answers to all of your problems. Too bad life isn't like this book.

## ***Acknowledgements***

I'd like to thank Ken McElvain, Chief Technical Officer of Synplicity, a brilliant engineer, for his technical assistance in reviewing examples and Verilog code and generally giving recommendations and advice. Thanks to Dan Hafeman of Mentor Graphics, another brilliant engineer who gave me advice and feedback. Carlo Treves was nice enough to proofread the manuscript while he was on vacation, and I'm grateful to him for his comments. Much credit and thanks go to my wife Carrie Zeidman, of Elan Studios, for doing the layout of the book and for creating the graphics as well as doing all of the groundwork for getting the book printed.

## ***About the Author***

Bob Zeidman is the president of Zeidman Consulting ([www.ZeidmanConsulting.com](http://www.ZeidmanConsulting.com)), a contract research and development firm. Since 1983, he has designed ASICs, FPGAs, and PC boards for RISC-based parallel processor systems, laser printers, network switches and routers, and other real time systems. His clients have included Apple Computer, Cisco Systems, Intel, Quickturn Design Systems, and Texas Instruments. Previously, Bob was the president of The Chalkboard Network, an e-learning company for high tech professionals. Among his publications are technical papers on hardware and software design methods as well as two other textbooks — *Designing with FPGAs and CPLDs* and *Verilog Designer's Library*. He has taught courses at engineering conferences throughout the world. Bob earned bachelor's degrees in physics and electrical engineering at Cornell University and a master's degree in electrical engineering at Stanford University.

## Chapter 1: About Hardware Description Languages



### Objectives

This chapter introduces you to the overall concept of hardware description languages and explains their many advantages over other methods of design entry, particularly schematic capture, for designing large, complex systems. The goal is to allow you to hold your own in an argument with an old-timer who still insists that schematic capture is the way to go.

---

---

Years ago, as integrated circuits grew in complexity, a better method for designing them was needed. Schematic capture tools had been developed which allowed an engineer to draw schematics on a computer screen to represent a circuit. This worked well because graphic representations are always useful to understand small but complex functions. But as chip density increased, schematic capture of these circuits became unwieldy and difficult to use for the design of large circuits. The transistor densities of Application Specific Integrated Circuits (ASICs) and Field Programmable Gate Arrays (FPGAs) grew to the point where a better tool was needed. That tool is a Hardware Description Language (HDL). As more engineers design complex systems on a chip, they have no option but to use HDLs. The major advantages of HDLs are:

1. Ability to handle large, complex designs
2. Different levels of abstraction
3. Top down design
4. Reusability
5. Concurrency
6. Timing
7. Optimization
8. Standards
9. Documentation



### The Learning Curve

I remember talking a number of years ago to another engineer at a startup company where we were both working. He was older than me - he had been one of the engineers that developed the original IBM 360 computer. He was now complaining about the schematic capture tools we were learning. "I hate these tools," he said. "They make it so tough. It used to be we would draw our schematics on paper with pencil. Make a mistake, simply erase it or start over. We didn't have to make sure each wire was straight or the right width. We didn't have to line up signal names or draw boxes with all these attributes. We didn't have to learn a whole new computer system. We just gave it to the draftsmen who cleaned them up and put everything in order. I spend too much time drawing and not enough time designing."

## Introduction to Verilog

This surprised me. I had worked one summer at a company where draftsmen cleaned up the schematics that I drew on paper. After that, every job I had, I used schematic capture tools. And loved them. To me, it took the power out of other people's hands and put it into mine. It may have taken me longer in the short term, but in the long term I didn't have to continually review schematics and look for miscommunications and mistakes. It meant that I could work my own hours without having to wait for a technician with whom I would have to review pages of schematics.

The other advantage of schematic capture was that suddenly, I could simulate my design. Something that was painful, if at all possible, previously. I could run automatic design rule checks. Modifications to existing designs were much easier. And a minimal level of documentation was standardized and almost automatic.

What the other engineer saw was the learning curve that he had to surmount rather than the eventual benefits. So now, many engineers may be inclined to react the same way to Hardware Description Languages (HDLs), the newest technology for designing circuits. But the biggest motivation for learning about HDLs is job security. In order to survive in the future, engineers will need to know HDLs. Industry analyst Dataquest (San Jose, CA) forecasts that most engineers that do not already use HDLs will be switching to HDL-based tools over the next few years. Luckily, you're one of the ones who decided to prepare now.

### What are HDLs?

Hardware Description Languages use statements, like programming language statements, in order to define, simulate, synthesize, and layout hardware. The two main HDLs are Verilog and VHDL. There are other, limited capability, languages such as ABEL, CUPL, and PALASM that are tailored specifically for designing PALs and CPLDs. They are not robust enough to cover the complexity required for most FPGAs and ASICs. However, both Verilog and VHDL can be used to design anything from the most complex ASIC to the least complex PAL. This paper will use examples in Verilog, and will focus on that language, but the general attributes of that language can be ascribed also to VHDL.

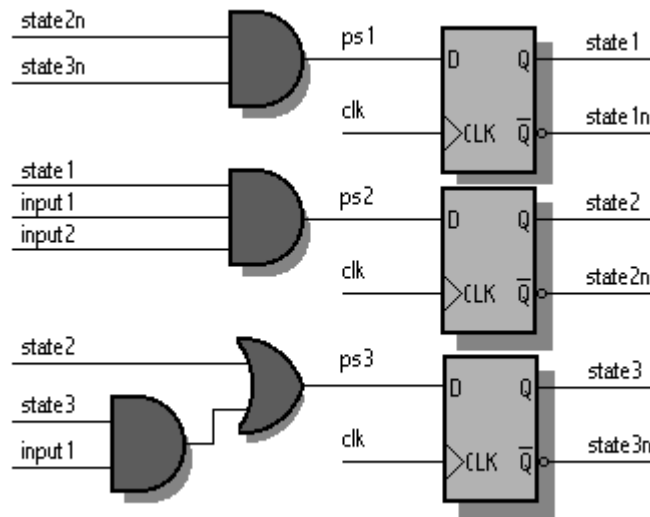


Figure 1. State machine schematic

A schematic for a simple state machine is shown in Figure 1. The equivalent Verilog code is shown in Example 1 below (don't worry about understanding the details of the code just yet -- we'll get to that later). One important difference to note is that schematic capture is limited to just the design aspect that involves physically selecting and connecting devices. HDLs, on the other hand, are used for all stages of the design as is explained further below, where the advantages of HDLs are enumerated.

```
// This module is used to implement the memory control state machine
```

## Introduction to Verilog

```
module state_machine(clk, input1, input2, state3);

    /* INPUTS */
    input clk;                // system clock
    input input1, input2;    // inputs from the adder

    /* OUTPUTS */
    output state3;           // can be used as the write signal

    /* DECLARATIONS */
    wire ps1;                // input to state1 flip-flop
    wire ps2;                // input to state2 flip-flop
    wire ps3;                // input to state3 flip-flop
    reg state1, state2, state3; // state bits

    assign ps1 = ~state2 & ~state3;
    assign ps2 = state1 & input1 & input2;
    assign ps3 = state2 | (state3 & input1);

    initial begin            // initialize the state machine
        state1 = 0;
        state2 = 0;
        state3 = 0;
    end

    always @(posedge clk) begin // clock in the new state on the
        state1 <= #3 ps1;      // rising edge of clk
        state2 <= #3 ps2;
        state3 <= #3 ps3;
    end
endmodule // state_machine
```

Example 1. State machine

### Different levels of abstraction

Algorithmic Level	Behavioral Models
Architectural Level	
Register Transfer Level (RTL)	Structural Models
Gate Level	
Switch Level	

Table 1. Levels of Abstraction

A hardware description language can be used to design at any level of abstraction from high-level architectural models to low-level switch models. These levels, from least amount of detail to most amount of detail, are given in

# Introduction to Verilog

Table 1. The top two levels use what are called Behavioral Models<sup>1</sup>, while the lower three levels use what are called Structural Models.

## Behavioral Models

Behavioral models consist of code that represents the behavior of the hardware without respect to its actual implementation. Behavioral models don't include timing numbers. Buses don't need to be broken down into their individual signals. Adders can simply add two or more numbers without specifying registers or gates or transistors. Behavioral models can be classified further as Algorithmic or Architectural.

## Algorithmic models

Algorithms are step-by-step methods of solving problems and producing results. No hardware implementation is implied in an algorithmic model. An algorithmic model is not concerned with clock signals, reset signals, or any kind of signals. It does not deal with flip-flops, gates, or transistors. There is no need to specify how a numeric quantity is represented in hardware, or what kind of memory device is used to store it.

For example, suppose we are designing an Arithmetic Logic Unit (ALU). This ALU takes two numeric inputs *operand\_a* and *operand\_b*, and a control input called *operation*. The ALU performs one of four operations, addition, subtraction, multiplication, or division, depending on the value of the *operation* input. The result is called *result\_c*. Our algorithmic model could look something like this:

```
if operation = 0, then result_c = operand_a + operand_b
if operation = 1, then result_c = operand_a - operand_b
if operation = 2, then result_c = operand_a * operand_b
if operation = 3, then result_c = operand_a / operand_b
```

In this description, we haven't considered how the numbers will be represented in hardware. Will they be 4-bit, 8-bit, or 16-bit quantities? How will we represent the sign of the number? How many clock cycles will it take to produce a result? These questions need not be considered when writing algorithmic models of hardware.

## Architectural models

Architectural models describe hardware on a very high level, using functional blocks like memory, control logic, CPU, FIFO, etc. These blocks may represent PC boards, ASICs, FPGAs, or other major hardware components of the system. An architectural model deals with hardware issues like how to represent signed integers, how much memory is required to store intermediate values, etc. An architectural description may involve clock signals, but usually not to the extent that it describes everything that happens in the hardware on every clock cycle. That is the responsibility of a Register Transfer Level (RTL) description. An architectural description is useful for determining the major functions of a design. An architectural model of our ALU might look like this:

```
main_block:

declare operand_a as a 16-bit bus
declare operand_b as a 16-bit bus
declare result_c as a 16-bit bus
declare mem as a 3 by 16 memory

wait for the rising edge of the clock signal
store operand_a in mem[0]
store operand_b in mem[1]
```

---

<sup>1</sup> I use my own nomenclature here for describing the various levels of abstraction. These levels do not have well-defined boundaries, and different individuals or organizations may have different, but equally valid, definitions.

## Introduction to Verilog

```
if operation = 0, then use adder_block
if operation = 1, then use subtractor_block
if operation = 2, then use multiplier_block
if operation = 3, then use divider_block
wait for five more rising edges of the clock signal
read result_c from mem[2]
```

In this example, the architecture of the hardware now consists of six sections: `main_block`, `adder_block`, `subtractor_block`, `multiplier_block`, `divider_block`, and `memory`. Only the `main_block` section is shown in the code above. A diagram of this architecture is shown in Figure 2.

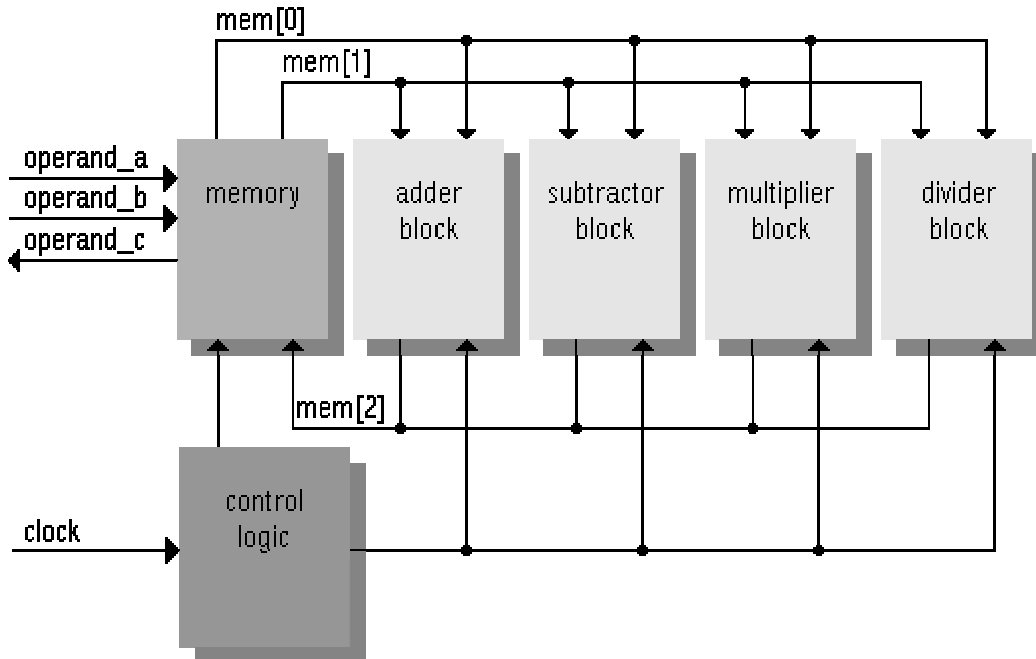


Figure 2. Architectural block diagram

We have divided the hardware into smaller building blocks, but each block still has a high level of functionality. We are concerned now with how to represent the numbers. We are concerned about clock signals, but not what the hardware is doing on every clock cycle.

### Structural Models

Structural models consist of code that represents specific pieces of hardware. Structural models can be further classified as register transfer level (RTL), gate level, or switch level models.

### Register transfer level (RTL) models

Register Transfer Level descriptions, commonly called RTL, specify hardware on a register level. In other words, RTL level code specifies what happens on each clock edge. Actual gates descriptions are avoided, although RTL code may use Boolean functions that can be implemented in gates. High level descriptions of hardware functions can be used, as long as these functions have a behavior that is associated with clock cycles. State machines are good examples of RTL descriptions. The functionality of a state machine can be complex, but the RTL state machine is defined by what occurs on each clock cycle. An example of some sequential logic is shown in Figure 3. The schematic is on the left while an RTL description is on the right.

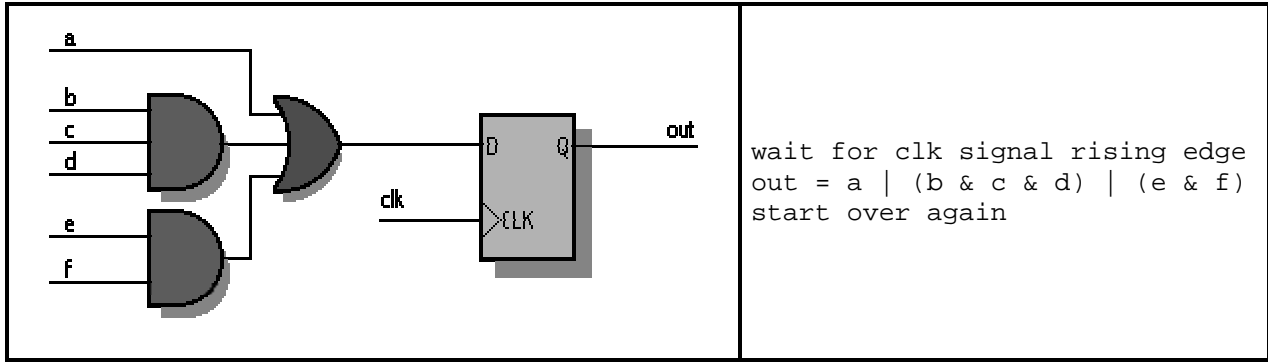


Figure 3. RTL Description of a Sequential Circuit

### Gate level models

Gate level modeling consists of code that specifies very simple functions such as NAND and NOR gates. The same sequential circuit shown in Figure 3 above would be described on a gate level using primitive NAND, NOR, and DFF gates as shown in Figure 4 below.

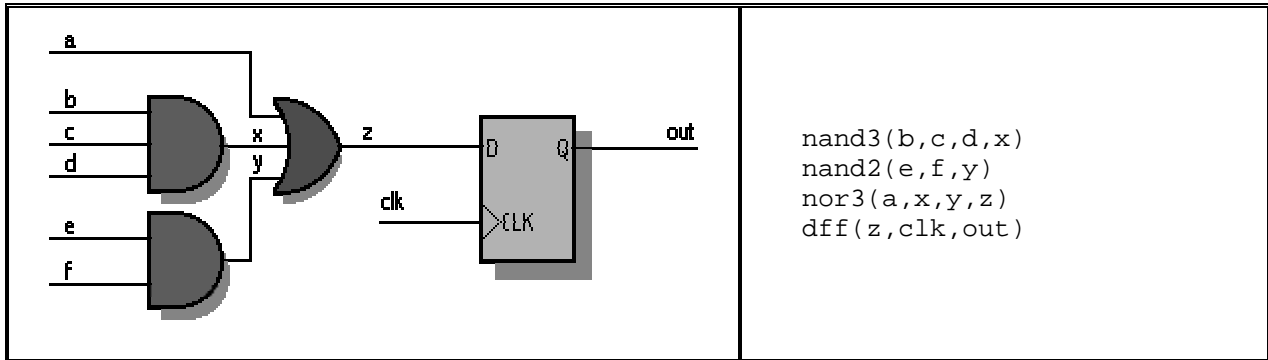


Figure 4. Gate Level Description of a Sequential Circuit

### Switch level models

Finally, the lowest level description is that of switch-level models, which specifies the actual transistor switches that are combined to make gates. ASIC vendors and other semiconductor manufacturers may use this level of modeling to simulate their circuits.

The advantage to HDLs is that all of these different levels of modeling can be done with the same language. This makes all the stages of design very convenient to implement. You don't need to learn different tools. You can easily simulate the design at a behavioral level, and then substitute various behavioral code modules with structural code modules. For system simulation, this allows you to analyze your entire project using the same set of tools. First, the algorithms can be tested and optimized. Next, the behavioral models can be used to partition the hardware into boards, ASIC, and FPGAs. The RTL code can then be written and substituted for behavioral blocks one at a time to easily test the functionality of each block. From that, the design can be synthesized into gate and switch level blocks that can be resimulated with timing numbers to get actual performance measurements. Finally this low level code can be used to generate a netlist for layout. All stages of the design have been performed using the same basic tool.

## Top-Down Design

Behavioral level modeling is particularly useful for doing top-down design. Top-down design is a methodology where the highest level functions and algorithms are described first, followed by more and more detailed designs that more directly relate to actual hardware.

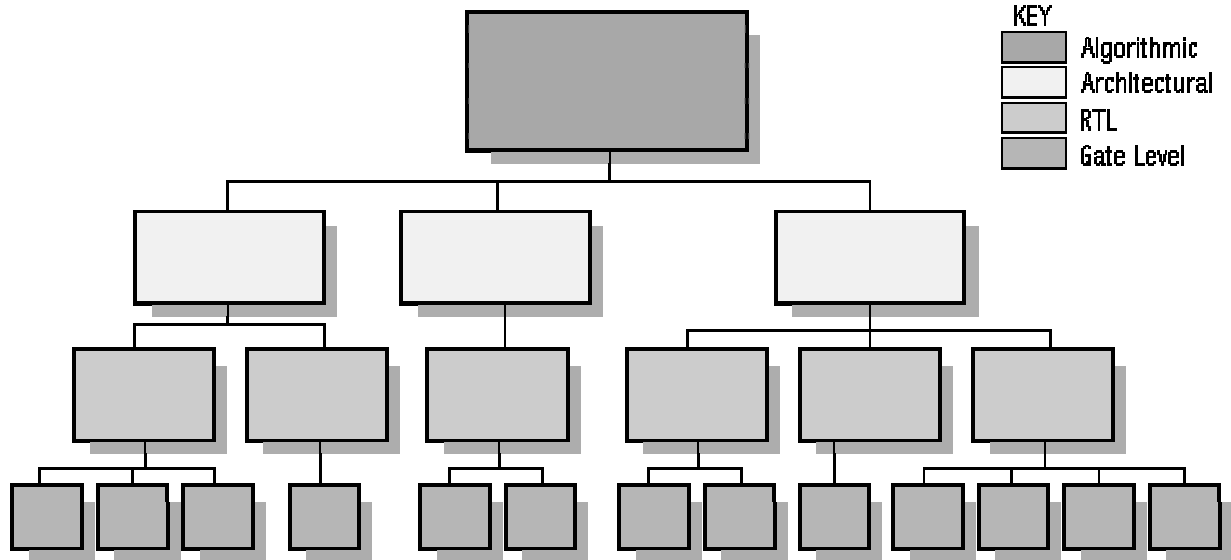


Figure 5. Top Down Design

Top-down design is the design method whereby high level functions are defined first, and the lower level implementation details are filled in later. A hardware design can be viewed as a hierarchical tree as shown in Figure 5. The top-level block represents the entire design, represented by algorithms. Each lower level block represents major functions of the design. Intermediate level blocks contain more detailed functionality. The middle levels consist of architectural and RTL descriptions. These RTL descriptions are eventually synthesized into the bottom row of gate level descriptions. This bottom level contains only gates and macrofunctions, which are vendor-supplied basic functions.

Top-down design is the preferred methodology for chip design for several reasons. First, chips often incorporate a large number of gates and a very high level of functionality. Attempting the design at a low level, without understanding the entire architecture, would be overwhelming. Scheduling the design would be a huge task. Top-down methodology forces the design to be partitioned into simpler pieces. This allows for better scheduling and resource allocation. For example, a junior engineer may be given a small piece of the chip to design while a senior engineer may be given several large pieces. Each engineer can complete his or her piece without needing to wait for the other team members to complete their pieces. If one piece gets delayed, the rest of the chip can still be completed on schedule. Even if only one engineer is responsible for the design, this partitioning makes a very complex task into a number of very simple tasks.

Also important is the fact that simulation is very much simplified using this design methodology. Simulation is an extremely important consideration in ASIC design since an ASIC cannot be blue-wired after production. For this reason, simulation must be done extensively before the ASIC is sent for fabrication. Using a top-down approach, first the behavioral description is simulated. This is done to determine that the overall function of the design is correct. Then the design is broken into smaller pieces of RTL code. Each RTL function block can be simulated independently, simply by substituting the RTL blocks into the equivalent behavioral block and rerunning the simulation. The results should be consistent. If not, you've found a bug. Because a simulation of the full RTL or gate level description can be very time consuming, mixing and matching RTL and behavioral descriptions greatly speeds up simulation time and, thus greatly speeds up bug detection and correction.

## Introduction to Verilog

This methodology also allows flexibility in the design. Sections can be removed and replaced with higher-performance or optimized designs without affecting other sections of the chip. This allows you to easily try out different designs by simply plugging in new hardware descriptions and simulating.

### Reusability

Reusability is a big advantage of HDLs. Code written in one HDL can be used in any system that supports that HDL. Schematics, on the other hand, are only useful in a particular schematic capture software tool. Even using the same tool, portability can be difficult if a module does not physically fit into the new design. A behavioral level HDL model, and most RTL level models can be easily used over and over again on multiple designs.

### Concurrency

Concurrency is an advantage that HDLs offer over normal software languages which can also be used to simulate hardware. With a normal software language, statements are executed sequentially. With an HDL, on the other hand, provisions have been added to support concurrent execution of statements. This is an absolute necessity since in a hardware design, many events occur simultaneously. For example, in a synchronous design, all flip-flops on a particular clock line must be evaluated simultaneously. While normal software languages can be used to model simultaneous events, it is up to the programmer to add the mechanism for handling this. With an HDL, the mechanism is built into the language.

### Timing

With schematic capture tools, when it comes time to simulate a design, the timing numbers are embedded in the netlist that is generated from the schematic. These timing numbers are based on parameters supplied by the vendors whose chips are being used. The user has some limited ability to change these numbers and some of the parameters. With HDLs, the timing numbers are explicitly stated in the code as shown in Example 2. Nothing is hidden from the user, and the user has complete control over these numbers. This makes it much easier to control and optimize the timing of your design.

```
module state_machine(sysclk, input1, input2, state3)
    . . .
    // Output delays are specified with the # symbol
    assign ps1 = #10 ~state1 & ~state2;
    assign ps2 = #5 state1 & input1 & input2;
    assign ps3 = #12 state2 | (state3 & input1);

    initial begin
        // initialize the state machine
        #13; // wait 13 time units before initializing
        state1 = 0;
        state2 = 0;
        state3 = 0;
    end

    always @(posedge sysclk) begin
        state1 <= #3 ps1; // output delay = 3 time units
        state2 <= #3 ps2;
        state3 <= #3 ps3;
    end
endmodule
```

Example 2. Code with explicit timing

## Optimization

HDLs are particularly useful for optimization. The most common method of designing ASICs and FPGAs involves writing an RTL level design and then using a software tool to “synthesize” your design. The synthesis process involves generating an equivalent gate level HDL description. Using various methods, the resulting description can be optimized to suit the particular target hardware. FPGAs, for example, typically have a very well-defined, large grain architecture. Mapping a gate level design to an FPGA would normally be difficult. However, you can write an RTL level model and synthesize it specifically for a particular FPGA. That same RTL description can be used in the future and synthesized for a particular ASIC technology.

## Standards

Both major HDLs, Verilog and VHDL, are public standards. Verilog was initially a privately developed language, which was released to the public in 1990 and is now maintained by the Open Verilog International (OVI), a consortium of companies that sustain and improve the language. It has been adopted by the Institute of Electrical and Electronic Engineers (IEEE) as the standard IEEE-STD-1364. VHDL is the VHSIC (Very High Speed Integrated Circuit) Hardware Description Language, and is supported by the U.S. Department of Defense. It was adopted by the IEEE as the standard IEEE-STD-1076. Because these languages are public standards, it ensures that a design written in the language can be accepted by every software tool that supports the language. It also means that you are not tied in to purchasing the tools of any one company, but can choose from tools offered by many vendors.

## Documentation

HDLs, being text-based, programming-type languages, lend themselves easily to documentation. This is not to say that the code is self-documenting. However, the code is relatively easy to read, and a code module shows much about the functionality, the architecture, and the timing of the hardware. In addition, as with any programming language, statements can be organized in ways that give more information about the design, and comments can be included which explain the various sections of code. It is up to the designer and the project leader to determine how to document the code, but the nature of HDLs certainly encourages this type of documentation.

## Large, Complex Designs

Large, complex designs require all of the above features. Because of this, HDLs are much better suited to large, complex designs than schematic capture or any other methods of design currently available.



### QUIZ 1: TRUE or FALSE?

1. HDLs can represent hardware at different levels of abstraction.
2. Timing numbers can be explicitly stated in HDL code.
3. Programming languages like C cannot handle concurrency.
4. HDLs have made schematic capture tools obsolete.
5. Many tools are available to optimize circuits designed with HDLs.
6. HDLs are better suited to describe large, complex designs than schematic capture tools.

## *Introduction to Verilog*

- 7. An HDL can model concurrently executing hardware functions.**
- 8. HDL code representing hardware functions can be reused in different designs.**
- 9. All HDLs are standards of the IEEE.**
- 10. According to IEEE-STD-1023, all HDLs must begin with the letter 'V'.**
- 11. Verilog is a better language than VHDL for the design of FPGAs.**
- 12. HDL code is easily readable, so documentation is not necessary.**