James M. Lee

VERILOG® Quickstart

3rd edition

A Practical Guide to Simulation and Synthesis in Verilog

CD ROM INCLUDED

VERILOG® QUICKSTART

A Practical Guide to Simulation and Synthesis in Verilog

Third Edition

THE KLUWER INTERNATIONAL SERIES
IN ENGINEERING AND COMPUTER SCIENCE

VERILOG® QUICKSTART

A Practical Guide to Simulation and Synthesis in Verilog

Third Edition

James M. Lee

Intrinsix Corp.

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

eBook ISBN: 0-306-47680-0
Print ISBN: 0-7923-7672-2

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF EXAMPLES

LIST OF TABLES

1 INTRODUCTION

Welcome to the world of Verilog! Once you read this book, you will join the ranks of the many successful engineers who use Verilog.

I have been using Verilog since 1986 and teaching Verilog since 1987. I have seen many different Verilog courses and many approaches to learning Verilog. This book generally follows the outline of the Verilog class that I teach at the University of California, Santa Cruz, Extension.

The Verilog language has been updated with the IEEE standardization in 1995, and now the update to the standard in 2001. In learning Verilog, it is important to current with the standards, however it should be noted that the Verilog language itself has changed little compared to the tools, workstations and techniques used by designers today vs. 1985. This third edition of Verilog Quickstart has been updated to reflect the current best practices in use today.

This book does not take a "cookie-cutter" approach to learning Verilog, nor is it a completely theoretical book. Instead, it describes some of the formal Verilog syntax and definitions, and shows practical uses. Once we cover most of the constructs of the language, the book examines how style affects the constructs you choose while modeling your design. This text is not intended as a complete and exhaustive reference on Verilog. For a comprehensive Verilog reference, I suggest one of the reference manuals from IEEE, Open Verilog International (OVI) or your tool vendor.

This book does not cover 100% of the Verilog language; it focuses on the 90% of Verilog that is used 90% of the time by designers who want to speed up their design cycle by verifying their designs in simulation and rapidly producing them through synthesis.

What is Verilog? In 1985, Automated Integrated Design Systems (renamed Gateway Design Automation in 1986) introduced a product named Verilog. It was the first logic simulator to seamlessly incorporate both a higher-level language and gate-level simulation. Before Verilog, there were many gate-level simulators and several higher-level language simulators, but there was no way to make them work together easily. About the same time, Gateway added the -XL algorithm to its product, creating Verilog-XL. It was the addition of this algorithm that put Verilog on the map.

The XL algorithm sped up gate simulation, thus making Verilog the fastest software gate-level simulator of the time. It was even faster than some of the then-current hardware accelerators. Today, there are several simulators that use the Verilog language.

Why were hardware description languages (HDLs) created? Verilog was invented as a simulation language. There were other simulation languages in use when Verilog was created, but Verilog was more complete and easier to use than its predecessors.

There is another key reason why HDLs were created. The United States Department of Defense (DOD) realized that they had a lot of electronics designed and built for them, and their products had a long life span. In fact, DOD might use equipment for upwards of twenty years. Over such periods semiconductor technology changed quite a bit. DOD realized they needed a technology-independent way to describe what was in the semiconductors they were receiving. Through a joint effort of the DOD and several companies, VHDL was created as a hardware description language to document DOD technology. VHDL and Verilog were developed at the same time, but independently.

Thus, two of the reasons HDLs were invented are simulation and documentation. Yet there is another common use for HDLs: Synthesis. Even before Verilog and VHDL were developed, the makers of programmable array logic (PAL) chips had created simple languages and tools (such as PALASM) to burn these chips. These languages accepted only simple equations and could create the correct bit pattern to make the chip reflect the functionality described in the language. Today, synthesis tools are much more robust and Verilog or VHDL may be used to describe many types of chips.

Why would you want to use an HDL? The simplest reason is to be more productive. An HDL makes you more productive in three ways:

1. *Simulation* By allowing you to simulate your design, you can see if the design works before you build it, which gives you a chance to try different ideas.

2. *Documentation* This feature lets you maintain and reuse your design more easily. Verilog's intrinsic hierarchical modularity enables you to easily reuse portions of your design as "intellectual property" or "macro-cells."

3. *Synthesis* You can design using the HDL, and let other tools do the tedious and detailed job of hooking up the gates.

This book focuses on the first two reasons because when you do these steps correctly, the third—synthesis—is an easily attainable goal. (Chapter 12 covers some synthesis specifics). I believe that if you truly understand Verilog, synthesis is not a problem. Furthermore, I think it is fine if not all your code is immediately synthesizable.

FRAMING VERILOG CONCEPTS

This section reviews some concepts you should already know. Some reflection on these concepts will help you learn Verilog by understanding how Verilog supports and opposes concepts you already understand.

The Design Abstraction Hierarchy

A circuit can be described at many levels. Figure 1-1 lists a few of them, from the abstract to the detailed. (Please note that many of these terms may mean different things to different people.)

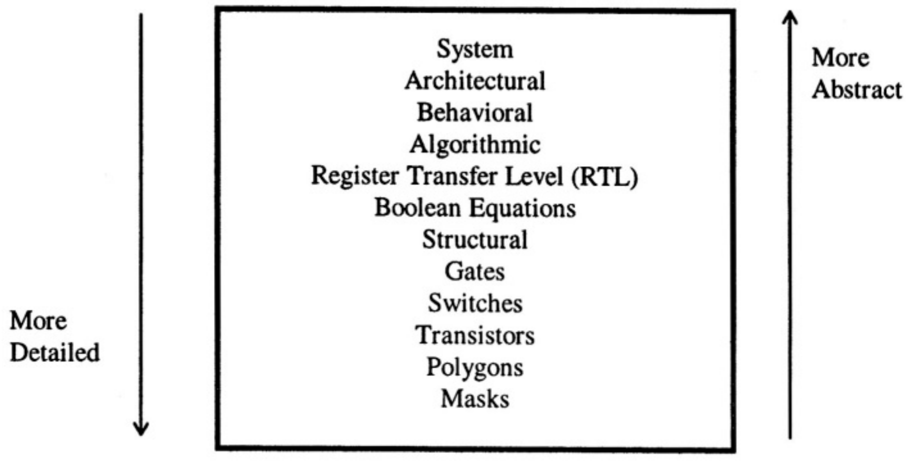

Figure 1-1 Design Abstraction Hierarchy

Which of these levels do you think Verilog can be used for? The answer to this question varies, but Verilog can definitely be used from the system level down to switches. However, Verilog is most commonly used from behavioral through gate levels. This book focuses on this commonly used range of design abstraction.

Types of Simulation

There are two types of simulation: Discrete (or event-driven), and continuous. Continuous simulation consists of a system of equations that represents the design problem. Simulators such as SPICE use continuous simulation.

However, Verilog (like most digital simulators) is an event-driven simulator. Simulation is considered event-driven when a change on an input causes a change on an output, which causes a further change on another input—In other words, event-driven simulation involves a chain of cause and effect.

Types of Languages

There are two types of HDLs: Loosely typed, and strongly typed. Without going into too much detail, some of the characteristics of each kind of HDL are described here.

A *loosely typed* language allows automatic type conversion, which lets you put the value 137 on an 8-bit bus. A *strongly typed* language would not permit you to do this because it would consider 137 to be an integer; an 8-bit bus is an array of 8 bits, and would not allow you to put an integer into an array.

Each type of language has its advantages; A loosely typed language will do what you mean most of the time. A strongly typed language will not allow you to make a mistake by combining the wrong types of objects. Strongly typed languages have conversion functions, so you could put the value 137 on an 8-bit bus by calling the integer to 8-bit array conversion function.

Verilog is a loosely typed language, whereas VHDL is a strongly typed language. But this fact does not make one language better than the other. If we look at the implicit type conversions as they take place, we gain an understanding of what Verilog is doing. Engineers know how to put the value 137 on an 8-bit bus. Implicitly, we convert the 137_{10} to 10001001_2 and put each of the bits on the bus in the correct order. For many engineers, a loosely typed language that does what they mean is just what they want.

Simulation versus Programming

Here is a not-so-simple question: If you assign *A* an initial value of 3, and *B* an initial value of 4 and execute the code below, what happens? What are the final values of *A* and B?

```
A = B
B = A
```

There are two possible answers. The final values are both 4, or they swap and *A* ends up with a final value of 4 and *B* ends up a final value of 3. How is this possible? We don't have enough information about the statements. We don't know whether they are sequential or concurrent. One key difference between a simulation language and a typical programming language is that in simulation we need a way to model both sequential and concurrent behavior. To do this, simulators introduce a notion of time.

HDL Learning Paradigms

There are two ways to learn an HDL: Start at the abstract and work toward the gate level, or start at the gate level. Example 1-1 shows an abstract example; Example 1-2 shows a gate-level example.

Example 1-1 Abstract Model of a Phone

```
/* Abstract behavioral system describing a telephone */
```

```
module office_phone;
parameter min_conversation=1, max_conversation=30,
          false=0, true=!false;
event ring, incoming_call, answer, make_call, busy;
reg off_hook;
integer seed, missed_calls;
initial begin
  seed=43;     // seed for call duration
  missed_calls=0;
  end

always @ incoming_call       // someone tries to call us
  if (! off_hook) -> ring;   // if not on the phone it rings
  else begin
    -> busy;  // else they get a busy signal
    $display($time," A caller got a busy signal");
    missed_calls = missed_calls + 1;
  end

always  @ring begin            // phone is ringing . . .
  $write($time," Ring Ring"); //do we want to answer it?
  if  ($random & 'b110) begin // yes we will answer it
    -> answer;
    off_hook = true;
    $display("  answered");
  end                  // no we do not want to answer
  else begin           //  this phone call
   missed_calls  = missed calls + 1;
   $display(" not answered missed calls =%d",
            missed_calls);
    end
  end

always  @make_call
 if (off_hook)
    $display($time," cannot make call phone in use");
  else
  begin
    $display($time,"  making call");
    off_hook = true;
  end
always wait(off_hook == true) begin //we are on the phone
                                     // wait the call duration
    #($dist_uniform(seed,         // a uniform distribution
       min_conversation,max_conversation))
      off_hook = false;
      $display($time," off phone");
    end
// might wait about 2 hours between making calls
always #($random & 255)  -> make_call;

// someone might call in within 4 hours
always #($random & 511)  -> incoming_call;
```

```
// Simulate two days worth of calls
initial #(60*24*2) $finish;

endmodule
```

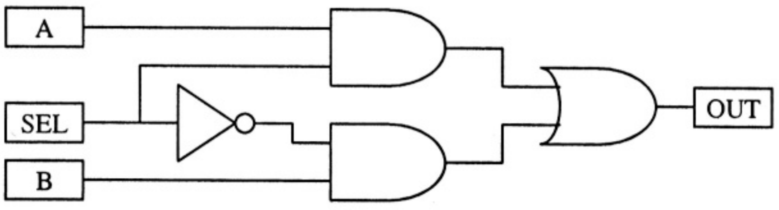

Figure 1-2 Gate-Level Model Mux Schematic

Example 1-2 Verilog for Gate-Level Mux

```
module mux(OUT, A, B, SEL);
output OUT;
input A,B,SEL;

not I5 (sel_n, SEL);

and I6 (sel_a, A, SEL);
and I7 (sel_b, sel_n, B);

or I4 (OUT, sel_a, sel_b);

endmodule
```

Most engineers have an easier time with the mux description in Example 1-2 than with that for the phone model in Example 1-1. Therefore, the approach of this book is to start with some gate-level modeling and work toward the constructs needed to create the phone model.

WHERE TO GET MORE INFORMATION

This book teaches you the Verilog language and some general techniques for modeling and debugging. Some information you want might be outside the scope of this book. Some of the other sources are simulator reference manuals; and the *comp.lang.verilog* usenet news group.

Reference Manuals

The IEEE standard for Verilog is 1364, and the IEEE standard may also be used as a reference. Verilog documentation falls into two categories: Reference manuals and user guides. Reference manuals provide details of a command or construct. User guides show you how to use a tool. This book falls in between: It teaches you the Verilog language, shows you how to model in Verilog, and describes the basics of using Verilog simulators. *Verilog 2001 - A guide to the new features of Verilog* by Stuart Sutherland (Kluwer Academic Publishers, ISBN 07923-7568-8) summarizes changes in the 2001 standard.

Usenet

The usenet is great source for information. There is a news group just for Verilog, called *comp.lang.verilog*. This news group sometimes has tips on modeling or news about Verilog tools. There is also a *comp.cad.cadence* news group that has news about Verilog-XL and other tools from Cadence Design Systems, Inc. The *comp.cad.synthesis* news group has news about synthesis tools for both Verilog and VHDL. As with most news groups there is a lot of banter, complaining, and philosophy mixed in with the occasional good tip. Perhaps the best single piece of information on the usenet regarding Verilog is the Frequently Asked Questions (FAQ) about Verilog. This document is updated and posted frequently and lists currently available tools and publications about Verilog.

2 INTRODUCTION TO THE VERILOG LANGUAGE

This chapter we looks at some of the formal definitions of the Verilog language: identifiers, white space, comments, numbers, text macros, modules, value set, and strengths.

IDENTIFIERS

Identifiers are the names Verilog uses for the objects in a design. Identifiers are the names you give your wires, gates, functions, and anything else you design or use. The basic rules for identifiers are as follows:

- May contain letters (a-z, A-Z), digits (0-9), underscores (_), and dollar signs ($).
- Must start with a letter or underscore.
- Are case sensitive (unless made case insensitive by a tool option).
- May be up to 1024 characters long.
- Other printable ASCII characters may be used in an *escaped identifier*.

What this means is that you are not limited to eight or sixteen characters to name things. You have over a thousand characters to use in the name of an identifier, so use names that make sense to you. Because names can start with a letter or underscore, and can contain letters and digits, you have quite a bit of flexibility.

Verilog does not have a standard notation for negated or active low signals. In this book, the standard for active low signals will be the name of the signal followed by *_n*. We use this notation to indicate active low signals because the notation is compatible with both Verilog and VHDL. (VHDL does not allow either leading or trailing underscores in names.) We make this recommendation to emphasize good habit from the beginning: Try to use naming conventions that will work both in Verilog and VHDL. It is likely that you will work with both VHDL and Verilog, so having one naming convention for negated signals is easier to remember.

Verilog is case sensitive, but VHDL and other tools are not. While you are establishing good habits for naming conventions consider using only one case. Using a single case for your identifiers will eliminate possible errors of disconnects when you type a wire name using different capitalization in Verilog, or a short when you move to a tool that does not consider case if you intend to have similar names with different capitalization.

Escaped Identifiers

Escaped identifiers allow you to use characters other than those noted above. The primary use of escaped identifiers is with automated tools and with translators that take a design from a format that allows names not legal in Verilog and converts the design and names to Verilog. Escaped identifiers follow these rules:

- Must start with a backslash (\).
- Must end with white space.

In Verilog the expression *carry/borrow* is not an identifier. It is an expression that says divide *carry* by *borrow.* If you want to use an identifier that would not normally be legal in Verilog, such as *carry/borrow* or *3sel,* you should form an escaped identifier. An escaped identifier is any sequence of printable characters that starts with a backslash (\) and ends with white space, so the identifiers *\3sel* and *\carry/borrow* are legal in Verilog.

WHITE SPACE

White space is the term used to describe the characters you use to space out your code to make it more readable. Verilog is not a white-space-sensitive language. Generally speaking, you can insert white space anywhere in your source code. The white-space characters are *space, tab,* and *return* (or *new line*). The only place that Verilog is sensitive to white space is inside quotes. You cannot have a new line inside quotes. For example, the code in Example 2-1, Example 2-2, and Example 2-3 is legal:

Example 2-1 Simple Hello Module

```
module hello1;
initial $display("Hello Verilog");
endmodule
```

Example 2-2 Hello Module without White Space

```
module hello2; initial $display("Hello Verilog");endmodule
```

Example 2-3 Hello Module with Extra White Space

```
module
hello3;
initial
$display(
"Hello Verilog"
                     )
                     ;
endmodule
```

The code in Example 2-4 is illegal in Verilog because there is a new line inside the quotes:

Example 2-4 Illegal Use of White Space

```
module hello4;
initial $display("
Hello Verilog
              ");
endmodule
```

COMMENTS

Verilog has two formats for comments: Single-line and block. *Single-line comments* are lines (or portions of lines) that begin with "//" and end at the end of a line. *Block comments* begin with "/*", end with "*/", and may span multiple lines. Verilog does not allow nested block comments.

Example 2-5 Comments

```
// this is a comment
/* this is also a comment
that spans multiple lines
*/
```

NUMBERS

If you have the number 10, do you know what base it is? Is it 10_2? 10_{10}? 10_{16}? How many bits are needed to hold it? In Verilog, the default is base ten, so the answer is 10_{10}.

In hardware modeling you might want to represent numbers of different bases and different bit widths. Why does it matter how many bits are used to hold the number? In simulation, the number of bits may matter for some operations. But for synthesis, the size of numbers becomes more important. You would not want synthesis to produce 32 bits of hardware where 8 bits would do, so it is a good habit to tell Verilog how many bits you want.

You just learned that you need to know the base (or radix) and the number of bits used to represent a number. You also need to know the value, so there are three pieces of information needed to form a number: The number of bits, the radix, and the value. Figure 2-1 shows the notation used in Verilog to fully represent a number.

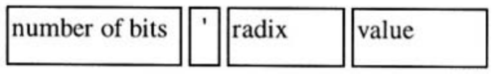

Figure 2-1 Number Format

Example 2-6 shows some fully specified numbers.

Example 2-6 Numbers

```
8'b10100101
16'habcd
```

The number of bits and radix are optional. The default radix is decimal. The default number of bits is implementation dependent, but is usually 32 bits. In this book we will assume the default number of bits is always 32.

The letters used for the radix are *b* for binary, *d* for decimal, *h* for hexadecimal, and *o* for octal. White space is allowed in numbers, so *1* '*b 1* is legal, but no space is allowed between the apostrophe and the radix mark. The radix specifiers are not case sensitive.

Table 2-1 Radix Specifiers

Radix Mark	Radix
'b 'B	Binary
'd 'D	Decimal (default)
'h 'H	Hexadecimal
'o 'O	Octal

TEXT MACROS

Verilog provides a text macro substitution facility. This is useful to define opcodes or other mnemonics you wish to use in your code. This is done with the grave accent key (backwards apostrophe) and the *define* keyword.

Example 2-7 Specifying a Text Macro

```
`define mycode 47
```

In Example 2-7, we defined the macro *mycode* to be 47. To implement the macro, we use the accent as shown in Example 2-8.

Example 2-8 Using a Text Macro

```
b = `mycode;
```

MODULES

The main building block in Verilog is the *module.* You create modules using the keywords *module* and *endmodule.* You build circuits in Verilog by interconnecting modules and the primitives within modules. Chapter 3 will introduce Verilog primitives. Thus far in the book you have seen three modules: the *phone* module, the *mux* module, and the *hello* module.

SEMICOLONS

Each Verilog statement ends with a semicolon. The only lines that do not need semicolons are those lines with keywords that end a statement themselves, such as *endmodule.*

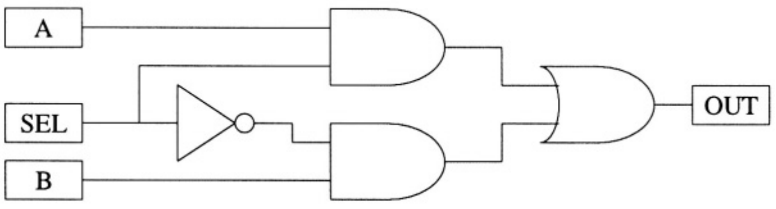

Figure 2-2 The Mux Example

Let's look at the mux example we used before and explain each line.

Example 2-9 Gate-Level Mux Verilog Code

```
1  module mux(OUT, A, B, SEL);
2  output OUT;
3  input A,B,SEL;
4
5  not I5 (sel_n, SEL) ;
6  and I6 (sel_a, A, SEL);
7  and I7 (sel_b, sel_n, B);
8
9  or I4 (OUT, sel_a, sel_b);
10 endmodule
```

- Line 1: module mux(OUT, A, B, SEL);
  This line declares the module name and its list of ports.

- Line 2: output OUT;
  This line tells Verilog the direction of the port *OUT*. *OUT* is the port name; *output* is a Verilog keyword used to declare port directions.
- Line 3: input A, B, SEL;
  This line tells Verilog the direction of the ports *A, B,* and *SEL*.
- Line 5: not I5 (sel_n, SEL);
  This line creates an instance of the built-in primitive *not*. The first port, *sel_n,* is the output, and the signal *SEL* is connected to the input of this NOT gate. *I5* is the instance name of this primitive.
- Lines 6 and 7: and I6 (sel_a, A, SEL); and I7 (sel_b, sel_n, B);
  These lines create instances of the built-in primitive AND gate. These gates have the instance names *I6* and *I7*.
- Line 9: or I4 (OUT, sel_a, sel_b);
  This line creates an instance of the built in primitive OR gate.
- Line 10: endmodule
  This line signals the end of the module.

VALUE SET

For logic simulation, we need more values than just zeroes and ones. You also need values to describe unknown values and high impedance (not driving). Verilog uses the values *x* to represent unknown and *z* to represent high impedance (not driving). Any bit in Verilog can have any of the values *0, 1, x,* or *z*.

STRENGTHS

Strengths are necessary in switch-level modeling. In Verilog, strengths are represented in a range from 0 (high impedance) to 7 (supply). There are four driver strengths: *supply, strong, pull,* and *weak*. There are three capacitive strengths: *large, medium,* and *small*. The capacitive strengths are used for storage nodes in switch-level circuits. This text is not focused on switch-level modeling and simulation. For more information on these topics, see an OVI or Cadence Verilog language reference. The strengths and values combine internally in Verilog to create a set of 120 possible states for a signal in Verilog.

Where do these 120 possible states come from? Consider the circuit in Figure 2-3.

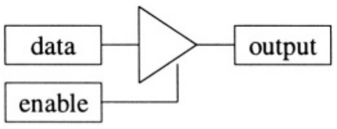

Figure 2-3 Three-State Buffer

If *data* is *1* and *enable* is *1,* what is *output*? (Answer: *1*.)

If *data* is *0* and *enable* is *1*, what is *output*? (Answer: *0*.)

If *data* is *1* and *enable* is *0*, what is *output*? (Answer: *z*.)

If *data* is *1* and *enable* is *x*, what is *output*? (Answer: *1* or *z*.)

How is this last answer possible? We can all agree that the answer to the last question is not *0.* Some of you might have chosen *x.* Consider the circuit in Figure 2-4.

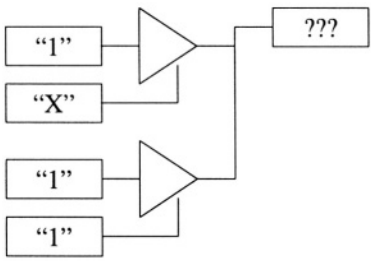

Figure 2-4 Two Three-State Buffers

What is *output*? The final output should be *1*, but if the top gate's output were *x,* the result would be *x.* So the 120 other states are used to express ambiguities and make the simulation more optimistic.

Numbers, Values, and Unknowns

Is *x* a number? How do you set a signal to the value *unknown*? *x* by itself is an identifier. If we want the value *x* we need to make it into a number. To make a number we need a number of bits, a radix, and a value. Therefore *1'bx* is a number with the value *x*.

There are a few more rules about numbers and values. Verilog does not sign extend values. It extends all values with zero, except those with *x* or *z* in the most significant place. Numbers with *x* and *z* in the most significant place extend with *x* and *z*, respectively.

Table 2-2 shows examples of numbers and their binary representation.

Table 2-2 Numbers and Their Values

Number	Value	Number	Value
8'b0	00000000	8'b1	00000001
8'bx	xxxxxxxx	8'hz1	zzzz0001
8'b1x	0000001x	8'bx1	xxxxxxx1
8'b0x	0000000x	8'bx0	xxxxxxx0
8'hx	xxxxxxxx	8'hz	zzzzzzzz
8'hzx	zzzzxxxx	8'h0z	0000zzzz

3 STRUCTURAL MODELING

One of the easiest ways to model designs in Verilog is with structural modeling, which is simply connecting devices. Even complex models can exhibit elements of structural modeling. Whether you are connecting a cache to a processor, or an inverter to an AND gate, you interconnect the models the same way. This chapter shows you how to connect your models. By the end of this chapter, you should be able to model and simulate simple circuits.

Structural modeling is often automated by capturing schematics and writing out netlists. Using structural modeling, you can model many circuits.

PRIMITIVES

Verilog has a set of twenty-six built-in primitives. These primitives represent built-in gates and switches. These built-in primitives are listed in Table 3-1

Table 3-1 Verilog Primitives

Logic Gates		
and	or	xor
nand	nor	xnor
Buffers		
buf	bufif0	bufif1
not	notif0	notif1
pulldown	pullup	
Transistors		
nmos	pmos	cmos
rnmos	rpmos	rcmos
tran	tranif0	tranif1
rtran	rtranif0	rtranif1

The primitives *and, nand, or, nor, xor,* and *xnor* represent simple logic functions with one or more inputs and one output. Buffers, inverters, and three-state buffers/inverters are represented by *buf, not, bufif1, bufif0, notif1,* and *notif0.* The *pullup* and *pulldown* primitives have a single output and no inputs, and are used to pull up or pull down a net. MOS-level unidirectional and bidirectional switches are represented by the remaining primitives.

Appendix A explains each of the primitives in more detail and provides a truth table for each.

Note that there are no built-in muxes or flip-flops. This is because there are too many different types of these to include them all, so Verilog provides user-defined primitives to model these. User-defined primitives are explained in Chapter 13.

PORTS

Ports in Primitives

The Verilog terminology for a connection or "pin" is *port.* All the built-in primitives (gates) have ports. The *pullup* and *pulldown* primitives have only one port. The first port of each of the built-in primitives (gates) is the output. This allows you, for example, to use the same *and* primitive to represent a 2-input or 4-

input AND gate. The only built-in primitives that can have more than one output port are the *buf* and *not* primitives, which can have many outputs, with the last terminal being the input.

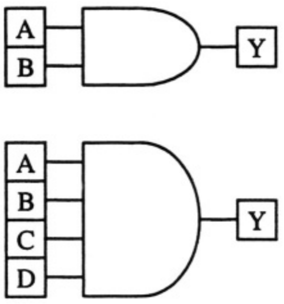

Figure 3-1 AND Gate Primitives

Example 3-1 Verilog Code for the 2-Input and 4-Input AND Gates

```
and ( Y, A, B ) ;
and ( Y, A, B, C, D ) ;
```

All of the multiple-input primitives may have as many inputs as you need, so you could have a 100-input AND gate if you needed it. However, not all Verilog clone simulators support this.

Ports in Modules

Modules can have ports. Two of the modules you have seen thus far (the *phone* and *hello* modules) did not have ports, but the *mux* module did. In general, if you are modeling a self-contained system, you will not have ports. But if you are modeling something that needs to be connected to something else, you will need ports to make those connections.

Verilog supports three port directions: *input, output,* and *inout* (the keyword for bi-directional ports). In Verilog, you must declare the ports in two places: First, as part of the port list in the module. Second, for the direction and size of all the module's ports using the *input, output,* and *inout* keywords. The 2001 standard allows you to combine the portlist and direction into a single declaration, as shown in Chapter 19.

INSTANCES

The word *instance* is not a Verilog keyword. Rather, it is the word we use to mean *make a copy of,* or *use.* When you use a built-in primitive, you make an instance or copy of the built-in gate and list its connections. When you make an instance of a built-in gate, you have the option to give it a unique name called an *instance name.* When you make an instance of a module you are required to give it an instance name.

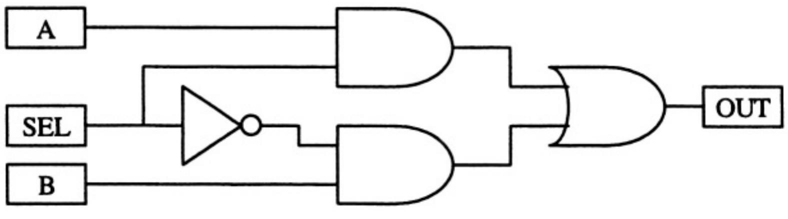

Figure 3-2 Gate-Level Model Mux Schematic

Example 3-2 Verilog for Gate-level Mux

```
module mux(OUT, A, B, SEL);
output OUT;
input A,B,SEL;

not I5 (sel_n, SEL);

and I6 (sel_a, A, SEL);
and I7 (sel_b, sel_n, B);

or I4 (OUT, sel_a, sel_b);

endmodule
```

As you can see from the mux example, there are four gates in the schematic. The Verilog code shows four instances, each one corresponding to a gate in the schematic.

HIERARCHY

We can connect modules inside other modules, creating *hierarchy.* For example, if you want to have a 2-bit mux, you can create it by using two 1-bit muxes from Example 3-2, as shown in Figure 3-3.

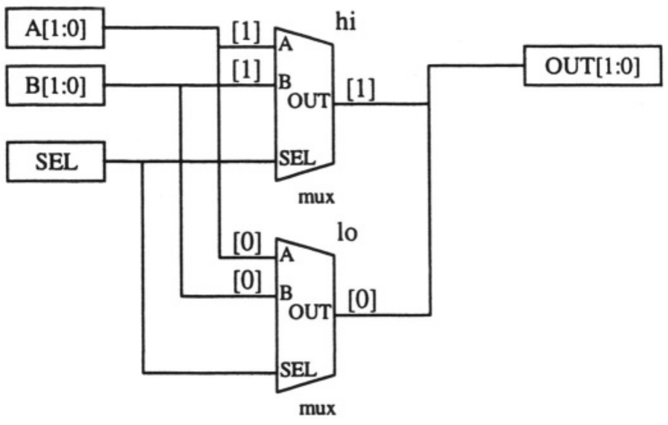

Figure 3-3 Connecting Two Muxes

Example 3-3 Hierarchical 2-Bit Mux

```
module mux2(OUT, A, B, SEL);
output [1:0] OUT;
input [1:0] A,B;
input SEL;

mux hi (OUT[1], A[1], B[1], SEL);
mux lo (OUT[0], A[0], B[0], SEL);

endmodule
```

You can make an even more hierarchical 4-bit mux by connecting two of these 2-bit muxes, as shown in Figure 3-4.

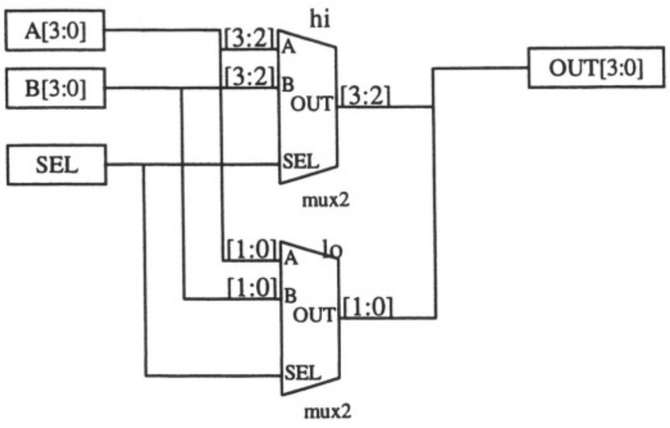

Figure 3-4 Hierarchical 4-Bit Mux

The Verilog for connecting two 2-bit muxes as shown in Figure 3-4 is shown in Example 3-4.

Example 3-4 Hierarchical 4-Bit Mux

```
module mux4(OUT, A, B, SEL);
output [3:0] OUT;
input [3:0] A,B;
input SEL;

mux2 hi (OUT[3:2], A[3:2], B[3:2], SEL);
mux2 lo (OUT[1:0], A[1:0], B[1:0], SEL);

endmodule
```

You can use this technique of making primitive instances and module instances to model most circuits. This technique is sometimes called netlist modeling or structural modeling. In more complex circuits, you can still use this technique to connect modules that contain constructs other than instances.

HIERARCHICAL NAMES

Figure 3-5, shows the 4-bit mux hierarchically expanded. There are four copies of the mux module. Each of the mux modules contains four gate instances, four ports, and three internal wires. Each of the four copies of the module has a unique hierarchical name. With hierarchical names, we can distinguish the copies so it is possible to uniquely identify each gate, port, and wire.

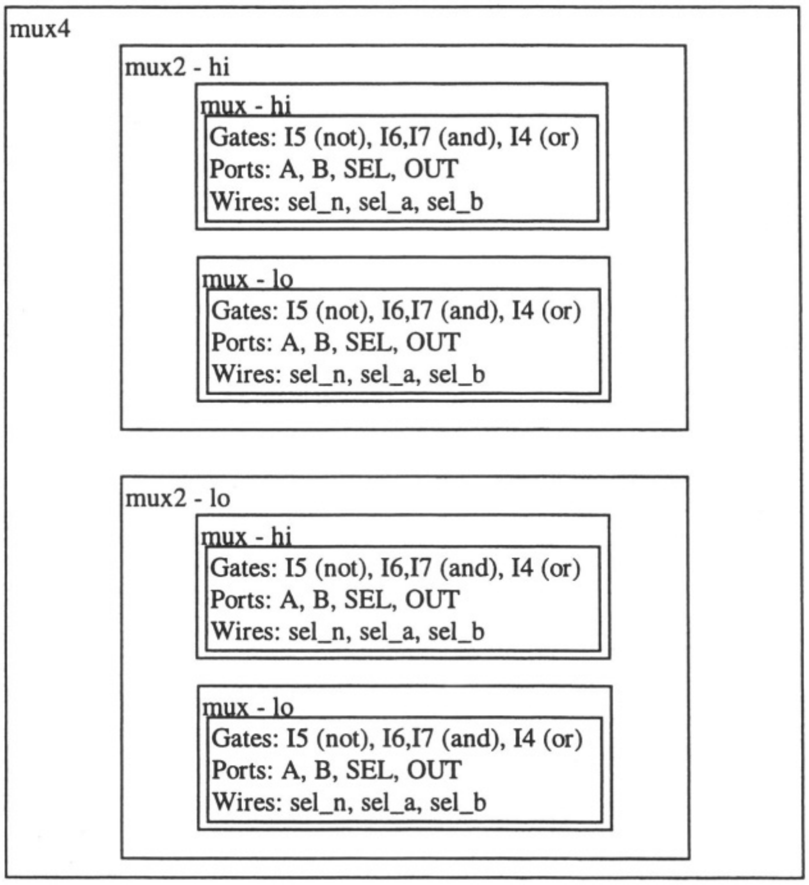
mux4
mux2 - hi
mux - hi
Gates: I5 (not), I6,I7 (and), I4 (or)
Ports: A, B, SEL, OUT
Wires: sel_n, sel_a, sel_b
mux - lo
Gates: I5 (not), I6,I7 (and), I4 (or)
Ports: A, B, SEL, OUT
Wires: sel_n, sel_a, sel_b
mux2 - lo
mux - hi
Gates: I5 (not), I6,I7 (and), I4 (or)
Ports: A, B, SEL, OUT
Wires: sel_n, sel_a, sel_b
mux - lo
Gates: I5 (not), I6,I7 (and), I4 (or)
Ports: A, B, SEL, OUT
Wires: sel_n, sel_a, sel_b

Figure 3-5 Mux4 Hierarchy Expanded

A hierarchical name can reference any object in a simulation. Hierarchical names have two forms: A downward path from the current module, or a name that starts at a top-level module and provides a complete path. Verilog uses the dot (.) to separate the elements in the path of a hierarchical name. Example 3-5 shows some hierarchical names.

Example 3-5 Hierarchical Names

```
lo.lo.sel_n
mux4.lo.lo.sel_n
```

Connect by Name

All of the hierarchy built by module instances in Example 3-3 and Example 3-4 are built by matching the port declaration order to the used to create the connections. This type of instantiation is called *connect by order* since the port order must be known and matched. Verilog also supports a *connect by name* syntax, where the port order does not need to be known, but the port names must be known. The *connect by name* syntax uses the hierarchical name for the ports to make the connects. Example 3-6 shows Example 3-4 re-written to use the *connect by name* syntax.

Example 3-6 Mux Connected by Name

```
module mux4cbn(OUT, A, B, SEL);
output [3:0] OUT;
input [3:0] A, B;
input SEL;

mux2 hi( .A(A[3:2]), .B(B[3:2]), .SEL(SEL), .OUT(OUT[3:2]) );
mux2 lo( .A(A[1:0]), .B(B[1:0]), .OUT(OUT[1:0]), .SEL(SEL) );

endmodule
```

Figure 3-6 clarifies the syntax for connect by name. The net to be connected to the port is in the parenthesis.

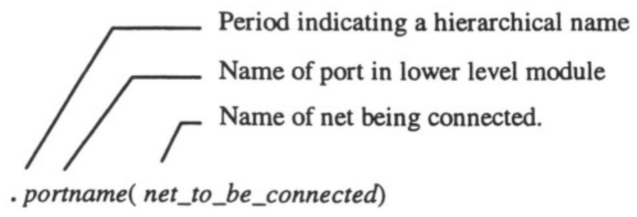
Period indicating a hierarchical name
Name of port in lower level module
Name of net being connected.
. portname(net_to_be_connected)

Figure 3-6 Syntax for Connect By Name

Top-Level Modules

When the Verilog simulator finishes compiling your modules, the first thing it reports is which module or modules are "Highest-level modules." Highest-level or top modules are modules that no other module has made an instance of. These non-referenced modules are considered to be at the top of the hierarchy. Usually there is only one top-level module, the test bench for your circuit. The test bench module is used to provide inputs or stimulus to a design. Chapter 18 discusses test benches.

Each instantiated module has a unique instance name. Since a top-level module is not instantiated, it has no instance name. Verilog automatically assigns an instance name to top-level modules. The instance name of a top-level module is the name of the module itself.

For example, because there is no test bench module for the mux4, it will be a top-level module.

You Are Now Ready to Run Your First Simulations

We will take a break now and do some exercises using Verilog.

Exercise 1 The Hello *Simulation*

As a simple test to see if your Verilog simulator works correctly, enter the code in Example 3-7 into a file called *hello.v*.

Example 3-7 Hello Verilog

```
module hello;
initial $display("Hello Verilog");
endmodule
```

To run a simulation with most Verilog simulators, you simply type the name of the simulator and the name(s) of the Verilog file(s).

To run the *hello* simulation, type *verilog hello.v.* You should get the results as shown in Example 3-7 Results.

Example 3-7 Results

```
Compiling source file "hello.v"
Highest level modules:
hello

Hello Verilog
```

Verify that you are able to enter a simple Verilog model and run the simulator. If you have trouble running the simulator, consult the documentation for the simulator. Once the *hello* simulation is complete, you are ready to move on to a more challenging exercise.

Exercise 2 The 8-Bit Hierarchical Adder

Now that you know that your Verilog simulator works, try to create some modules. Use the schematic in Figure 3-7 to create a module *adder* using the built-in primitives listed in Table 3-1.

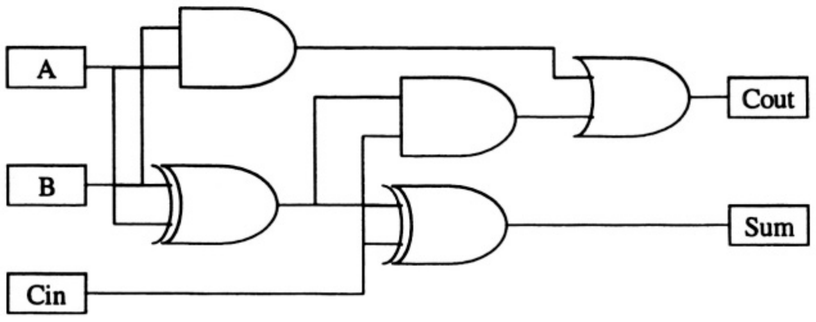

Figure 3-7 *Adder* Schematic

When you write the Verilog for the adder you will need to decide on names for the three internal wires. You can use any names you prefer, but your wire names must be legal identifiers. Some suggested wire names are *half_carry_ab,* for the output of the top AND gate, *half_sum* for the output of the first EXCLUSIVE-OR, and *half_carry_cin* for the output of the second AND gate.

Next, connect two of your adders to create an *adder2* as shown in Figure 3-8. You will need an extra signal, *internal_carry,* to connect the *carry_out* of your low-order adder to the *carry_in* of your high-order adder. You will also need instance names for the two adder modules. The simplest instance names to use are *hi* and *lo.*

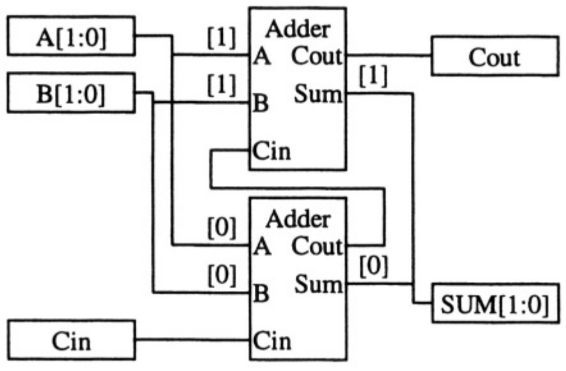

Figure 3-8 *Adder2* Schematic

Connect two of the *adder2s* together to form a 4-bit adder as shown in Figure 3-9.

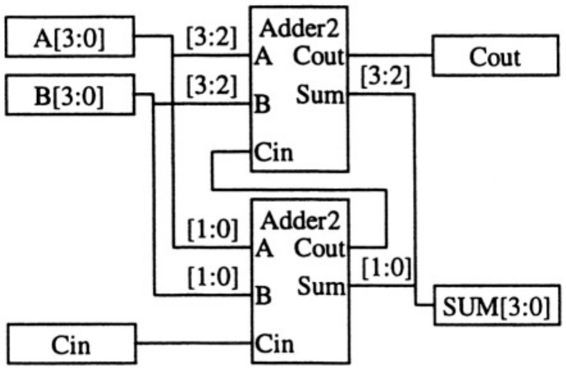

Figure 3-9 *Adder4* Schematic

Connect two of the *adder4s* together to form a 8-bit adder as shown in Figure 3-10.

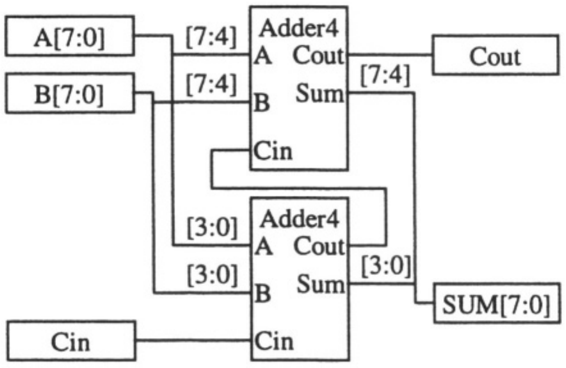

Figure 3-10 *Adder8* Schematic

Finally, simulate your 8-bit adder with the provided test bench *test_adder.v* shown in Example 3-8. You should get the results as shown in Example 3-8 Results.

Example 3-8 Adder Test Module

```
module  test_adder;
reg [7:0] a,b;
reg carry_in ;
wire  [7:0]  sum;
wire  carry_out;

adder8 dut(carry_out,  sum,  a,b,  carry_in);

initial begin
    a = 0; b = 0; carry_in = 0;
    # 100 if  (sum !== 0)  begin
        $display("sum is wrong");
        $finish;
       end

    a = 1; b = 0;  carry_in = 0;
    # 100 if  (sum !== 1)  begin
        $display("sum is wrong");
        $finish;
       end

    a = 0; b = 0; carry_in = 1 ;
    # 100 if  (sum !== 1)  begin
        $display("sum is wrong");
        $finish;
       end

    a = 5; b = 6; carry_in = 1;
    # 100 if  (sum !== 12)  begin
        $display("sum is wrong");
        $finish;
       end

    a = 200; b = 55; carry_in = 1;
    # 100 if  (sum !== 0)  begin
        $display("sum is wrong");
        $finish;
       end

    a =18; b = 200; carry_in = 1;
    # 100 if  (sum !== 219)  begin
        $display("sum is wrong");
        $finish;
       end
    $finish ;

  end
endmodule
```

Note that the results of this simulation, shown in Example 3-8 Results, did not give us any meaningful results other than the fact that it finished at time 600. In this case, this is the correct result for the simulation because if the simulation had finished at a time before 600, there was an error in one of the adders.

Example 3-8 Results, Output from Exercise 2

```
Compiling source file "test_adder.v"
Compiling source file "adder8.v"
Highest level modules:
test_adder

L51 "test_adder.v": $finish at simulation time 600
```

Chapter 4 examines the parts of Verilog used for modeling test inputs and collecting results. It explains what the lines in *test_adder.v* mean and how to improve this test bench to print out some results.

4 STARTING PROCEDURAL MODELING

Using the structural modeling technique from Chapter 3, you can model many different types of circuits. One of the reasons that Verilog gained popularity was the ease with which it allowed mixing behavioral modeling techniques with structural modeling. Before Verilog, there were both structural modeling and simulation tools, and there were even behavioral languages and tools, but no one tool combined both behavioral and structural modeling.

Before the creation of Verilog, you needed to know three languages: One for the netlist (as in the structural modeling covered in Chapter 3); one to create the stimulus for your circuit; and one to process the output from the simulation. Using Verilog is more efficient than older simulators: You only need to learn one language. In Verilog, you use the same language for structural modeling, behavioral modeling, creating the stimulus, and analyzing results.

Hopefully you have taken the time to run the simple hierarchical 8-bit adder at the end of Chapter 3. You will note that the results of that simulation give no indication of the inputs and outputs of the circuit, so it is difficult to tell if the circuit really works correctly. Therefore, the first behavioral aspects of the Verilog language we will look at are the parts of the language you use to print results.

STARTING PLACES FOR BLOCKS OF PROCEDURAL CODE

Procedural Verilog code is like programming in a computer language—with one large exception: Procedural Verilog code adds a concept of time. With a programming language, code is started at a particular location, for example, at the first line or main function. In Verilog, code starts running in one of two places: at the *initial* statement and at the *always* statement. Do not assume that you can have only these two statements in your code. You can have as many *initial* statements and *always* statements as you want in your simulation or module. However, if all the code is started at the *initial* and *always* statements, how can you know the order in which the statements will run? This is where the model of time comes into effect.

The *initial* Keyword

Verilog interprets the *initial* keyword to mean "start here at time 0." Do not let the keyword throw you off track. Sometimes people think the *initial* keyword is used only for *initialization.* The keyword *initial* is used not only for initialization, but also as a place for starting code. Look at the test bench for the 8-bit adder. It has an *initial* statement with several statements to apply the stimulus and check the results from the adder.

Verilog starts all *initial* statements at time 0. The time at which the statements finish depends on the code in the *initial* block. When the statements finish, the *initial* block is done. However, it is possible to have an *initial* block that never finishes.

Example 4-1 An *initial* Block

```
initial $display("Hello Verilog");
```

The most simple *initial* block was shown in the *hello* simulation. Example 4-1 repeats this simple *initial* block that starts at time 0 and prints out the message "Hello Verilog." This *initial* block is then finished. If you want to do more than one operation in an *initial* block, you will need to use a *begin-end* block or *fork-join* block, covered later in this chapter.

The *always* Keyword

The *always* keyword is similar in behavior to the *initial* keyword. Verilog also begins to run *always* statements at time 0. The difference between *initial* and *always* statements is what happens when the statements finish running. The *always* block starts again when it finishes. An *always* block is like an *initial* block with an infinite loop. If we change the *hello* simulation from an *initial* to an *always* as shown in Example 4-2, the simulation would continue to print until we kill the simulation. Simulation time would remain stuck at time *0*.

Example 4-2 An *always* Block

```
always $display("Hello Verilog");
```

Remember that *always* statements can create infinite loops. Some infinite loops are useful, but we will consider it an error to have a zero-delay *always* loop.

Just as the *initial* statement should be remembered as “start here at time 0,” remember the *always* statement as “start here at time 0, and when done, start again.”

Delays

Every statement in Verilog may have a delay before it is run. If we have three *initial* statements as in the module in Example 4-3, we know they will all start at time 0. But they must run in some order: Which one will run first? There is really no way to tell. Not only is there no way to tell, if you run this module on another simulator, that simulator might run this module in a different order.

Example 4-3 Three *Initial* Statements

```
module three_initial;

initial $display("Initial Statement 1");

initial $display("Initial Statement 2");

initial $display("Initial Statement 3");

endmodule
```

The model in Example 4-3 creates a race condition at time 0. If it is important to have the statements run in a particular order, you can introduce delays to control the order in which the statements are executed.

Example 4-4 Three *Initial* Statements with Delay

```
module three_initial_with_delay;

initial #1 $display("Initial Statement 1");

initial $display("Initial Statement 2");

initial #2 $display("Initial Statement 3");

endmodule
```

In Example 4-4, all of the *initial* statements are started at time 0, but now the first one waits one time unit before it continues. In the meantime, the second statement (which also started at time 0 and has no delay) prints out the message "Initial Statement 2" and is finished. The third statement has a delay of two time units, so it waits even longer. At time 1, the first statement finishes running, and then at time 2 the third statement finishes running, and the whole simulation is done.

The " # " symbol is the delay operator in Verilog. The best way to think of # is to remember that # means "wait for some amount of time."

begin-end Blocks

Initial and *always* can have only one statement. However, you will often need more than a single statement in your design. The *begin-end* block allows a set of sequential statements to follow an *initial* or *always* statement. Example 4-5 shows a simple *begin-end* block.

Example 4-5 Simple *begin-end* Block

```
module initial_begin;

initial
      begin
            $display("Statement 1");
            $display("Statement 2");
            $display("Statement 3");
      end
endmodule
```

The statements in the *begin-end* block are sequential so we know the statements will execute in the order you would expect. The *begin-end* block in Example 4-5 has no delays in it, so this *initial* statement still finishes at time 0. In *begin-end* blocks, delays are additive. Example 4-6 shows what happens when we change the block and introduce delays.

Example 4-6 *begin-end* Block with Delay

```
module initial_begin_with_delay;

initial
       begin
              #1 $display("Statement 1");
              $display("Statement 2");
              #2 $display("Statement 3") ;
       end

endmodule
```

Like all *initial* statements, the *initial* statement in Example 4-6 starts at time 0. The *begin-end* block starts, and the first statement has a delay, so this block waits until time 1. At time 1, the delay expires, and "Statement 1" is printed. The next statement has no delay, so at time 1, "Statement 2" is also printed. The third statement has a delay of two time units. The delay is encountered at time 1. Therefore, the simulator waits until time 3 before continuing (1 + 2 = 3). At time 3, "Statement 3" is printed, the *begin-end* block will be done, and the *initial* block will finish.

To gain a better understanding of the sequence statements, consider the situation of two *begin-end* blocks that are started from separate *initial* statements, as shown in Example 4-7.

Example 4-7 Multiple *begin-end* Blocks

```
module initial_two_begin;

initial
       begin
              #1 $display("Statement 1");
              $display("Statement 2");
              #2 $display("Statement 3");
       end

initial
       begin
              $display("Block 2 Statement 1");
              #2 $display("Block 2 Statement 2");
              #2 $display("Block 2 Statement 3");
       end

endmodule
```

Example 4-7 Results shows the output from the simulation of the model in Example 4-7.

Example 4-7 Results 1

```
Block 2 Statement 1
Statement 1
Statement 2
Block 2 Statement 2
Statement 3
Block 2 Statement 3
```

Does Example 4-7 Results 1 show the results you expected? Why did we get these results? The best way to understand the sequence of statements in Example 4-7 is to trace the sequence of events in the simulator.

Both *initial* blocks start at the same time. However, the first *initial* block encounters a delay, so the first event that occurs is the *$display* in the second block. To give you a step-by-step description of what happens, Verilog has a trace mode. Example 4-7 Results 2 shows the results of running the simulation with the trace mode.

Example 4-7 Results 2

```
L3  "i2b.v"  (i2b): INITIAL
L4  "i2b.v"  (i2b): BEGIN
L4  "i2b.v"  (i2b): #1
L10  "i2b.v"  (i2b): INITIAL
L11  "i2b.v"  (i2b): BEGIN
L12  "i2b.v"  (i2b): $display ("Block 2 Statement 1")
Block 2 Statement 1
L11  "i2b.v"  (i2b): #2
SIMULATION TIME IS 1
L4  "i2b.v"  (i2b): #1 >>> CONTINUE
L5  "i2b.v"  (i2b): $display ("Statement 1")
Statement 1
L6  "i2b.v"  (i2b): $display ("Statement 2")
Statement 2
L4  "i2b.v"  (i2b): #2
SIMULATION TIME IS 2
L11  "i2b.v"  (12b): #2 >>> CONTINUE
L13  "i2b.v"  U2b) : $display ("Block 2 Statement 2")
Block 2 Statement 2
L11  "i2b.v"  (i2b): #2
SIMULATION TIME IS 3
L4  "i2b.v"  (i2b): #2 >>> CONTINUE
L7  "i2b.v"  (i2b): $display ("Statement 3")
Statement 3
L8  "i2b.v"  (i2b): END
```

```
SIMULATION TIME IS 4
L11 "i2b.v"  (i2b): #2 >>> CONTINUE
L14 "i2b.v"  (i2b): $display ("Block 2 Statement 3")
Block 2 Statement 3
L15 "i2b.v"  (i2b): END
```

You activate the Verilog trace option either with the *-t* command line option, or by using the *$settrace* system command. However, tracing large designs is not very practical. As you can see, the trace option produces extensive output even for a simple test case.

fork-join Blocks

The *fork-join* block is similar to the *begin-end* block: It is also used to group statements. In *begin-end* blocks, the statements are sequential, and the delays are additive. In *fork-join* blocks, the statements are concurrent, and the delays are independent, or absolute from the time the *fork-join* block starts.

If the code in Example 4-7 is changed by substituting two *fork-join* blocks for *begin-end* blocks, the behavior will be different, as shown in Example 4-8.

Example 4-8 *fork-join* Blocks

```
module i2f;

initial
       fork
             #1 $display("Statement 1");
             $display("Statement 2");
             #2 $display("Statement 3");
       join

initial
       fork
             $display("Block 2 Statement 1");
             #2 $display("Block 2 Statement 2");
             #2 $display("Block 2 Statement 3");
       join

endmodule
```

Both *initial* statements still start at time 0, and each *initial* statement has a *fork-join* that starts at time 0. The first statement to run in the first block is the “Statement 2” line because it has no delay. The “Block 2 Statement 1” also runs at time 0. In this example there are two statements to be run at time 0, and three statements to be run at time three. The results shown are only one possible result. There is no guarantee of order when multiple statements need to be run at the same time. Sample results of this run are shown in Example 4-8 Results 1.

Example 4-8 Results 1

```
Statement 2
Block 2 Statement 1
Statement 1
Statement 3
Block 2 Statement 2
Block 2 Statement 3
```

Example 4-8 Results 2 shows the sample results with the *trace* option.

Example 4-8 Results 2

```
L3 "i2f.v" (i2f): INITIAL
L4 "i2f.v" (i2f): FORK
L4 "i2f.v" (i2f): #1
L6 "i2f.v" (i2f): $display ("Statement 2")
Statement 2
L4 "i2f.v" (i2f): #2
L10 "i2f.v" (i2f): INITIAL
L11 "i2f.v" (i2f): FORK
L12 "i2f.v" (i2f): $display ("Block 2 Statement 1")
Block 2 Statement 1
L11 "i2f.v" (i2f): #2
L11 "i2f.v" (i2f): #2
SIMULATION TIME IS 1
L4 "i2f.v" (i2f): #1 >>> CONTINUE
L5 "i2f.v" (i2f): $display ("Statement 1")
Statement 1
SIMULATION TIME IS 2
L4 "i2f.v" (i2f): #2 >>> CONTINUE
L7 "i2f.v" (i2f): $display ("Statement 3")
Statement 3
L8 "i2f.v" (i2f): JOIN
L11 "i2f.v" (i2f): t2 >>> CONTINUE
L13 "i2f.v" (i2f): $display ("Block 2 Statement 2")
Block 2 Statement 2
L11 "i2f.v" (i2f): #2 >>> CONTINUE
L14 "i2f.v" (i2f): $display ("Block 2 Statement 3")
Block 2 Statement 3
L15 "i2f.v" (i2f): JOIN
```

As you can see from Example 4-8 Results 2, a *fork-join* block finishes when its last statement finishes.

Note that *fork-join* and *begin-end* blocks are themselves single statements. You can nest *fork-join* and *begin-end* blocks. You can also nest *begin-end* blocks within *begin-end* blocks; *fork-join* blocks within *fork-join* blocks; and *begin-end* blocks within *fork-join* blocks.

Although there are four possible ways to nest these blocks, two of the combinations are generally impractical. Nesting *begin-end* blocks within *begin-end* blocks has no benefit because all the statements are sequential already. When *begin-end* blocks are nested, it is usually for control flow, such as in the adder test module at the end of Chapter 3, in which the inner *begin-end* blocks contain the statements controlled by the *if*. Nesting a *fork-join* block in *a fork-join* block is impractical unless there is a delay outside the inner *fork-join* block.

Example 4-9 contains two *initial* blocks and an *always* block. The first *initial* block has only one statement in it: a delay of 50 time units and a *$finish* keyword. The first *initial* statement is necessary to make the simulation terminate; without this statement, the *always* block would keep the simulation running forever.

Example 4-9 Combining *begin-end* and *fork-join* Blocks

```
module befjia;
initial #50 $finish;
initial begin
    #1 $display(" b 1");
    #1 fork
        #1 $display("  b 1 f 1");
        $display("  b 1 f 2") ;
        #5 $display("  b 1 f 3");
        #2 begin
            $display("  b 1 f 4 b 1");
            #1 $display("  b 1 f 4 b 2");
            $display("  b 1 f 4 b 3");
        end
    join
    $display(" b 2");
end

always fork
    # 3 $display(" f 1");
    begin
        #1 $display("   f 2 b 1");
        #2 $display("   f 2 b 2");
        #3 $display("   f 2 b 3");
    end

    begin
        #10 $display("    f 3 b 1");
        #9 $display("   f 3 b ");
        #8 $display("   f 3 b3") ;
    end

    # 5 fork
        #1 $display("    f 4 f 1");
        #2 $display("    f 4 f 2") ;
        #3 $display("    f 4 f 3") ;

    join
    # 1 $display(" f 5");
join

endmodule
```

Notice that the *always* block repeats when the *fork-join* block finishes running. Can you can calculate the time at which each of the statements will print out?

Example 4-9 Results 1 shows possible results of simulating the code in Example 4-9. It is possible that different simulators might execute the statements scheduled for the same time unit in a different order.

Example 4-9 Results 1

```
b 1
    f 2 b 1
  f 5
    b 1 f 2
  f 1
    f 2 b 2
    b 1 f 1
    b 1 f 4 b 1
    b 1 f 4 b 2
    b 1 f 4 b 3
    f 2 b 3
    f 4 f 1
    b 1 f 3
  b 2
    f 4 f 2
    f 4 f 3
    f 3 b 1
    f 3 b 2
    f 3 b 3
    f 2 b 1
  f 5
  f 1
    f 2 b 2
    f 2 b 3
    f 4 f 1
    f 4 f 2
    f 4 f 3
    f 3 b 1
    f 3 b 2
L2 "befjia.v": $finish at simulation time 50
```

Example 4-9 Results 2 shows the results of the simulation of Example 4-9 with tracing, so you can see when each statement executes.

Example 4-9 Results 2 Trace Output from Combined *begin-end* and *fork-join* Blocks

```
L2  "befjia.v": initial
L2  "befjia.v": #50
L3  "befjia.v": initial
L3  "befjia.v": begin
L4  "befjia.v": #1
L18  "befjia.v":  always
L18  "befjia.v":  fork
L19  "befjia.v":  #3
L20  "befjia.v":  begin
L21  "befjia.v":  #1
L26  "befjia.v":  begin
L27  "befjia.v":  #10
L32  "befjia.v":  #5
L38  "befjia.v":  #1
SIMULATION TIME IS 1
L4  "befjia.v": #1 >>> CONTINUE
L4  "befjia.v": $display(" b 1");
  b 1
L5  "befjia.v": #1
L21  "befjia.v": #1 >>> CONTINUE
L21  "befjia.v": $display("   f 2 b 1");
    f 2 b 1
L22  "befjia.v":  #2
L38  "befjia.v": #1 >>> CONTINUE
L38  "befjia.v": $display("  f 5");
  f 5
SIMULATION TIME IS 2
L5  "befjia.v":  #1 >>> CONTINUE
L5  "befjia.v":  fork
L6  "befjia.v":  #1
L7  "befjia.v":  $display("   b 1 f 2");
    b 1 f 2
L8  "befjia.v":  #5
L9  "befjia.v":  #2
SIMULATION TIME IS 3
L19  "befjia.v": #3 >>> CONTINUE
L19  "befjia.v": $display("  f 1");
  f 1
L22  "befjia.v": #2 >>> CONTINUE
L22  "befjia.v": $display("   f 2 b 2");
    f 2 b 2
L23  "befjia.v":  #3
```

```
L6  "befjia.v":  #1 >>> CONTINUE
L6  "befjia.v":  $display("   b 1 f 1");
    b 1 f 1
SIMULATION TIME IS 4
L9  "befjia.v":  #2 >>> CONTINUE
L9  "befjia.v":  begin
L10 "befjia.v":  $display("   b 1 f 4 b 1");
    b 1 f 4 b 1
L11 "befjia.v":  #1
SIMULATION TIME IS 5
L32 "befjia.v":  #5 >>> CONTINUE
L32 "befjia.v":  fork
L33 "befjia.v":  #1
L34 "befjia.v":  #2
L35 "befjia.v":  #3
L11 "befjia.v":  #1 >>> CONTINUE
L11 "befjia.v":  $display("   b 1 f 4 b 2");
    b 1 f 4 b 2
L12 "befjia.v":  $display("   b 1 f 4 b 3");
    b 1 f 4 b 3
L13 "befjia.v":  end
SIMULATION TIME IS 6
L23 "befjia.v":  #3 >>> CONTINUE
L23 "befjia.v":  $display("   f 2 b 3");
    f 2 b 3
L24 "befjia.v":  end
L33 "befjia.v":  #1 >>> CONTINUE
L33 "befjia.v":  $display("   f 4 f 1");
    f 4 f 1
SIMULATION TIME IS 7
L8  "befjia.v":  #5 >>> CONTINUE
L8  "befjia.v":  $display("   b 1 f 3");
    b 1 f 3
L14 "befjia.v":  join
L15 "befjia.v":  $display(" b 2");
  b 2
L16 "befjia.v":  end
L34 "befjia.v":  #2 >>> CONTINUE
L34 "befjia.v":  $display("   f 4 f 2");
    f 4 f 2
SIMULATION TIME IS 8
L35 "befjia.v":  #3 >>> CONTINUE
L35 "befjia.v":  $display("   f 4 f 3") ;
    f 4 f 3
L37 "befjia.v":  join
SIMULATION TIME IS 10
L27 "befjia.v":  #10 >>> CONTINUE
L27 "befjia.v":  $display("   f 3 b 1") ;
    f 3 b 1
L28 "befjia.v":  #9
SIMULATION TIME IS 19
L28 "befjia.v":  #9 >>> CONTINUE
L28 "befjia.v":  $display("   f 3 b 2");
    f 3 b 2
```

```
L29 "befjia.v"  #8
SIMULATION TIME IS 27
L29 "befjia.v": #8 >>> CONTINUE
L29 "befjia.v": $display(    f 3 b 3");
    f 3 b 3
L30 "befjia.v": end
L39 "befjia.v": join
L18 "befjia.v": always
L18 "befjia.v": fork
L19 "befjia.v": #3
L20 "befjia.v": begin
L21 "befjia.v": #1
L26 "befjia.v": begin
L27 "befjia.v": #10
L32 "befjia.v": #5
L38 "befjia.v": #1
SIMULATION TIME IS 28
L21 "befjia.v": #1 >>> CONTINUE
L21 "befjia.v": $display("   f 2 b 1");
    f 2 b 1
L22 "befjia.v": #2
L38 "befjia.v": #1 >>> CONTINUE
L38 "befjia.v": $display(" f 5");
  f 5
SIMULATION TIME IS 30
L19 "befjia.v": #3 >>> CONTINUE
L19 "befjia.v": $display(" f 1");
  f 1
L22 "befjia.v": #2 >>> CONTINUE
L22 "befjia.v": $display("   f 2 b 2");
    f 2 b 2
L23 "befjia.v"  #3
SIMULATION TIME IS 32
L32 "befjia.v": #5 >>> CONTINUE
L32 "befjia.v": fork
L33 "befjia.v": #1
L34 "befjia.v": #2
L35 "befjia.v": #3
SIMULATION TIME IS 33
L23 "befjia.v": #3 >>> CONTINUE
L23 "befjia.v": $display("   f 2 b 3");
    f 2 b 3
L24 "befjia.v": end
L33 "befjia.v": #1 >>> CONTINUE
L33 "befjia.v": $display("   f 4 f 1");
    f 4 f 1
SIMULATION TIME IS 34
L34 "befjia.v": #2 >>> CONTINUE
L34 "befjia.v": $display("   f 4 f 2");
    f 4 f 2
SIMULATION TIME IS 35
L35 "befjia.v": #3 >>> CONTINUE
L35 "befjia.v": $display("   f 4 f 3");
    f 4 f 3
```

```
L37 "befjia.v": join
SIMULATION TIME IS 37
L27 "befjia.v": #10 >>> CONTINUE
L27 "befjia.v": $display("   f 3 b 1") ;
    f 3 b 1
L28 "befjia.v": #9
SIMULATION TIME IS 46
L28 "befjia.v": #9 >>> CONTINUE
L28 "befjia.v": $display("   f 3 b 2") ;
    f 3 b 2
L29 "befjia.v": #8
SIMULATION TIME IS 50
L2 "befjia.v": #50 >>> CONTINUE
L2 "befjia.v": $finish;
L2 "befjia.v": $finish at simulation time 50
```

Summary of Procedural Timing

One of the most important concepts in Verilog modeling is knowing *when* a procedural statement will be run. The preceding section introduced most of the key words and symbols used to control when a procedural statement will be run. A common cause of incorrect model behavior and even of syntax errors is incorrectly specifying, or omitting, statements that control when your code should be run. If you don't know when your code should be run, perhaps the simulator or synthesis tool will have the same problem.

Table 4-1 Summarizes the keywords presented to determine procedural timing. This list is expanded as more concepts are introduced.

Table 4-1 Procedural Timing keywords

Keyword	Definition
`initial`	`Start here at time zero, run only once.`
`always`	`Start here at time zero, when done, run again.`
`begin - end`	`Sequential grouping of procedural statements.`
`fork - join`	`Concurrent grouping of procedural statments.`
`#`	`Wait some amount of time.`

5 SYSTEM TASKS FOR DISPLAYING RESULTS

The *hello* simulation and the previous chapter's examples gave you a preview of one way to print out information: The *$display* system task. All of the commands to print out results are relatives of the *$display* system task.

What Is a System Task?

As you learn the Verilog language, you will see that Verilog is a flexible language for modeling. There are some special built-in commands for system functions such as printing messages or reading and writing files. The special commands are called *system tasks* and they all begin with the "$" symbol. The "$" symbol is also used to indicate system functions.

$display and Its Relatives

Using the *$display* system task is the basic way to print out results. The simplest form of *$display* is shown in the *hello* simulation in Chapter 2 and is repeated in Example 5-1.

Example 5-1 Displaying a String

```
$display("Hello Verilog");
```

This simple form of *$display* simply prints the string between the quotation marks, followed by a new line.

Example 5-2 Displaying a Single Value

```
$display(a);
```

The form of *$display* shown in Example 5-2 prints out the value of *a* in the default radix, which is decimal. This is a common way to debug a simulation interactively. You can use the *$display* command in your source code or as an interactive command.

Example 5-3 Displaying Multiple Values

```
$display(a, b);
$display(a, , b);
```

The two lines in Example 5-3 show the values of both *a* and *b.* In the first line, the values *a* and *b* are run together, as in 1234. The extra comma in the second *$display* line is not a typo: It adds an extra space in the output. Verilog is not white-space-sensitive, so the language defines the extra comma as a command to insert extra space in the printout. Use this capability to improve the readability of your output. Thus, if the first line in Example 5-3 were to yield the possibly ambiguous value 1234, the second line would eliminate ambiguity by yielding the values 123 and 4.

Example 5-4 Using Format Specifiers with *$display*

```
$display("The value of a is %b, The value of b is %b", a, b);
```

Example 5-4 shows the most common form of the *$display* system task. This form uses format specifiers—in this case, the format specifier *%b*— and then assigns a value to the format specifiers. In Example 5-4, the value of *a* is assigned as binary for the first *%b* and the value of *b* as binary for the second *%b.* (Readers familiar with C programming will notice the similarity to the *printf* function.) This form of *$display* is most common because of its flexibility to print in any radix and combine the printing of text with the values.

The general form for $*display* is

$display([optional format specifier],[value],[value...]);

The *$display* command can be used to print out binary, decimal, hexadecimal, or octal values. The radix is controlled with format specifiers. The most common format specifiers are listed in Table 5-1.

Table 5-1 Format Specifiers

Symbol	Format
`%b`	`binary`
`%d`	`decimal`
`%h`	`hexadecimal`
`%o`	`octal`
`%s`	`string`

Other Commands to Print Results

The *$display* command has several relatives: *$write, $strobe,* and $*monitor. $write* and *$strobe* are very similar to $*display*, and *$monitor* is a special, more powerful command.

$write is similar to *$display:* They both print results when encountered. The only difference between the two is that *$display* automatically puts in a new line at the end of the results, whereas *$write* does not. If you need to print many results on a line and need to use more that a single *$display* statement, use *$write* statements for the first part(s) of the line and then a *$display* for the rest of the line. You could decide never to use *$display,* and just use *$write* and put a new line in manually.

Example 5-5 Two *$display* Statements

```
module two_display;
initial
  begin
    $display("first half  ");
    $display("second half");
  end
endmodule
```

Example 5-5 Results

```
first half
second half
```

Example 5-6 Combining *$write* and *$display*

```
module write_display;
initial
  begin
    $write ("first half ");
    $display(" second half");
  end
endmodule
```

Example 5-6 Results

```
first half second half
```

What happens in the case of a value that changes while you are printing it out? Does Verilog display the old value or the new? If you are using *$display,* an alternative is to put more delay before the *$display* statement. However, there is a special form of *$display* called *$strobe.* If you want to print out your results only after all values are finished changing at the current time unit, use *$strobe. $strobe* waits until just before time is going to advance, then it prints. With *$strobe* you always get the new value.

If you want to print results as they change, use the *$monitor* system task. Unlike *$display,* which prints only once, *$monitor* automatically prints out whenever any of the signals it is printing changes, so you only need to call it once. Only one *$monitor* can be active at a time. If you want to change what is being printed, just execute another *$monitor* system task and the new *$monitor* becomes the active print-on-change system task.

Because *$monitor* can produce a lot of output, there are two more special system tasks for stopping and restarting *$monitor.* To stop the *$monitor* from printing, use the *$monitoroff* command. To restart the *$monitor,* use the *$monitoron* command. Remember that there can be only one *$monitor* active in your simulation at a time. The last one executed is the only one in effect.

Writing to Files

By default, Verilog puts all the output that goes to your screen into a log file called *verilog.log*. You can view the results of your simulation by looking at the log file. Chapter 21 provides details on the log file.

Along with sending output to the screen and log file, Verilog can write up to thirty-one additional files at the same time. File output is accomplished by declaring an integer that is used to represent the file and then opening the file. Once the file is opened, output commands similar to the ones previously described may be used to write to the file.

Example 5-7 Writing to a File

```
module f1;
integer f;
initial begin
    f = $fopen("myFile");
    $fdisplay(f, "Hello Verilog File");
end
endmodule
```

Example 5-7 opens a file called *myFile* and prints the message “Hello Verilog File” into it. The file is closed automatically at the end of simulation. Since only thirty-one files can be opened at a time, the *$fclose* function can be called to close a file.

For each of the commands covered so far, there is an *f* prefixed version of the command for printing data to files. All the file output commands require the first argument to be the file integer. The other command arguments are just like those for *$display*. Table 5-2 lists the screen and file output commands.

Table 5-2 Screen and File Output Commands

Screen + Log	File Output
`$display`	`$fdisplay`
`$write`	`$fwrite`
`$strobe`	`$fstrobe`
`$monitor`	`$fmonitor`

Even though the addition of files may imply that you can have thirty-two *$monitors*, you cannot. There still can only be one *$monitor* active in a simulation.

The integers used to represent the files have exactly one bit set in them. A single *$fdisplay* command can write to more than one file. The trick is to use the **"|"** symbol to connect the file numbers, as shown in Example 5-8. One other trick is the numbering of the files: *1* is reserved for the screen and log file, so the first file opened will be *2* and the next, *4.* Example 5-8 shows how to write to multiple files.

Example 5-8 Writing to Multiple Files

```
module f2;
integer file1, file2;
initial begin
    file1 = $fopen("file1");
    file2 = $fopen("file2");
    $display("The number used for file 1 is %0d", file1);
    $display("The number used for file 2 is %0d", file2);
    $fdisplay(file1, "Hello File 1");
    $fdisplay(file2, "Hello File 2");
    $fdisplay(file1 | file2, "Hello both files");
    $fdisplay(file1 | file2 | 1, "Hello files and screen");
    $fdisplay(file1, "Good Bye File 1");
    $fdisplay(file2, "Good Bye File 2");
    $fclose(file1);
    $fclose(file2);
end
endmodule
```

The resulting output in file1 is shown in Example 5-8 Results 1 in file 1.

Example 5-8 Results 1 in file1

```
Hello File 1
Hello both files
Hello files and screen
Good Bye File 1
```

The resulting output in file2 is shown in Example 5-8 Results 2 in file 2.

Example 5-8 Results 2 in file2

```
Hello File 2
Hello both files
Hello files and screen
Good Bye File 2
```

The output on the screen and in the log file are shown in Example 5-8 Results 3.

Example 5-8 Results 3 Output on Screen and in Log File

```
The number used for file 1 is 2
The number used for file 2 is 4
Hello files and screen
```

Advanced File IO Functions

Subsequent chapters describe Verilog memories and how to read a file into a memory. Until the 2001, standard there was no way to read a file into Verilog other to load data into a memory. The 2001 standard greatly enhances file input and output capabilities.

The 2001 standard defines *$ferror, $fflush, $fgetc, $fgets, $fread, $fscanf, $fseek, $fsscanf, $ftel, $rewind, $sformat, $swrite, $swriteb, $swriteh, $swriteo* and *$ungetc,* as new system functions. These functions work on files opened with *$fopen*, when *$fopen* is called with a "mode" similar to the ANSI C *fopen* function call. Each of these new functions works similar to the ANSI C functions by the same name. The details of these functions are not explained in this Quick-Start book. These functions are similar to the ANSI standard and documentation can be found in your C and Verilog vendors documentation.

Setting the Default Radix

All of the commands for formatting output can be used with or without format specifiers. When you use a format specifier, the radix for each value printed out is set individually. If you do not use a format specifier, the default radix is decimal. Often it is desirable to print out values without having to use a format specifier. When debugging, it is inconvenient to have to use a format specifier just to see a value in a different radix. Thus, Verilog provides four types of each of the output functions with a different default radix. Table 5-3 shows the output commands and their default radixes.

Table 5-3 Enumeration of All Output Commands

Decimal	Binary	Hexadecimal	Octal
`$display`	`$displayb`	`$displayh`	`$displayo`
`$fdisplay`	`$fdisplayb`	`$fdisplayh`	`$fdisplayo`
`$write`	`$writeb`	`$writeh`	`$writeo`
`$fwrite`	`$fwriteb`	`$fwriteh`	`$fwriteo`
`$strobe`	`$strobeb`	`$strobeh`	`$strobeo`
`$fstrobe`	`$fstrobeb`	`$fstrobeh`	`$fstrobeo`
`$monitor`	`$monitorb`	`$monitorh`	`$monitoro`
`$fmonitor`	`$fmonitorb`	`$fmonitorh`	`$fmonitoro`

Special Characters

You have already seen a few of the special characters used for formatting output. This section lists a few more that are useful to format output. To print a percent character, use *%%*. The hierarchical name where the *$display* command is being executed can be printed using *%m*. The *%%* and *%m* format specifiers do not have a companion argument in the comma separated value list following the format string. Table 5-4 lists the format specifiers in Verilog.

Table 5-4 Format Specifiers

Symbol	Description
`%b`	`Binary with leading zeroes`
`%0b`	`Binary with no leading zeroes`
`%v`	`Value and strength`
`%d`	`Decimal`
`%0d`	`Decimal with leading spaces truncated`
`%h`	`Hexadecimal with leading zeroes`
`%0h`	`Hexadecimal with no leading zeroes`
`%o`	`Octal with leading zeroes`
`%0o`	`Octal with no leading zeroes`
`%c`	`Character`
`%s`	`String`
`%m`	`Hierarchical name`
`%t`	`Time format`
`%f`	`real in decimal format`

`%e`	`real in exponential format`
`%g`	`real in the shorter of %f or %e`
`%%`	`The % character`
`\n`	`New line`

The Current Simulation Time

The current simulation time can be printed by calling the system function *$time* or *$realtime.* Example 5-10 shows a simple way to print the current simulation time.

The simulation time is normally unit-less, but you can assign time units and precision. See Appendix A for details on the *`timescale* directive. *$time* and *$realtime* can be printed out using those units and precision using the *$timeformat* system task.

Example 5-9 shows how to use *$timeformat*, and the results show the differences between *$time* and *$realtime*. When the time scale allows non integer delays, *$timeformat* specifies the number of decimal places to print. *$time* will have only the integer portion of the time, but *$realtime* contains the fractional time units.

Example 5-9 Printing out the current time with units

```
`timescale 1ns/10ps
module timeformat;
initial
  begin
    $timeformat(-9, 2, "ns", 7);
    #50   $display("It is now %t (time).",$time);
          $display("It is now %t (realtime).",$realtime);
    #1.01 $display("It is now %t (time).",$time);
          $display("It is now %t (realtime).",$realtime);
    #50   $display("It is now %t (time).",$time);
          $display("It is now %t (realtime).",$realtime);
    #1000 $display("It is now %t (time).",$time);
          $display("It is now %t (realtime).",$realtime);
  end
endmodule
```

Figure 5-1 Shows the definition of the arguments to the *$timeformat* system task call.

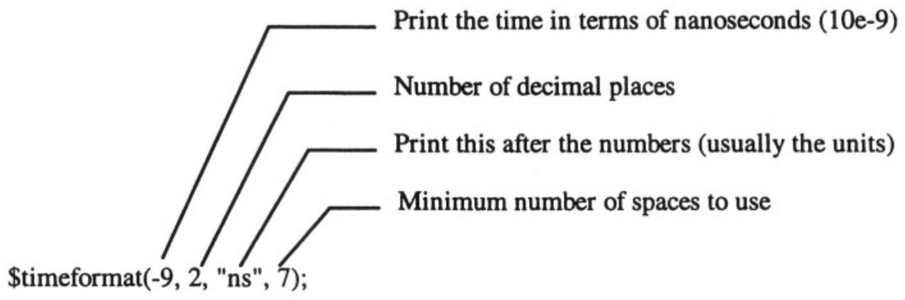
Print the time in terms of nanoseconds (10e-9)
Number of decimal places
Print this after the numbers (usually the units)
Minimum number of spaces to use
$timeformat(-9, 2, "ns", 7);

Figure 5-1 Time format details

Example 5-9 Results

```
It is now   50.00ns  (time).
It is now   50.00ns  (realtime).
It is now   51.00ns  (time).
It is now   51.01ns  (realtime).
It is now  101.00ns  (time).
It is now  101.01ns  (realtime).
It is now 1101.00ns  (time).
It is now 1101.01ns  (realtime).
```

Suppressing Spaces in Your Output

Verilog allocates space in your output to accommodate the largest possible value for the item you are trying to print. This section shows you how to suppress leading spaces in your output.

When you try to print out the time value using the *$display* command, as shown in Example 5-10, you may not get the desired result.

Example 5-10 *$display* with *$time*

```
$display("time = %d", $time);
```

The output has many leading spaces, as shown in Example 5-10 Results.

Example 5-10 Results

```
time =                    100
```

Why is there so much space between the equal sign and the time value? How can you avoid that white space?

Consider Example 5-11 and its results.

Example 5-11 Leading Spaces in *$monitor* with *$time*

```
initial
  $monitor($time,
    "reset: %b clk: %b load %b: up/^dn: %b data: %h",
       reset, clk, ldena, up, data);
```

Example 5-11 Results

```
                 100 reset: 1 clk: 1 load 0: up/^dn: 1 da ...
```

Note that the time value, which is the first item in the *$monitor* list, is drastically indented, which could cause the data to wrap to the next line. This may make the results harder to read. How can you make the *$monitor* message start at the left margin?

You know the size of objects in your design from whatever Verilog code you have written. Consider the Verilog code in Example 5-12 and the results of that code.

Example 5-12 Spaces Used To Print an 8-Bit Value

```
reg [7:0] a;
initial begin
  a=3;
  $display(
  "Decimal a='%d', Hex a='%h', Octal a='%o', Binary a='%b'.",
   a, a, a, a) ;
```

Example 5-12 Results

```
Decimal a='  3', Hex a='03', Octal a='003', Binary
a='00000011'.
```

Note the single quotes (included in both the Verilog code and the output) that allow you to see the exact sizes of the results.

Consider the "Decimal" results first. *a* is an 8-bit register. In Verilog, registers are always unsigned. Because the largest 8-bit number is 255, three spaces are needed to print the highest value for *a.* Note that *%d* does not print leading zeroes.

In the "Hex" results, each character represents 4 bits; therefore two spaces are needed to display the value for *a.* Hex provides leading zeroes.

For the "Octal" results, each character represents 3 bits. Thus, three spaces are needed to display the value for *a.* Octal provides leading zeroes.

Finally, for the "Binary" results, each character represents 1 bit, so eight spaces are need to display the value for *a.* Binary provides leading zeroes.

We can extend this to an explanation of time. Time is a 64-bit unsigned value. Therefore, the largest value that can be displayed is 18446744073709551615, or twenty digits. That is why the examples in this section have so many extra spaces.

Now that we know the reason for the extra spaces, we can create a solution. Change the code in Example 5-12 to the format shown in Example 5-13, and note the results.

Example 5-13 Suppressing Leading Spaces and Zeroes

```
$display("Dec '%0d', Hex '%0h', Oct '%0o', Bin '%0b'",
           a ,a, a, a);
```

Example 5-13 Results

```
Dec. '3', Hex '3', Oct '3', Bin '11'
```

There are two points to remember here:

1) Verilog normally uses fixed-width fields, which makes creating columnar output easy.

2) We can override the leading spaces and zeroes by inserting a 0 between % and the radix code in the format specifier.

PERIODIC PRINTOUTS

The *$monitor* system task prints automatically when any signal changes, however it is often deceiving to read the output from *$monitor,* since many signals can change in a short period of time, or there may be long periods with no changes. It is often desirable to print results in a periodic format. You can combine the *always,* a delay, and the *$display* to create a periodic printout. Example 5-14 provides a simple example of a periodic printout.

Example 5-14 Periodic Printout

```
always #100 $display(" …", … ) ;
```

When to printout results

If you are printing results periodically, you need to choose a good time to print the results. Think about the basic timing. When do inputs change? When are the outputs stable? For example, a system with a clock period of 100 with the clock rising on the even 100's (100, 200, etc.) would likely have signals changing at or just after the even 100's, so printing out just before the clock would be ideal to capture stable values from the previous cycle.

Example 5-15 Periodic Printout Before the Clock

```
always
  begin
    #99 $display("...",…);
    #1 ; // rounds out the cycle to 100
  end
```

A FINAL SYSTEM TASK

With an *always* block as shown in Example 5-14 or Example 5-15, you might wonder when the simulation will ever stop. Although the *$finish* system task is not directly used to print results, it is mentioned here. When a *$finish* system task is encountered, simulation terminates, so all periodic printouts will cease. The *$finish* task can be issued as *$finish; $finish(1);* or *$finish(2);* The difference between the three versions is the amount of simulation statistics printed.

Exercise 3 Printing Out Results from Wires Buried in the Hierarchy

Now that you know the basics of *$display, initial,* and *always,* you can modify the test bench from Exercise 2 to print out a message at the start of simulation.

You may merely want to print a simple message such as “start of adder testing,” or column headings for the data you will be printing. Hint: you can do this by modifying the existing *initial* block or by adding a new *initial* block of your own.

Next, modify the test bench by adding some statements to print results every 50 time units. Use *$display* to print the results from all the inputs (a, *b*, and *cin*) and outputs (*sum* and carry*_out*). How are the results different if you use *$strobe*? Hint: You will need to add an *always* block to do this. You need not worry if your results fail to print at time 0 or time 600: As long as you print your results every 50 time units and have results appearing between 50 to 550 time units, the exercise is successful.

Now that you are printing the top-level signals correctly, try and print out some signals buried in the hierarchy. Modify the *$display* or *$strobe* statements to include printing out the values from internal carries between the 2-bit adders. There are three *carry* signals between the 2-bit adders.

You have now successfully added *$display* statements to a module. This is one of the easiest ways to debug a design. With your practice at hierarchical names in Chapter 3, you can print out any signal in the design from the test bench.

6 DATA OBJECTS

DATA OBJECTS IN VERILOG

This chapter introduces the different types of data you can work with in Verilog: Nets, regs, integers, times, parameters, events, and strings.

Nets

Nets (sometimes called wires) are the most common data object in Verilog. Nets are used to interconnect modules and primitives, as discussed in Chapter 3. You used nets in Exercise 2. There are net types representing wired OR, wired AND, storage nodes, pullups, and pulldowns. The default net type is a plain wire with no special properties.

The net types are listed in Table 6-1.

Table 6-1 Net Types

Net Type	Description
`wire`	`Default net type, a plain wire`
`tri`	`Another name for wire`
`wand`	`Wired AND`
`triand`	`Another name for wand`
`wor`	`Wired OR`
`trior`	`Another name for wor`
`tri1`	`Wire with built-in pullup`
`tri0`	`Wire with built-in pulldown`
`supply1`	`Always 1`
`supply0`	`Always 0`
`trireg`	`Storage node for switch-level modeling`

wire is the default net type. You can change the default net type to one of the other types, but most nets are of type *wire. wire* and *tri* are the same type of net. The reason for having two names for this type of net is that some people may want to distinguish in their designs those nets that are expected to tri-state from those that do not. You can use *tri* to distinguish a net that you expect to have high impedance values or multiple drivers. If two drivers are driving a net of type *wire* or *tri* and one driver has the value 1 and the other has the value 0, the result will be *x.*

wand and *triand* are wires that represent wired AND logic. Wired AND logic is similar to open-collector TTL logic. If any driver on the net is 0, the resulting value is 0. Verilog does not distinguish between *wand* and *triand.* The different names are only for use in documenting your model.

Wired OR logic is represented with the *wor* and *trior* net types. With these net types, if any driver is a 1, the result is 1. As with the previous net types, *wor* and *trior* are equivalent.

If nothing is driving a wire in TTL logic, the inputs default to 1. You can use the *tril* net type to model this situation. If nothing is driving a net of type *tril,* the default value is 1. As with *tril,* if nothing is driving a net of type *tri0,* the value is 0.

Use the *supply1* and *supply0* net types to model power supply nets. These nets are always 1 or 0 with a strength of supply. Even if you drive something onto these nets, they always retain their distinct values.

The *trireg* net type is used in switch-level modeling for storage nodes. The *trireg* net has a capacitive size associated with it. Because *trireg* is an abstraction of a storage node, the capacitors never decay. See appendix A for the interaction between capacitive size and gate drive strength.

In Verilog, a wire can be 1 bit wide or much wider. A wire that is more than 1 bit wide is called a *vector* in Verilog. (Although such a wire is also known as a *bus,* this book uses the term *vector* for a wire wider than 1 bit.) To declare a wire more than one bit wide, a *range* declaration is used.

Ranges

Ranges specify the most-significant and least-significant indexes of a vector. The maximum width of a vector is dependent on the simulator being used. The IEEE 1364 standard states that a simulator must support at least 1024-bit wide vectors. Verilog-XL supports vectors up to one million (1,000,000) bits wide, though some simulators have no limit on width.

You specify the number of bits in a wire with a bit range. The range [7:0] is an 8-bit range, as is [0:7]. Ranges can be either ascending or descending. Furthermore, ranges do not need to be zero-based: The range [682:690] is a 9-bit range.

Example 6-1 shows several net declarations.

Example 6-1 Net Declarations

```
wire a, b, c; // Three 1-bit nets of type wire.
wire [7:0] d, e, f; // Three 8-bit vectors.
supply1 vcc;
supply0 gnd;
trior [26:2] data_bus; // A 25-bit vector.
```

Each net declaration can declare several nets of the same type and size. The range is associated with the net type declaration, not the net name. Therefore, Example 6-2 is incorrect. The 2001 Verilog standard includes multi-dimensional arrays, which makes Example 6-2 legal. Since this is a change to the language, individual tool support of this feature may vary. While the Syntax may now be legal, it is considered wrong since it is an array of 1 bit wires, vs. the desired 8 bit bus.

Example 6-2 Incorrect Net Declaration

```
wire a[7:0]; // WRONG although syntax ok in IEEE1364-2001
```

The left-most index is always the most significant bit. Verilog is only concerned with the number of bits in the range. Use of ascending and descending ranges is entirely up to you and your conventions. Verilog does not need the ranges to be zero- or one-based.

Implicit Nets

In the *mux* and *adder* examples, in chapter 3, nets were used even though none were declared. Verilog implicitly declares nets for every port declaration. Every connection made in a module instance or primitive instance is also implicitly declared as a net, if it is not already declared. Nets implicitly declared from a port declaration carry the size and name of the port and are the default net type, usually *wire*. Nets that are implicitly declared because they are part of an instance are 1 bit wide and of the default net type. If you need a net to be more than 1 bit wide, you must explicitly declare it.

You can set the default net type by using the Verilog compiler directive *`default_nettype*. This compiler directive sets the net type for all implicitly declared nets. Compiler directives start with the grave accent key, *not* the apostrophe. Example 6-3 shows two settings the default net type.

Example 6-3 Setting Default Net Type

```
`default_nettype tri1;
`default_nettype wand;
```

Ports

Ports were introduced in Chapter 3 with structural modeling. A port declaration implies a net of the default type and the same range as port. Ports can be re-declared as a different net type if desired, however the ranges must match. Example 6-4 shows port declarations and re-declarations.

Example 6-4 Port Declarations

```
module portexample( a, b, c, d);
 input  [7:0]  a;   //  implies wire  [7:0]  a;
 input  [3:0]  b;
 tri1  [3:0]  b;   //  if b is not driven it wil be 4'b1111
 inout  [7:0]  c;
 triand [7:0] c; // multiple drivers on c will be anded
 output  [5:0]  d;
wire  [7:0]  d;  // Wrong the ranges dont match!
```

Regs

Regs are used for modeling in procedural blocks. The next chapter explains usage of the *reg*. The *reg* data type does not always imply modeling of a flip-flop, latch, or other form of register. The *reg* data type can also be used to model combinatorial logic. A register can be 1 bit wide or declared a vector, just as with nets. Vector registers can be accessed a bit at a time or a part at a time. Example 6-5 shows some register declarations.

Example 6-5 Reg Declarations

```
reg a, b, c; // Three 1-bit registers.
reg [8:15] d, e, f; // Three 8-bit registers.
```

Part of a *reg* can be referenced or assigned to by using a bit- or part-select notation. Remember that the leftmost bit is the most significant, regardless of how the range is declared. When you select a part or slice of a register, be sure the range of the part matches the range direction (ascending or descending) of the original register. Also, if you select a range that is not within the original register, the result will be *x*, and is not an error.

Example 6-6 Selecting Bits and Parts of a Reg

```
e[15] // Refers to the least significant bit of e.
d[8:11] // Refers to the four most significant bits of d.
```

Memories

Memories are arrays of registers. A memory declaration is similar to a *reg* declaration with the addition of the range of words in the memory. The range of words can be ascending or descending, as with the range of a vector. The range of words does not need to be zero- or one-based; it can start anywhere. It is usually most convenient to declare a memory as zero-based with an ascending range. Verilog uses 2 bits of computer memory for each bit of simulated memory because a bit of simulated memory may contain the values 0, 1, *x*, or *z*. When referencing a memory, you can access only the entire word of memory, not the individual bits.

Example 6-7 Memory and Register Declarations

```
reg [7:0] a, b[0:15], c[971:960];
reg d, e[8:13];
```

In Example 6-7, *a* is an 8-bit register, *b* is a memory of sixteen 8-bit words, and *c* is a memory of twelve 8-bit words. The second *reg* declaration declares *d* as a 1-bit register and *e* as a 1-bit wide memory of six words.

It can be difficult to distinguish memory word references from the reference of a bit of a register. There is no direct way to reference a bit of a memory. Therefore, word references in a memory (which look exactly like bit references in a word) can only be distinguished if you know the data type of the referenced element. Refer to the declaration to determine whether you are referencing a bit or a word. Example 6-8 shows selecting bits in *regs* and words in memories.

Example 6-8 Selecting Bits in Regs and Words In Memories

```
a [3]  // Refers to bit three of the register a.
b[3]  // Refers to the fourth 8-bit word in the memory b.
b[3][3]  // This is not legal through 1995 standard.
```

Initial Value of Regs

The initial value of a *reg* or array of *regs* is *'bx* (unknown). IEEE1364-2001 defines a method to initialize a *reg* as part of the declaration. Example 6-9 shows the declaration of five *regs, a, b, c, d,* and *e: Regs a* and *c* are not given initial values and default to unknown. As with the other IEEE1364-2001 changes tool support for these language features may not be immediate or complete. The standard does not specify an order of evaluation of these initial values versus an *initial* block with a procedural assignment to the same reg.

Example 6-9 Reg Declaration with Initialization

```
reg  [7:0]  a;          //  initial value will be 8'bx;
reg  [7:0]  b = 8'd3;  //  initial value will be 3
reg  [3:0]  c, d=3, e=4;
```

Integers and Reals

Integers in Verilog are usually 32 bits wide. Strictly speaking, the number of bits in an integer is machine-dependent. Verilog was created when 36-bit machines were common, so a 36-bit machine would have an integer of 36 bits. Today most machines work with 32-bit integers. For this book, we assume that integers are 32 bits wide.

Integers are signed; *regs* are unsigned. If you want to do signed arithmetic, use an *integer*. Otherwise an *integer* is similar to a 32-bit *reg*. Note that it is possible to do signed math using nets and registers, but your modeled logic must explicitly model the sign extension.

Integers, reals, and 32-bit registers each physically hold a 32-bit value. The difference between them is in their interaction with operators and what the data means.

A *reg* merely represents bits and is treated as an unsigned integer. An integer is signed and can hold a negative number. A real holds a floating-point number in IEEE format.

Integers are declared with the *integer* keyword, not *int* as in the C programming language.

Like integers, *reals* are 32-bit floating-point values. Integers and reals are difficult to pass through ports because in Verilog, ports are always bits or bit vectors. Reals are declared with the *real* keyword.

Example 6-10 shows how to declare integers and reals.

Example 6-10 Declaring Integers and Reals

```
integer i, j, k;
real x, y;
```

Time and Realtime

Verilog uses the *time* keyword to represent the current simulation time. ***time*** is double the size of an integer (usually 64 bits) and is unsigned. If your model uses a *timescale* you can use *realtime* to store the simulation time and time units. You can declare variables of type *time* or *realtime* in your models for timing checks, or in any other operations you need to do with time. See Appendix A.

The built in functions *$time* and *$realtime* return the current simulation time.

Example 6-11 Declaring Variables of Type *time*

```
time t1, t2;
realtime rt1, rt2;
```

```
initial begin
  #50 t1 = $time;
     rt1 = $realtime;
  #50 t2 = $time - $t1;
     rt2 = $realtime - rt1;
end
```

Parameters

Parameters are run-time constants that take their size from their value automatically. The default size of a parameter is the size of an integer (32 bits). For backwards compatibility, you can declare parameters with ranges to make them bigger or smaller than their default size. Parameters are chiefly useful in creating modules with adjustable sizes or delays. Even though parameters are run-time constants, their values can be updated at compilation time. Each instance of a module with parameters can have different values for those parameters at run time. Unlike the declarations of *net, reg, integer, real,* and *time,* when you declare *parameter,* it is assigned a default value. Parameters may be strings. Parameters may be used in subsequent declarations.

Example 6-12 Parameters

```
parameter message = "Hello Verilog";
parameter size =8;
parameter delay =3;
parameter prog = size * delay;
parameter msb = size -1;
parameter low = 0;
wire [msb:0] a;      // parameter msb +1 determine width of a
reg [size-1:low] b;  // size and low determine width
```

Events

Events were first used in the phone example in Chapter 1. They are usually used in very abstract models. An event does not represent any real hardware. Events have no value or duration. They are used to signal that something has occurred to trigger something else to happen. Events cannot be passed through ports.

Example 6-13 Events

```
event birth;
event acknowledge, parity_error;
```

Strings

Verilog does not have a unique string data type. Rather, strings are stored in long registers using 8 bits (1 byte) to store each character. When declaring a register to store a string, you must declare a register of at least eight times the length of the string. Constant strings are treated as long numbers, as shown in Example 6-14.

Example 6-14 Strings

```
module string1;
reg[8*13 : 1] s;
initial begin
    s = "Hello Verilog";
    $display("The string %s is stored as %h", s, s);
end
endmodule
```

The result is shown in Example 6-14 Results.

Example 6-14 Results

```
The string Hello Verilog is stored as
48656c6c6f20566572696c6f67
```

Multi-Dimensional Arrays

The 2001 IEEE Verilog standard removes some old restrictions and adds new functionality for multi-dimensional arrays. Example 6-2 becomes legal with the 2001 standard, and the last line of Example 6-8 becomes the selection of a bit within a word.

Example 6-15 Multi-Dimensional Arrays of nets

```
wire [7:0] a; // old style array of wires (bus)
wire [7:0] b[7:0]; // New array of array of wires
wire c[7:0]; // Array of wires.
wire d[7:0][7:0]; // two dimensional array of wires
```

Example 6-15 shows many possible declarations for single and multi-dimensional arrays now possible with the IEEE 1364-2001 standard. Support for multi-dimensional arrays as with any of the language features may be tool specific.

Example 6-16 shows declarations of three objects: *bus, rom* and *screen.* The declaration of *bus* is a single 8-bit *reg,* this declaration is equivalent to the declarations in Example 6-5. The declaration of *rom* is an array of 256 *regs,* 8-bits wide, equivalent to Example 6-7. The final declaration of screen is a two dimensional array of 8-bit words.

Example 6-16 Multi-Dimensional Arrays of Regs

```
reg [7:0] bus, rom[0:255], screen[0:1023][0:767];
```

Accessing Words and Bits of Multi-Dimensional Arrays

The addition of multi-dimensional arrays adds a much needed syntax to the Verilog language that also enables selecting a bit from a word in a memory. You can access a word of a multi-dimensional array, or a bit of a word, but you can not access a range of words.

Example 6-17 Accessing Multi-Dimensional Arrays

```
rom [5]            // an 8 bit word from rom
rom [5] [6]        // A bit of one of the rom words
screen [1][2]      // an 8 bit word of screen
screen [1] [2] [3] // a bit from screen
screen [1]         // not legal
```

PORTS AND REGS

Up to this point, examples of ports have been nets going through ports. As you move towards procedural modeling in Verilog, you may want to have ports that are *regs.* It is legal to declare only output ports as registers. It is a common error to declare *input* or *inout* ports as registers.

Because the only way to get a value into a *regs* is with a procedural assignment, which will be explained in the next chapter, it neither makes sense nor is legal in Verilog to have a reg as an *input* port on the inside of a module.

However, a *reg* may drive an output port, so it is legal for an *output* port to be a reg.

input and *inout* ports must always be nets, but *output* ports can be *reg.* To make an output port a *reg,* first declare it as an *output* then declare it again as a *reg.* A simple example is shown in Example 6-18.

Example 6-18 Output as a Reg

```
`define REG_DELAY 1
module dff(q, clk, d);
input clk, d;
output q;
reg q;

always @(posedge clk) q <= #(`REG_DELAY) d;

endmodule
```

Figure 6-1 shows the possible relationships of ports and *regs.* In procedural modeling, you will often want to declare an *inout* port as a *reg,* but this will not work. An internal *reg* is needed along with a method to connect the *reg* to the *inout* port. Chapter 12 shows how to connect a *reg* to an *inout* port.

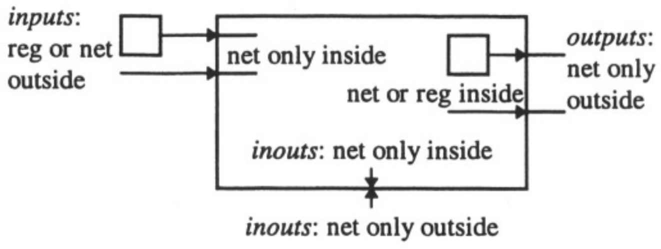

Figure 6-1 Relationships of ports and regs

7 PROCEDURAL ASSIGNMENTS

The data types *reg, integer, real,* and *time* can only be assigned values in procedural blocks of code. These assignments are called *procedural assignments.* They are similar to variable assignments in other programming languages. When the statement is executed, the variable on the left-hand side of the assignment receives a new value.

The destination of a procedural assignment is never a *wire.* The procedural assignment is one of three types of assignments you will learn in Verilog. For now, just remember that the left-hand side of a procedural assignment is a *reg.* The left-hand side can contain an *integer, time,* or *real,* but these data types can be thought of as abstractions of *regs.*

There are three varieties of the procedural assignment: The simple procedural assignment, the procedural assignment with an intra-assignment delay, and the nonblocking procedural assignment, all of which are described in this section.

Example 7-1 Simple Procedural Assignments

```
module ia;
integer i, j;
reg [7:0] a, b;
initial begin
    i = 3;
    j = 4;
    a = i + j;
    b = a + 1;
    #10 i = a;
    j = b;
end
endmodule
```

In Example 7-1, the first four assignments occur at time 0, followed by a delay of 10 time units, and then the last two assignments take place. This example shows how an assignment can have no delay or have a delay before the assignment.

Example 7-2 Procedural Assignments with *fork-join*

```
module iaf1 ;
integer i, j;

initial begin
    i = 3;
    j = 4;
    fork
        #1 i = j;
        #1 j = i;
    join
end
endmodule
```

In module *iaf1,* what are the final values of *i* and *j*? The answer is indeterminate. At time 0, i and *j* are assigned the values 3 and 4. At time 1, *j* is sampled and its value assigned to *i,* and the value of *i* is sampled and applied to *j.* Even though the module contains a *fork-join* block and the changes should happen at the same time, we don't know the result because both values are sampled and changed at the same time. If the code is changed to use an intra-assignment delay, we can be sure they will exchange values.

The *intra-assignment delay* is a special form of the procedural assignment with a delay in the middle. With the delay on the right-hand side of the equal sign, the right-hand side is evaluated immediately, but the assignment is delayed. The operation of a procedural assignment with an intra-assignment delay is sample the values on the right had side, delay, then assign.

Example 7-3 ***fork-join*** with Intra-assignment Delays

```
module iaf2;
integer i, j;

initial begin
    i = 3;
    j = 4;
    fork
        i = #1 j;
        j = #1 i;
    join
end
endmodule
```

With the intra-assignment delay, the values of *i* and *j* are sampled at time 0. (They are sampled at time 0 because there are no delays between them and the *initial* statement, which started at time 0). Then there is a delay of 1 and *i* and *j* are assigned their new values. Adding intra-assignment delay creates a special form of the procedural assignment – with a delay in the middle. With the delay on the right-hand side of the equal sign, the right-hand side is evaluated immediately, but the assignment is delayed. The operation of a procedural assignment with an intra-assignment delay is sample the values on the right had side, delay, then assign.

In Example 7-3, the *fork-join* block is started at time 0 and finished at time 1 because a *fork-join* block finishes when the last statement in the *fork-join* block is completed. In this case, both statements take one time unit to complete.

Example 7-4 ***fork-join*** with Multiple Delays

```
module iaf3;
integer i, j;

initial begin
    i = 3;
    j = 4;
    fork
        #1 i = #1 j;
        #1 j = #1 i;
    join
end
endmodule
```

Delays can be added before the assignments. Even with these additional delays, they still exchange values. In Example 7-4, *i* and *j* are sampled at time 1 and assigned their new values at time 2. The module finishes running at time 2. This model is exactly the same as the one in Example 7-5.

Example 7-5 *fork-join* with Simplified Delays

```
module iaf4;
integer i, j;

initial begin
    i = 3;
    j = 4;
    #1 fork
        i = #1 j;
        j = #1 i;
    join
end
endmodule
```

The intra-assignment delays do not change the amount of time taken to run the statement—they merely insert a delay between the sampling and the assignment. This is more easily visible in Example 7-6.

Example 7-6 Effect of Intra-assignment Delays on Time Flow

```
module iab;
integer i, j;

initial begin
    i = 3;
    j = 4;
    begin
        #1 i = #1 j;
        #1 j = #1 i;
    end
end
endmodule
```

Simulation is started at time 0 at the *initial* statement, when *i* and *j* get their first values, 3 and 4. Simulation continues until the first *#1* and waits until time 1. At time 1, j is sampled (having the value 4); at time 2, the value 4 is assigned to *i,* and the statement is completed.

At time 2, simulation continues to the *#1 j = # 1 i* statement, when the simulation waits until time 3, based on the first *#1* in that statement.

At time 3, *i* is sampled (with the value 4); at time 4 the value 4 is assigned back to *j*. Without the *fork-join* block, the statements are sequential and *i* and *j* do not exchange values. Although the extra *begin-end* blocks add some clarity to your code, they have no effect on this design and could be removed.

There is one more form of the procedural assignment, the *nonblocking assignment.* The nonblocking assignment uses a different assignment operator and changes the amount of time the statement takes to execute. The nonblocking assignment allows the next statement (in sequential code) to commence sooner, and defers when the assignment will take place. Example 7-7 shows nonblocking assignments.

Example 7-7 Nonlocking Assignments

```
module ianb;
integer i, j;

initial begin
    i = 3;
    j = 4;
    begin
        i <= #1 j;
        j <= #1 i;
    end
end
endmodule
```

With the nonblocking assignment, the intra-assignment delay does not block. The delay in the assignment is hidden. This is the new sequence of events: At time 0, *i* and *j* receive their values, and the inner *begin-end* block starts. In the first nonblocking assignment, *j* is sampled at time 0, and the value 4 is scheduled to be assigned to *i* at time 1 (based on the *#1*).

The assignment statement finishes at time 0. However, the assignment of *j* to *i* is deferred to time 1, because it is a nonblocking assignment. Then the second nonblocking assignment statement starts at time 0, *i* is sampled, and the value 3 is scheduled to be assigned to *j* at time 1. The *begin-ends* finish at time 0, but the behavior does not complete until time 1 when the assignments are completed. The nonblocking assignment breaks the normal flow of Verilog execution and schedules the assignment to take place at a later time.

PROCEDURAL ASSIGNMENTS, PORTS AND REGS

The previous chapter ended with the relationship of ports and regs. Now that procedural assignments have been introduced, the relationship should be more clear. The left hand side or destination of a procedural assignment must be a reg. Procedural assignments are a powerful way to create combinatorial or sequential logic. Chapter 9 will describe how to create combinatorial and sequential logic. Remember if you want to use the power of the procedural assignment to create logic, the output of the assignment, and the module will need to be a reg.

BEST PRACTICES WITH PROCEDURAL ASSIGNMENTS

The examples presented up to this point have been abstract, and have shown the details of the workings of the procedural assignment. The procedural assignment is the main component of procedural modeling, therefore learning best practices will minimize errors. The procedural assignment can be used to model two types of hardware: Combinatorial logic and sequential logic.

Procedural Assignment for Combinatorial Logic

When modeling combinatorial logic it is recommended to use the blocking procedural assignment with no delays. Example 7-8 shows some combinatorial logic. Remember that although the *reg* must be used as the destination of a procedural assignment, the *reg* can still be used to model combinatorial logic.

Example 7-8 Combinatorial Procedural Assignments

```
`define IO_ADDRESS 16'h1234
`define REG_ADDRESS 16'h5678
module addressdecoder(address, wr, rd, reg_rd, reg_wr,
                        io_rd, io_wr);
input  [15:0] address;    // address from processor
input rd, wr;             // read and write signals
output reg_rd, reg_wr;    // signals to register block
output io_rd, io_wr;      // signals to Io block
reg reg_rd, reg_wr;       // declared as reg as required
reg io_rd, io_wr;         // for procedural assignments
reg io_sel, reg_sel;      // internal signals
always @(address or rd or wr)
  begin
    io_sel = (address == 'IO_ADDRESS) ;
    reg_sel = (address == `REG_ADDRESS);
    io_rd = io_sel & rd;
    io_wr = io_sel & wr;
    reg_rd = reg_sel & rd;
    reg_wr = reg_sel & wr;
  end
endmodule
```

Procedural Assignment for Sequential Logic

Sequential logic, flip-flops, registers, state machines, etc., are quite natural to model with the procedural assignment and *reg*. The best practice to model sequential logic is to use the non blocking assignment with an intra-assignment delay. Example 7-9 shows a register created with a sequential procedural assignment

Example 7-9 Sequential Procedural Assignment

```
`define REG_DELAY 1
module addressregister(clk, reset, address, reg_address);
input clk, reset;
input [15:0] address;
output [15:0] reg_address;
reg [15:0] reg_address;

always @(posedge clk)
  if(reset)
    reg_address <= #(`REG_DELAY) 16'h00;
  else
    reg_address <= #(`REG_DELAY) address;
endmodule
```

Philosophy of Intra-assignment Delays for Sequential Assignments

In Example 7-9, the intra-assignment delay is shown as a text macro. This allows the delay to be zero, one, random, or any other value desired. The delay should be non-zero, but much shorter than the clock period. Unfortunately some code checking tools (lint tools) may flag intra-assignment delays for sequential logic as a warning since synthesis tools ignore these delays. These false warnings should always be ignored. The more important warning is when the delays are omitted from sequential logic that is modeled with non-blocking procedural assignments.

One of the benefits of the intra-assignment delay is the visibility and clarity it adds to a waveform. You can easily tell if a signal arrived in time for the clock and determine which signals are created as a result of the clock.

One of the most important reasons for using a delay is matching pre-synthesis and post-synthesis simulations. The clock-to-out delay of flip-flops is non-zero. With the intra-assignment delay, the clock-to-out delay is modeled. In a pre-synthesis simulation with gated clocks or generated clocks it is possible that data will be seen on the wrong edge if the delays are omitted.

Finally, a word about event ordering and bad practices. In general, event ordering can not be predicted with the exception of a sequence of statements in a single *begin-end* block. Different simulators may execute the same code in slightly different order. A simulation that depends on event ordering rather than timing is likely to be plagued by zero delay race conditions and may give different results on different simulators. It may have difficulties matching pre-and post-synthesis results. Users have been known to use '#0' to nudge event ordering. The '#0' should be avoided and seen as an error. The non-blocking assignment without a delay is equivalent to '<= #0', since the assignment takes place in this time unit but later. Therefore, the non-blocking without a delay should be considered an error.

Conventions Moving Forward

The remainder of the book uses the blocking with no delays for combinatorial logic, and the non-blocking with a text macro for an intra-assignment delay for sequential logic. You should follow this practice as well with all your hardware models. The only violations of this convention you will find are either abstract examples or test-benches.

8 OPERATORS

Operators in Verilog can be divided into several categories. One way to categorize the operators is by the number of operands they take. For example, the + symbol takes two operands, as in *a* + *b*. When an operator takes two operands, it is called a *binary operator*. Verilog, like most programming languages, has many binary operators. Verilog also includes unary operators (which take only one operand), and a ternary operator (which takes three operands).

Another way to group the operators is by the size of what they return. Some operators, when operating on vectors, return a vector. But two types of operators return a single-bit value even if they are passed vectors. The operators that return only a single bit are either reduction or logical operators.

BINARY OPERATORS

Most of the operators in Verilog take two operands, and fall into the category of binary operators. This includes a set of arithmetic, bit-wise, and logical operators.

Table 8-1 Arithmetic Operators

Symbol	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder)
**	Power (exponent) New in IEEE1364-2001

The definitions of the arithmetic operators are similar to other programming languages. IEEE 1364-2001 adds the power operator that was previously not part of Verilog.

Table 8-2 Bit-wise Operators

Symbol	Description
|	OR
&	AND
^	Exclusive OR
<<	Shift left
<<<	Signed shift left New in IEEE1364-2001
>>	Shift right
>>>	Signed shift right New in IEEE1364-2001

The bit-wise |, &, and ^ operators typically will be used with two operands of the same size, and return a value of the same size. The shift operators can end up creating a larger (left shifts) or smaller (right shifts) result. All of the shift operators except *signed shift right* >>> zero fill. The *signed shift right* fills with whatever the most significant bit of the right left hand operator was.

Table 8-3 Logical Operators

Symbol	Description
&&	Logical AND
||	Logical OR

All the binary operators take two arguments that are 1 or more bits long and return a result of 1 or more bits. Logical operators return a one bit result. There are no size restrictions on the operands or results. For example, you can add two 8-bit values and put the result into 4 bits and you would have the four least significant bits. You could also put the result of that addition into a 9-bit result and you would have the carry along with the result.

Example 8-1 Using Operators

```
reg [7:0] a, b, r8;
reg [3:0] r4;
reg [8:0] r9;

  r4 = a + b ; // Gets the four least significant bits.
  r9 = a + b; // Gets the whole result plus a carry out.
  r4 = a >> b; // a shifted right by b bits,
               // four least significant bits of the result.
               // msb's are filled with zeros
  r8 = a >>> b; // a shifted right by b bits,
                // msb's are filled with a[7]
  r8 = r4 | r9; // all right justified, msb lost.
```

UNARY OPERATORS

The unary operators take only one operand to their right for input and consist of negation operators and reduction operators. The unary negation operators are shown in Table 8-4.

Table 8-4 Negation Operators

Symbol	Description
~	Bitwise negation (complement)
!	Logical negation

The bit-wise negation operator can be combined with the bit-wise AND, reduction AND, reduction OR, and exclusive OR operators to make even more bit-wise functions.

Example 8-2 shows the difference between the bit-wise and logical negations. Bit-wise operators return a value of the same size as the operand. Logical operators return only a 1-bit value. Example 8-2 and Example 8-8 show the difference between bit-wise and logical operators.

Example 8-2 Distinguishing between Bit-wise and Logical Operators

```
module uop;
reg [7:0] a, b, c;
initial begin
  a=0;
  b='b10100101;
  c='b1100xxzz;
  $display("Value %b Bitwise '~' %b logical '!' %b",a,~a,!a);
  $display("Value %b Bitwise '~' %b logical '!' %b",b,~b,!b);
  $display("Value %b Bitwise '~' %b logical '!' %b",c,~c,!c);
end
endmodule
```

Example 8-2 Results

```
Value 00000000 Bitwise '~' 11111111 logical '!' 1
Value 10100101 Bitwise '~' 01011010 logical '!' 0
Value 1100xxzz Bitwise '~' 0011xxxx logical '!' 0
```

Other languages such as C, include other unary operators. For example, “++” and “--”. Verilog does not include these unary operators.

REDUCTION OPERATORS

Reduction operators are a special case of the bit-wise operators. The reduction operators act like unary operators in that they take only one operand. The reduction operators act on a multiple-bit operand and reduce it to a single bit.

Table 8-5 Reduction Operators

Symbol	Description
&	Reduction AND
|	Reduction OR
^	Reduction exclusive OR (parity)
~&	Reduction NAND
~|	Reduction NOR
~^	Reduction exclusive NOR

Example 8-3 shows the usage of some of the reduction operators. Reduction operators operate on a vector and return a single bit. Example 8-3 Results shows the results of various reduction operators.

Example 8-3 Using Reduction Operators

```
module redop;
reg [7:0] example[1:5];
integer i;
initial begin
  example[1] = 0;
  example[2] = 'hff;
  example[3] = 'b10101101;
  example[4] = 'b110011zz;
  example[5] = 'b1111111x;
  $display("reduction operators");
  for(i=1; i<=5; i=i+1)
    $display("Value %b, & = %b, | = %b, ^ = %b",
        example[i], &example[i], |example[i], ^example[i]);
end
endmodule
```

Example 8-3 Results

```
reduction operators
Value 00000000, & = 0, | = 0, ^ = 0
Value 11111111, & = 1, | = 1, ^ = 0
Value 10101101, & = 0, | = 1, ^ = 1
Value 110011zz, & = 0, | = 1, ^ = x
Value 1111111x, & = x, | = 1, ^ = x
```

TERNARY OPERATOR

The ternary operator takes three operands and uses the question mark (?) and colon (:) to indicate the operation. A ternary operation is essentially an *if-then-else* statement in an expression. The first operand is logically evaluated. If it is true, the second operand is returned. If the first operand is not true, the third operand is returned.

Example 8-4 Ternary Operator

```
result = a ? b : c ;
```

Table 8-6 Truth Table for Ternary Operator

a	Result
1	b
0	c
x	common bits of b and c, as is, mismatched bits are 'bx

The ternary operator is useful for describing 2-to-1 muxes and three-state buffers.

Example 8-5 Using the Ternary Operator for a Three-State Buffer

```
module buf16(out,in,enable);  // 16 bit three-state buffer
input [15:0] in;
output [15:0] out;
input enable;
assign out = enable ? in : 16'bz;  // This is a continuous
                                   // assignment.  It will be
                                   // explained next chapter.
endmodule
```

EQUALITY OPERATORS

The set of operators used to determine equivalence, greater than, and less than is similar to other languages you might know, with a few additions. Because Verilog includes the values of unknown *x*, and high impedance *z*, it provides some special equivalence checks. Because one of the operands in an equality check may be unknown, the result may also be unknown. Table 8-7 lists the equality operators, and is followed by truth tables for all the equality operators.

Table 8-7 Equality Operators

Symbol	Description
==	Equivalence
===	Literal equivalence
!=	Inequality
!==	Literal inequality
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

The === and !== operators are special in that they will never return *x;* their output is always 0 or 1.

A rule of thumb for equivalence vs. literal equivalence operators is based in hardware. In hardware there is no *x,* it would be *1* or *0.* Therefore use == and != for synthesizable hardware. Test benches should test and catch *x* and *z.* Test benches should use === and !==.

The truth tables for all of the equality operators are shown in Table 8-8 through Table 8-15.

Table 8-8 Truth Table for *a* == *b*

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

Table 8-9 Truth Table for *a* === *b*

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

Table 8-10 Truth Table for *a* != *b*

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

Table 8-11 Truth Table for *a* !== *b*

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

Table 8-12 Truth Table for *a* < *b*

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

Table 8-13 Truth Table for *a* <= *b*

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

Table 8-14 Truth Table for *a* > *b*

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

Table 8-15 Truth Table for *a* >= *b*

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

If you are not sure of how an operator works, you could write a simple Verilog module to test it. Example 8-6 shows such a module.

Example 8-6 Module To Test an Operator

```
/* module to test operators */
module test_op;
reg a,b,result;
reg [1:4] values;

'define op ==

integer i,j;

initial begin
  values = 4'b01xz; // all possible values

$display("  A == B = " ) ;
$display ( "__________") ;
  for(i=1;  i<=4;  i=i+1)
    for(j=1; j<=4; j=j+1) begin
      a = values[i];
      b = values[ j ];
      result = a ` op b;
      $display("  %b    %b   %b",a,b,result);
    end
end
endmodule
```

CONCATENATIONS

You can make larger operands with concatenations. Concatenations are legal both as a result on the left-hand side of the equals and as operands on the right-hand side of the equals. The concatenation is indicated with curly braces {}.

The repeat operator is a special case of the concatenation, and is indicated with two sets of curly braces and a number to indicate how many times the value is to be repeated.

One word of caution: It is illegal to have an unsized number in a concatenation. Because you use a concatenation to create a specific number of bits, it would be pointless not to size a constant in a concatenation. However, if you have not established the habit of sizing the constants in your Verilog code, you will have problems with concatenation.

Concatenation can be used on both sides of an assignment. You can use concatenation to create a larger place for a result.

Example 8-7 Concatenations

```
// This is an incomplete example.
// The register declarations are
// included so you can see the size of
// these operands.
reg [3:0] a, b;  // Some 4-bit registers.
reg [7:0] c, d;  // Some 8-bit registers.
reg [11:0] e, f; // Some 12-bit registers.

c = {a,b}; // The most significant bit of c is the most
significant bit of a.
e = {b,a,b};
f = {3{a}}; // Three copies of a make 12-bits.
b = {4{e==f}} // Make a 4-bit mask of e==f.
f = {a,d}; // 4 bits + 8 bits = 12 bits.
e = {2{1'b1,a,1'b0}} // = 1aaaa01aaaa0,
                     // aaaa is the value of a.
{a,b} = d;
{a,b,c,d,e,f} = {f,e,d,b,c,a} + 1;
```

Table 8-16 Operator Order of Precedence

Operators	Description
`! ~`	Negation (highest precedence)
`* / %`	Arithmetic multiplication and division
`+ -`	Addition and subtraction
`<< >>`	Shifts
`< <= > >=`	Relational
`== === != !==`	Equality
`&`	Bitwise AND
`^ ^~`	Exclusive OR and exclusive NOR
`|`	Bitwise OR
`&&`	Logical AND
`||`	Logical OR
`? :`	Ternary (lowest precedence)

LOGICAL VERSUS BIT-WISE OPERATIONS

The set of logical operators includes all the relational operators: The logical AND (&&), logical OR (||), and negation (!). What distinguishes the logical operators from the bit-wise operators is the size of the value returned.

Bit-wise operators return a vector of the size of the operands (or destination, whichever is largest); logical operators return a single-bit value. The logical negation and relational operators have already been demonstrated in Example 8-2. The more confusing operators to look at are the *logical OR* (||) and *logical AND* (&&). Example 8-8 compares bit-wise and logical operators.

Example 8-8 Bit-wise and Logical Operations

```
Bitwise
      10101011
    & 01010101
     ==========
      00000001

Logical
      10101011
   && 01010101
   ==========
             ?
```

To solve the logical AND operation in Example 8-8, first convert the vector values to logical values. To convert a vector value to a logical value, ask this question: Is the value true or false? For a value to be true, it must have at least one bit that is 1.

In Verilog (as in other languages), only 0 is false. Because Verilog also includes unknown *x* and high impedance *z* as values, a logical value can also be unknown. If the vector contains *x's* or *z*'s but no 1s, then the logical value of the vector would be unknown. The convert-to-logical implicit conversion is similar to the reduction OR. See the reduction OR operation in Example 8-3 for some examples of convert-to-logical.

The logical AND operation from Example 8-8 is completed as follows:

Example 8-8 Results

```
    10101011  converted to logical ⇒ 1
 && 01010101  converted to logical ⇒ 1
                                    ===
                                     1
```

OPERATIONS THAT ARE NOT LEGAL ON REALS

The following operators are not legal on reals. Because a real is not treated as a vector of bits, several operators that work on bits and vectors of bits are not legal for use with reals. These operators are listed in Table 8-17.

Table 8-17 Operators Not Legal on Reals

Operator	Description
`{}`	Concatenation
`%`	Modulus
`=== !==`	Literal equality
`& | ~ ^`	Bitwise operators
`& |`	Bitwise reduction
`<<< << >> >>>`	Shifts

WORKING WITH STRINGS

Strings are stored in long registers. Each character in the string takes 8 bits. All the operators that work on registers also work on strings. Example 8-9 demonstrates the addition and concatenation of strings.

Example 8-9 Operators and Strings

```
module string2;
reg[8*13 : 1] s1,s2,s3;
initial begin
    s1 = "Hello";
    s2 = " Verilog";
    s3 = "abb";
    s3 = s3 + 1;
    if ( {s1,s2} != "Hello Verilog") begin
        $display("%s != %s", {s1,s2},
                        "Hello  Verilog");
        $display("%h != %h", {s1,s2},
                        "Hello Verilog");
    end
    $display("s3 = %s is stored as %h",
                s3, s3);
end
endmodule
```

As you can see from the results shown in Example 8-9 Results, when you concatenate two strings, the zero padding that was in the string remains in the resulting concatenation. Because strings are merely stored in long registers, addition to strings will increment the characters in the string, as shown in Example 8-9 Results.

Example 8-9 Results

```
Hello      Verilog != Hello Verilog
000000000000000048656c6c6f000000000020566572696c6f67   !=
48656c6c6f20566572696c6f67
s3 =            abc is stored as 00000000000000000000616263
```

COMBINING OPERATORS

You may be wondering why we are emphasizing exclusive NOR in Example 8-10. For two reasons: First, you can do things many ways in Verilog and this applies also to the exclusive NOR. The other reason why these examples are interesting is the order of precedence and sizing of the expressions.

Example 8-10 Combinations of Operators for Exclusive NOR

```
reg [7:0] a, b;  // Some eight-bit registers.
reg [8:0] r9;  // a nine-bit register
  r9 = ~ (a ^ b ) ; // Exclusive NOR of a and b
  r9 = a ^~b ;    //most significant bit is '1'.
  r9 = a ~^b ;
  r9 = ~a ^ b;
```

SIZING EXPRESSIONS

The most significant bit of all the exclusive NOR examples in Example 8-10 is 1. This is because 0 XNOR 0 = 1 . The sizing of the expression is done by first expanding all of the operands to the largest size of the operands and destination. Once all the operands have been expanded, the value is computed. If the destination is smaller than the size of the computed value, the value is truncated. Verilog never zero fills after computing the result. So, in Example 8-10, the two 8-bit values *a* and *b* are expanded to 9-bit values with the most significant bit 0. Finally, Verilog performs the operations to generate the 9-bit result.

SIGNED OPERATIONS

Nets, *regs*, and times in Verilog are unsigned; only *integer* and *real* types are signed by default. The IEEE 1364-2001 standard enhances net, reg, port and constant declarations to allow signed values other than *integer* and *real.* An operation is sign extended when the operands involved are signed. Example 8-11 shows the *signed* key word added to various declarations. Signed values use 2's complement format.

Example 8-11 Signed Declarations

```
module signunsign(a,b,c,d);
  input [7:0] a;         // unsigned
  input signed [7:0] b;  // signed
  output [7:0] c;        // unsigned
  output signed [7:0] d;  //signed

  wire signed [7:0] e;   // signed.
  reg signed [7:0] f;    // signed.
  reg [7:0] g;           // unsigned

endmodule
```

Signed Constants

Chapter 2 introduced the syntax for specifying the radix of constants in Verilog. The ability to use signed math dictates the need for signed constants. The letter *S* can be added between the apostrophe and radix letter to indicate a signed constant as with the radix specifiers of Chapter 2, the *S* may be lower case or capitol.

Example 8-12 Signed Constants

```
4'shf   //  '-1'
8'sb11111111 // '-1'
8'sb01111111  //127
-4'sb1111 // 1  because -(-1)
```

Table 8-18 shows all radix specifiers. The effect of signed constants can be seen in Example 8-13.

Example 8-13 shows three ways to write the expression "minus 12 divided by 3." Note that *-12* and *-'d12* both evaluate to the same 2's complement bit pattern, but, in an expression, the *-'d12* loses its identity as a signed negative number.

Table 8-18 Radix Specifiers

Radix Mark	Radix
'b 'B	Binary
'sb 'Sb 'sB 'SB	Signed Binary
'd 'D	Decimal (default)
'sd 'Sd 'sD 'SD	Signed Decimal
'h 'H	Hexadecimal
'sh 'Sh 'sH 'SH	Signed Hexadecimal
'o 'O	Octal
'so 'So 'sO 'SO	Signed Octal

Example 8-13 Effect of Signed Constants

```
integer I;
  I = -12 / 3;       // The result is -4.
  I = -'d 12 / 3;    // The result is 1431655761.
  I = -'sd 12 / 3;   // The result is -4
  I = -4'sd 12 / 3;  // -4'sd12 is the negative of
                     // the 4-bit quantity 1100,
                     // which is -4. - (-4) = 4 .
```

This Page Intentionally Left Blank

9 CREATING COMBINATORIAL AND SEQUENTIAL LOGIC

So far you have learned structural modeling and enough high level code to apply stimulus and to display results from your circuits. You have also read about Verilog's rich set of operators and data objects to use as operands. In this chapter you will learn how to use the operators to model circuits at a higher level of abstraction than merely structural. At the end of this chapter is an exercise based on the operators introduced in Chapter 8, and the high level constructs presented in this chapter.

CONTINUOUS ASSIGNMENT

The *continuous assignment* is the simplest of the high level constructs. A continuous assignment is just like a gate: It drives a value out onto a wire. A *continuous assignment* is different from a *procedural assignment* in a few ways. First, the destination (left-hand side, or LHS) is always a wire. Second, the continuous assignment is automatically evaluated when any of the operands change. Unlike a procedural assignment, the continuous assignment cannot occur in a block of sequential code. The *continuous assignment* is always a module item by itself. Finally, a continuous assignment always models combinatorial logic. It is true that you can create logic that feeds back into itself and mimics storage, but still it is combinatorial.

Example 9-1 shows a simple 16-bit, three-state buffer using a continuous assignment.

Example 9-1 Three-State Buffer Using a Continuous Assignment

```
module buf16(out,in,enable); // 16-bit, three-state buffer
input [15:0] in;
output [15:0] out;
input enable;
assign out = enable ? in : 16'bz;  // Continuous assignment
endmodule
```

In Example 9-1, the wire out gets either *in* or *z* depending on the value of *enable*. Whenever *enable* or *in* changes, the continuous assignment is evaluated and a new value for *out* is calculated.

The continuous assignment can be used with all the operators to create a result of any size. A single continuous assignment can quickly model large combinatorial circuits. For example consider, the small modules in Example 9-2 and Example 9-3

Example 9-2 A 128-Bit Adder In a Continuous Assignment

```
module add128(cout, sum, a, b, cin);
             // 128 bit adder
input [127:0] a, b;
input cin;
output [127:0] sum;
output cout;
/* This continuous assignment models hundreds
   of gates. The MSB of the add, carry is
   assigned to cout by making the addition
   in 129 bits using a concatenation on the LHS.
*/
assign {cout,sum) = a + b + cin;
endmodule
```

The continuous assignment in Example 9-2 uses a concatenation on the left-hand side of the assignment to catch the carry-out bit.

Example 9-3 Continuous Assignment Multipiler

```
module mul64(prod, a, b); // Simple multiplier
input [31:0] a, b;
output [63:0] prod;
assign prod = a * b; // Thousands of gates !!!
endmodule
```

The continuous assignment is a quick and easy way to model when the combinatorial logic can be expressed as a simple equation. A simple buffer (for example, *assign a=b*;), a mux using the ternary operator, an arithmetic function, or a complex set of Boolean operators. These can all be modeled using the continuous assignment.

Before leaving the continuous assignment, see Table 9-1 to compare the two types of assignments you've learned so far.

Table 9-1 Comparison of Procedural and Continuous Assignments

Assignment Type	LHS	When Evaluated	Where in module
Procedural	reg	When encountered	Procedural block
Continuous	net	Whenever RHS changes	On its own

LHS = left-hand side (the destination of the assignment);

RHS = right-hand side (the operands in the assignment)

The continuous assignment can be used to connect a register or several registers to a net. Consider Figure 9-1.

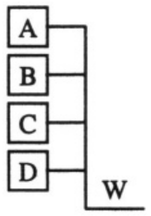

Figure 9-1 Connecting Four Registers to a Wire

Connecting four registers to wires as shown in Figure 9-1 can easily be modeled in Verilog, as shown in Example 9-4.

Example 9-4 Connecting Four Registers to a Wire

```
module regnet;
reg a, b, c, d;
wire w;

assign w=a;
assign w=b;
assign w=c;
assign w=d;

endmodule
```

An alternate form of the continuous assignment may be written when a wire is declared. Example 9-5 shows a simple continuous assignment where the wire *c* has the value *a* | *b*.

Example 9-5 Alternate Form of Continuous Assignment

```
module aca;
reg a, b; .
wire c = a | b; // shorthand continuous assignment
endmodule
```

Continuous assignments may also have delays. Multiple continuous assignments may be combined in one statement and separated by commas. Example 9-6 shows a few more combinations of continuous assignments.

Example 9-6 Many forms of Continuous Assignments

```
module mca;
reg a, b, c, d;
wire y, yb, a1, a2;
wire [3:0] bus = {a, b, c, d};
wire #(3,2) parity = ^bus;
assign #1 a1 = a & b,
          a2 = c & d,
          y=a1|a2,
          yb = ~y;
endmodule
```

Example 9-6 shows a continuous assignment as a wire declaration that combines the four registers into a bus. The next continuous assignment generates parity on the bus with a rise delay of 3 and a fall delay of 2. The final set of continuous assignments forms an AND-OR-INVERT gate with a total *a-to-y* delay of 3 because each of the continuous assignments has a delay of 1.