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.

Created in the United States of America

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 cover 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.)

System
Architectural
Behavioral
Algorithmic
Register Transfer Level (RTL)
Boolean Equations
Structural
Gates
Switches
Transistors
Polygons
Masks

More
Abstract

More
Detailed
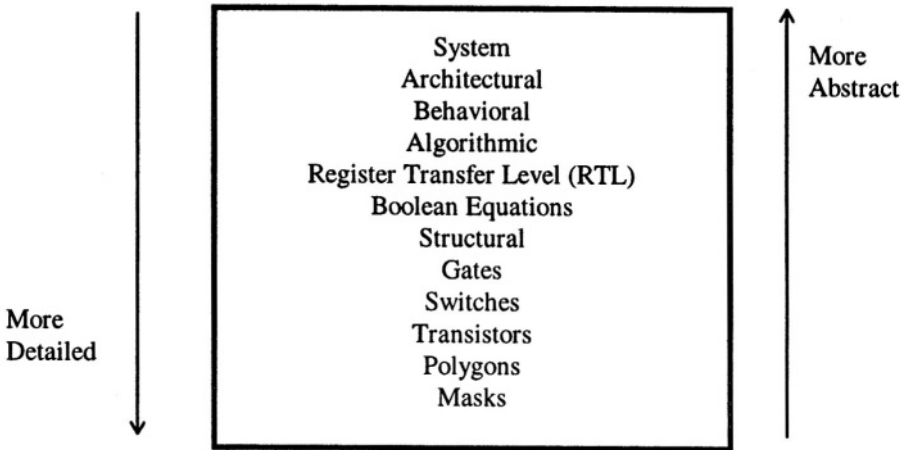
Figure 1-1 Design Abstraction Hierarchy

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
```
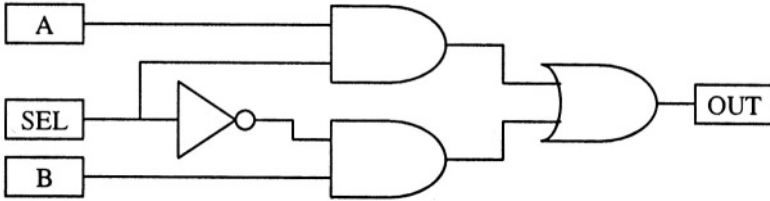

Figure 1-2 Gate-Level Model Mux Schematic

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.
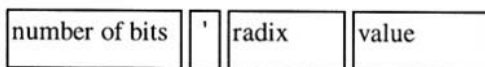
Figure 2-1 Number Format

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.*
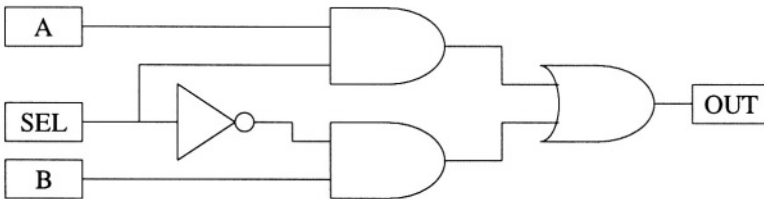
Figure 2-2 The Mux Example

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.
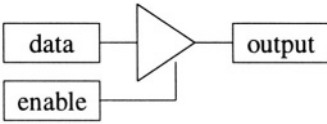
Figure 2-3 Three-State Buffer

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.
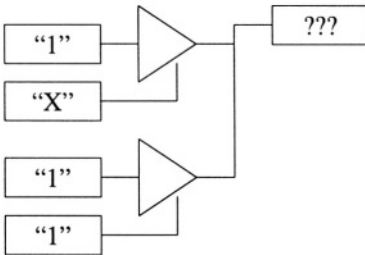
Figure 2-4 Two Three-State Buffers

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

This Page Intentionally Left Blank

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.
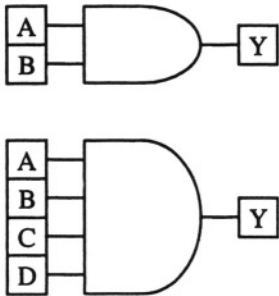
Figure 3-1 AND Gate Primitives

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.
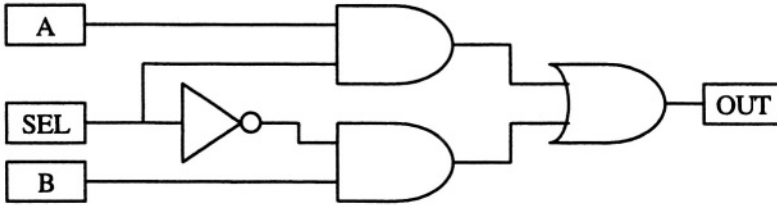
Figure 3-2 Gate-Level Model Mux Schematic

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.
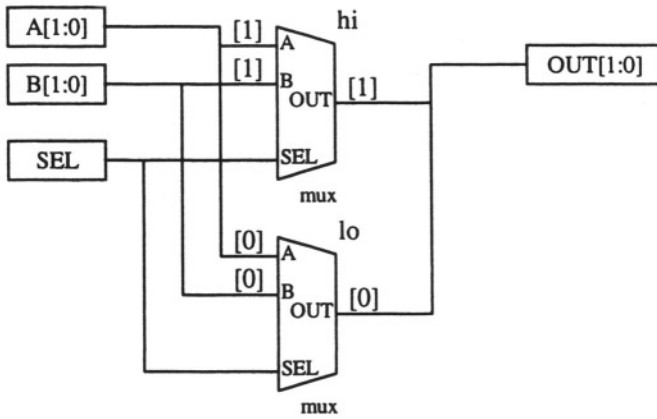
Figure 3-3 Connecting Two Muxes

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.
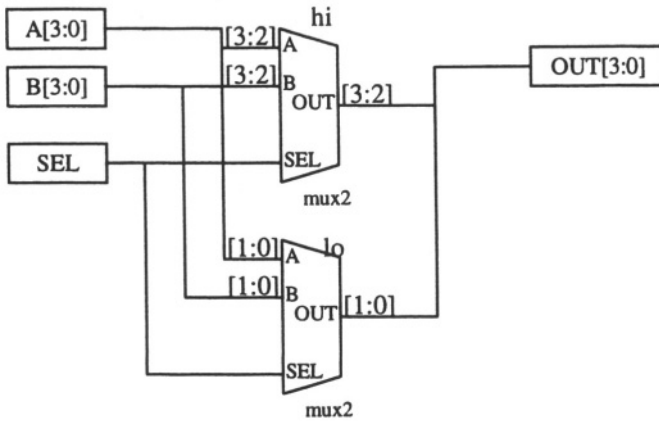
Figure 3-4 Hierarchical 4-Bit Mux

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.
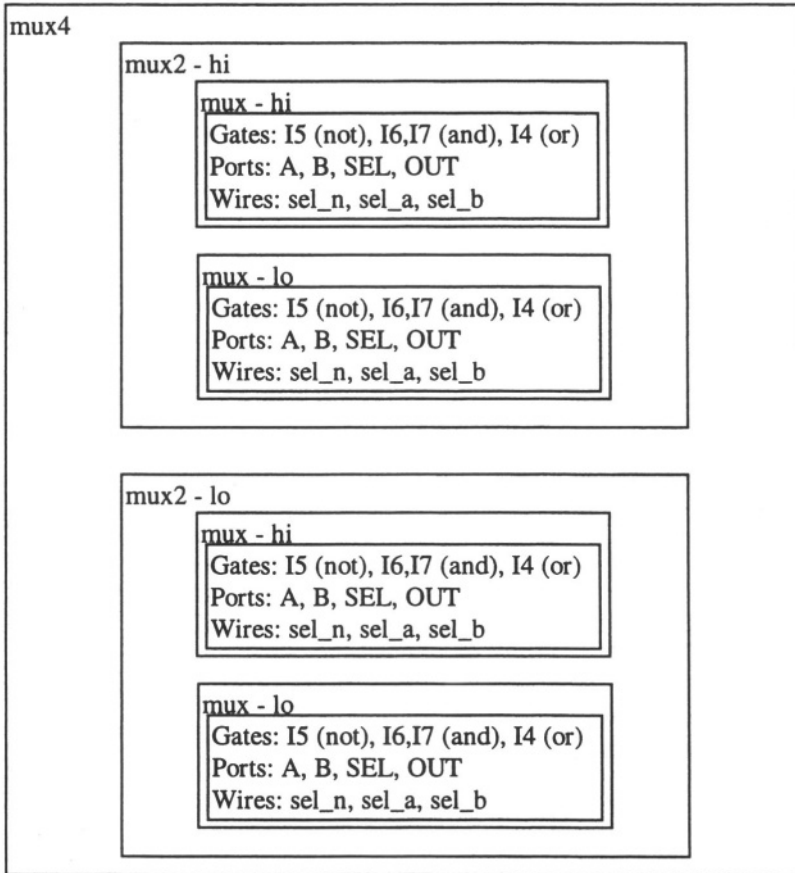
Figure 3-5 Mux4 Hierarchy Expanded

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*)
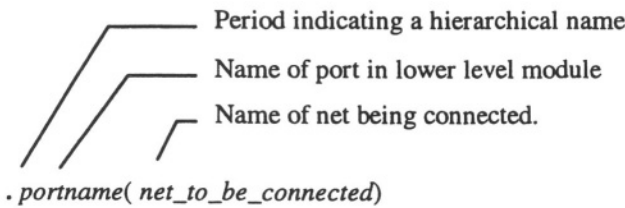
Figure 3-6 Syntax for Connect By Name

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.
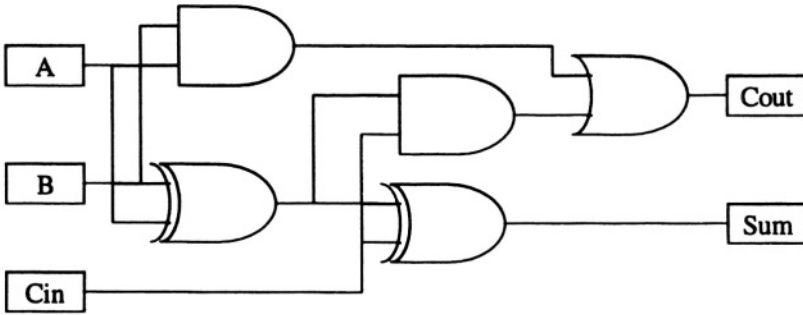
Figure 3-7 *Adder* Schematic

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.*
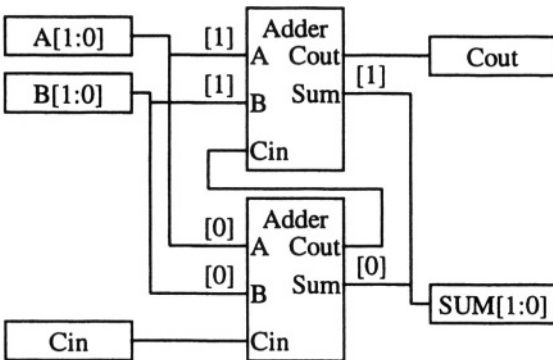
Figure 3-8 *Adder2* Schematic

Connect two of the *adder2s* together to form a 4-bit adder as shown in Figure 3-9.
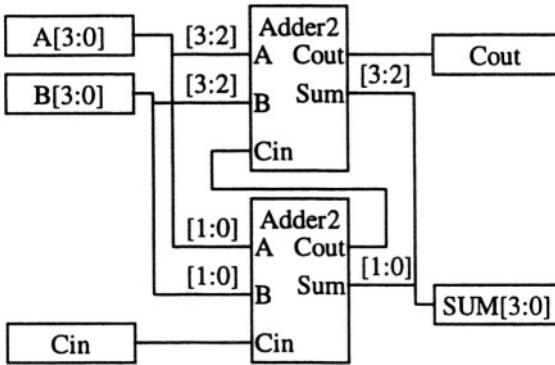
Figure 3-9 *Adder4* **Schematic**

Connect two of the *adder4s* together to form a 8-bit adder as shown in Figure 3-10.
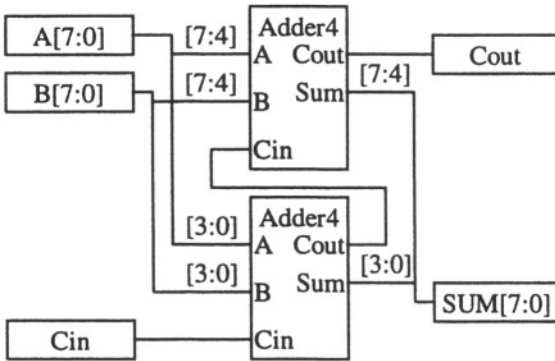
Figure 3-10 *Adder8* **Schematic**

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 12f;

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.

$timeformat(-9, 2, "ns", 7);
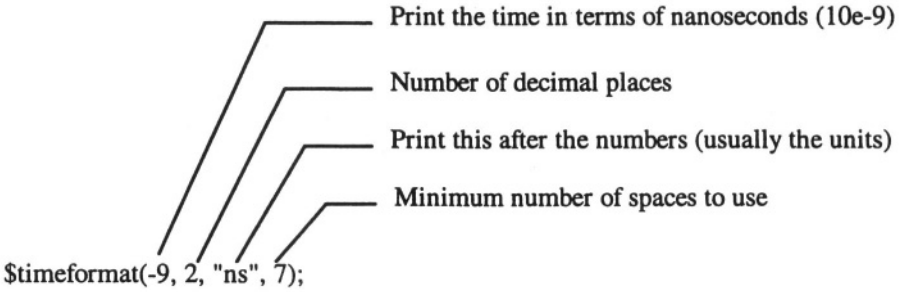
Figure 5-1 Time format details

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 *tri1* net type to model this situation. If nothing is driving a net of type *tri1,* the default value is 1. As with *tri1,* 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 tril;
`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;
tril [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.
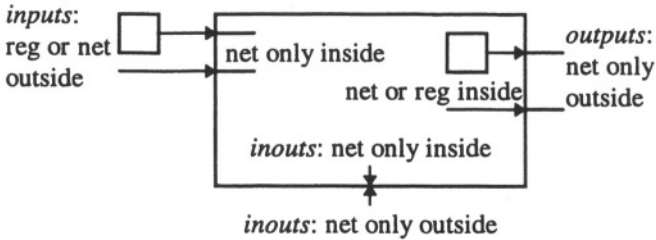
Figure 6-1 Relationships of ports and regs

This Page Intentionally Left Blank

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] = 'b1100llzz;
  example[5] = 'b11111l1x;
  $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 1100llzz, & = 0, | = 1, ^ = x
Value 11111l1x, & = 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-l 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] sl,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
0000000000000000048656c6c6f00000000020566572696c6f67   !=
48656c6c6f20566572696c6f67
s3 =            abc is stored as  0000000000000000000000616263
```

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.
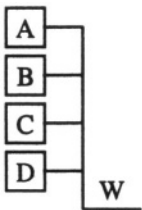
Figure 9-1 Connecting Four Registers to a Wire

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.

EVENT CONTROL

For controlling the flow of execution through procedural code, only the delay operator "#" has been introduced so far. The # is sufficient if you know how much time you want to delay. But what if you want to delay until the change on a signal? The change (or edge) operator is the "@" ("at" sign). The @ operator is best described as "wait for event." The event that the @ waits for is normally any change on a signal. The @ can also be used with the *event* data type. In addition to @, there are three related keywords for working with edges or events: *posedge* and *negedge* for selecting only one edge, and *or* for waiting for a change on more than one signal.

The simplest common model using the @ is a D flip-flop. How does a flip-flop work? The flip-flop waits for the rising edge of the clock, then the output *q* gets the value of *d*. In Verilog, a D flip-flop can be described as shown in Example 9-7.

Please note that though Example 9-7 presents an accurate description of a D flip-flop, this would not be the most efficient way to model a flip-flop. The model in Example 9-7 would be better used to model a wide register, for example, a register of 64 or 128 bits. Such a register could be modeled like this just by changing the width of *d* and *q*. If you wanted to model a single flip-flop, you might be better off with a user-defined primitive. User-defined primitives are introduced in Chapter 13.

Example 9-7 Waiting for an Event

```
`define REG_DELAY 1
module dff(clock, d, q);
input d, clock;
output q;
reg q;
always @(posedge clock)
    q <= # (`REG_DELAY) d;
endmodule
```

The behavior of the continuous assignment can be mimicked by using the *wait for event (@)* on multiple signals. By looking for all the signals to change, the *always* can be used to model combinatorial logic. Example 9-8 and Example 9-9 compare a mux with a continuous assignment and a mux with an *always* block.

Example 9-8 Mux Using Continuous Assignment

```
module muxca(a,b,sel,y);
          // mux with continuous assignment
input  a,b,sel;
output y;

assign y = sel ? a : b;

endmodule
```

Example 9-9 Mux Using *always* Block

```
module muxae(a,b,sel,y); // mux with always
input  a,b,sel;
output y;
reg y;

always @(a or b or sel)
    if (sel)
        y = a;
    else
        y = b;

endmodule
```

Both of the muxes in Example 9-8 and Example 9-9 behave identically for *sel = 1* or *0.* However, they behave differently when *sel = x.* They will synthesize to similar logic.

The *always* Block for Combinatorial Logic

Example 9-9 shows how the *always* block can be used to model combinatorial logic. If the code in an *always* block executes whenever its inputs change, it will model combinatorial logic. The *or* keyword is used to separate a list of events for the *wait for event* (@) operator. The list of events is often called a *sensitivity list.* The IEEE 1364-2001 standard adds two new shorter ways to specify the sensitivity of an *always* block; a comma ',' can be used instead of the *or* keyword to separate a sensitivity list, as shown in Example 9-10.

The new standard also defines the combinatorial sensitivity list using the '@*' as shown in Example 9-11. As with any of the 2001 updates to the Verilog Language, the comma and star for sensitivity lists may not be supported by all tools.

Example 9-10 *always* Block Using Comma

```
module muxaec(a,b,sel,y);  // mux with always using comma
input a,b,sel;
output y;
reg y;

always @(a, b,  sel)
    if  (sel)
        y = a;
    else
        y = b;

endmodule
```

Example 9-11 Combinatorial *always* Block

```
module muxaes(a,b,se1,y);  // mux with always using star
input a,b,sel;
output y;
reg y;

always @*
    if  (sel)
        y = a;
    else
        y = b;

endmodule
```

Event Control Explained

The mux shown in Example 9-12 anct only when *sel* changes, so is not an operating mux and can be built in hardware.

Example 9-12 Incorrect Mux

```
module bad_mux(a,b,sel,y);   // bad model
input a,b,sel;
output y;
reg y;

always @sel
    if  (sel)
        y = a;
```

```
    else
        y = b;
endmodule
```

Note that the parentheses are not needed with @ if you are only looking for any change on a single signal. The parentheses are only necessary if you are going to use an expression after @.

One final thing to remember about the *wait for event (@):* Don't interpret "*always @A*" as "whenever *A* changes." Remember that *always* is defined as a loop that starts at time 0, then when it finishes, it starts over again. So "*always @ A*" describes the situation of waiting for *A,* and having *A* change. Consider Example 9-13.

Example 9-13 *always* Explained

```
module  not_always;
reg clock;
always  begin
    #5 clock = 0;
    #5 clock = 1;
end

always @(posedge clock)begin
    $display("clock edge at %0d",  $time);
    #11 $display ("waiting for the clock");
end
endmodule
```

In Example 9-13, the second *always* block sees the first rising edge at time 10. The delay of 11 between the two *$display* statements expires at time 21. In the meantime, there was a rising clock edge at time 20 that is missed. This demonstrates that the *always @(posedge clock)* does not consistently mean whenever the clock rises. It means whenever the clock rises and we are waiting for it.

The @ can be thought of as an edge-sensitive event control. There is also a level-sensitive event control: *wait*. The *wait* statement will not block execution if its condition is true. If the condition is not true, it will wait until it becomes true. Consider Example 9-14:

Example 9-14 Using *wait*

```
module  wait_example;
reg  [7:0]  a;
reg  b;

initial  begin
    wait  (a==3)
        $display("not waiting for a==3 time %0d", $time);
    wait(b)
        $display("not waiting for b time %0d",  $time);
    wait  (a==4)
        $display("not waiting for a==4 time %0d", $time);
end

initial  begin
    #3 a = 3;
        $display("value of a is now %0d at time %0d", a,$time);
    #1 a = 4;
        $display("value of a is now %0d at time %0d",a,$time);
    #1 b = 1;
        $display("value of b is now %0d at time %0d",b,$time);

end
endmodule
```

In Example 9-14, the first *initial* block starts and waits until *a* becomes 3, which happens at time 3 from the second *initial* block. The first *initial* block then continues and prints its first message at time 3. At time 4, *a* becomes 4, but nothing else happens, because now the first *always* block is waiting for *b*. Waiting for *b* by itself is equivalent to waiting for *b* to become non-0, *b* becomes 1 at time 5, so the second message is printed. The final *wait* waits for *a* to become 4, but *a* is already 4. Therefore, the *wait* does not block execution and the third message is also printed.

Example 9-14 Results

```
value of a is now 3 at time 3
not waiting for a==3 time 3
value of a is now 4 at time 4
value of b is now 1 at time 5
not waiting for b time 5
not waiting for a==4 time 5
```

While the code in Example 9-14 does not show a great use for the *wait,* it does show how *wait* works. Example 9-15 presents some code that shows a more intelligent use of *wait.*

Example 9-15 Using *wait* To Detect an Unknown

```
initial wait (^data_bus === 1'bx) begin
    $display("Error unknown on the data bus!");
    $stop;
end
```

Example 9-15, which uses the *wait* statement, is more efficient than Example 9-16, which uses @. If you are wondering how the condition *(^data_bus === 1'bx)* checks for an unknown on the data bus, here is how it works. If you use the reduction exclusive OR operator on an unknown entity, the result is unknown. Thus using the reduction exclusive OR on the entire data bus yields an unknown if any of the bits are unknown. Example 9-16 illustrates a less efficient example of checking for an unknown on a bus.

Example 9-16 Using *always* To Detect an Unknown

```
always @databus
    if (^data_bus === 1'bx) begin
        $display("Error unknown on the data bus!");
        $stop;
    end
```

Summary of Procedural Timing

One of the most important concepts in Verilog modeling is knowing *when* a procedural statement will be run. The remaining key words and symbols that indicate when a procedural statement will be run were just introduced. A common cause for incorrect model behavior and even syntax errors is incorrectly specifying, or omitting, an indication of when your code should be run. If you don't know when your code should be run, perhaps the simulator or synthesis tool may have the same difficulty.

Table 9-1 summarizes the keywords presented to determine when a statement will be run.

Table 9-1 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.
@	Edge sensative, wait for something to change.
posedge	Modifier for '@' change 0-1 or 0-x or x-1.
negedge	Modifier for '@' change 1-0 or 1-x or x-0.
wait()	Level sensative delay, if the condition is true, continue with no pause, if false pause until true.
assign	Continuous (concurrent) assignment to net, evaluated when any input changes.
->	Used only with the event data type to signal that an abstract event has occurred.

Remember that with the '@', procedural flow must be waiting at the '@' for the event to be seen. Other delays or event controls in a loop or block of code may effect when you are waiting at the '@'.

Also remember that an *always* loop with no delay or event controls is a zero delay loop. A zero delay loop does not allow simulation time to advance and your simulation can hang. An *always* with only *wait* statements or a path to avoid the delay or event controls may potentially be a zero delay loop.

This Page Intentionally Left Blank

10 PROCEDURAL FLOW CONTROL

Verilog has a rich set of procedural statements that can be used for modeling combinatorial logic or sequential logic. This chapter explains the *if, case* and looping constructs.

THE *IF* STATEMENT

A few of the examples from previous chapters have shown the *if* (statement. The *if* in Verilog has a few quirks. You may have noticed that in Verilog, the *if* has no corresponding *then* or *endif*. Rather, Verilog has an *else*. The simple form of *if* is shown in Example 10-1. An *if* with *else* is shown in Example 10-2.

Example 10-1 Simple *if*

```
if (condition)
    statement
```

Example 10-2 *if* with *else*

```
if (condition)
    statement
else
    statement
```

Each branch of an *if* can only have a single statement. If you need more than one statement, then you need to use a *begin-end* block or *a fork-join* block.

The condition can be any expression, or even just a single value. If the condition evaluates to 0 or unknown, then the condition is considered false, and the *if* clause is not executed. Thus, if the condition evaluates to 1 or more, the *if* clause is executed, but if it evaluates to 0 or unknown, the *else* clause executes if it is present.

Because the *if has* no *endif,* it is easy to become confused when there are multiple *if's* with *else's.* Consider Example 10-3: Which *if* does *else* go with?

Example 10-3 Nested *if* with *else*

```
if (A)
    if (B) do_something;
else
    just_do_it;
```

Under what condition does the design "just do it"? Does this happen when *A* is not true, and does the indenting make the result appear? The answer is: When *A* is true and *B* is not true. Why? An *else* is always associated with the immediately preceding *if* that does not have an *else* associated with it. In Example 10-3, the *else* is associated with the latest *if,* which is the *if(B).*

The *if* or *else* clauses may use a *begin-end* or *fork-join* when multiple statements are needed.

Remember, for combinatorial logic with an *if* should have an *else* or you might imply a latch. Sequential logic with an *if* statement does not need an *else* if you intend the *reg* to hold its old value.

THE *CASE* STATEMENT

Verilog has three varieties of *case* statement: *case, casex,* and *casez.* The *case* statement is a convenient way of decoding multiple conditions or opcodes. Example 10-4 illustrates a simple 4-to-1 mux using a *case* statement.

Example 10-4 The *case* Statement

```
module mux4a(y,a,b,c,d,sel) ;
input a,b,c,d;
input [1:0] sel;
output y;
reg y;

always @(a or b or c or d or sel)
    case ( sel )
        0: y = a;
        1: y = b;
        2: y = c;
        2'b11 : y = d;
        default : y = 1'bx;
    endcase
endmodule
```

A *case* statement begins with the expression *case* and ends with *endcase*. The expression being compared is in parentheses following the keyword *case*.

Each of the *case* items (possible matches for the expression being compared to) is denoted by a colon and followed by a single statement. If more than one statement needs to be executed for a particular *case* item, enclose the statements in a *begin-end* block or *fork-join* block. When one case item statement has executed. Flow automatically resumes at the *endcase,* this eliminates the need for a *break* statement like C uses. If you want multiple conditions to execute the same statement, you can separate them with commas before the colon. Only the first condition that is true is executed. The keyword *default* can be used to catch any of the conditions not explicitly stated.

In Example 10-4 for the first three *case* items, a decimal number was used. This is acceptable, but inelegant. For the third *case* item, a more fully specified number in which also declared the number of bits to be the same as the size of the *case* expression (2 bits) was specified. The final *case* item, *default,* actually catches twelve possible *case* items that exist when either or both of the select lines are unknown or high impedance. This is shown in Example 10-5.

Example 10-5 *case* **Matching** *x* **and** *z*

```
module mux4b(y,a,b,c,d,sel);
input a,b,c,d;
input [1:0] sel;
output y;
reg y;

always @(a or b or c or d or sel)
    case ( sel )
        2'b00 : y = a;
        2'b01 : y = b;
        2'b10 : y = c;
        2'b11 : y = d;
        2'b0x, 2'b1x, 2'bzx, 2'bxx,
        2'bx1, 2'bx0, 2'bxz, 2'bzz,
        2'bz0, 2'bz1, 2'b1z, 2'b0z : y = 1'bx;
    endcase
endmodule
```

In the *case* statement, as with the === operator, *x* matches *x* and *z* matches *z*.

Although *case* does not allow wildcards for matches, the *casex* and *casez* do. If you use *casez*, *z* or *?* can be used to match any value.

Let's model a simple loadable counter. This counter has a *load* and a *reset,* both of which are active on the rising edge of the clock, *reset* has priority over *load,* and if *load* is not set, then the counter counts up or down based on the state of the *up* signal. This could be modeled with some *if-else* statements, but Example 10-6 uses *casez.*

Example 10-6 Using *casez*

```
`define REG_DELAY 1
module counta(clock, reset, load, up, load_data, count);
input clock,reset, load, up;
input [15:0] load_data;
output [15:0] count;
reg [15:0] count;

always @(posedge clock)
    casez ({reset,load,up}) // concatenate control signals
        3'b1zz : count <= 16'h0000;
        3'b01? : count <= #(`REG_DELAY) load_data;
        3'b001 : count <= #(`REG_DELAY) count + 16'h0001;
        3'b000 : count <= #(`REG_DELAY) count – 16'h0001;
        default : count <= #{`REG_DELAY) 16'bx;
    endcase
endmodule
```

In Example 10-6, both *z* and *?* are used as wildcards.

Note that *casex* differs from *casez* in the don't care values. In *casez, z* and *?* represent don't cares. However, in *casex,* only *x* and *z* are both don't care. Table 10-1 summarizes which values will be matched by each type of case statement.

Table 10-1 Summary of Case Values and Match per Case Type

Value	Case	Casez	Casex
0	0	0	0
1	1	1	1
x	x	x	0 1 x z
z	z	0 1 x z	0 1 x z
?	unused	0 1 x z	unused
default	0 1 x z	0 1 x z	0 1 x z

The next two examples compare the loadable up-down counter implementations. Example 10-7 uses the case statement. Since the *case* matches each bit exactly, it is necessary to specify many more case items for the reset and load operations. Example 10-8 repeats the counter using an *if* statement.

Example 10-7 Counter Using *case*

```
`define REG_DELAY 1
module countb(clock, reset, load, up, load_data, count);
input clock,reset, load, up;
input [15:0] load_data;
output [15:0] count;
reg [15:0] count;

always @(posedge clock)
    case ({reset,load,up})  // concatenate control signals
        3'b100, 3'b101, 3'b10x, 3'b10z,
        3'b110, 3'b111, 3'b11x, 3'b11z,
        3'b1x0, 3'b1x1, 3'b1xx, 3'b1xz,
        3'b1z0, 3'b1z1, 3'b1zx, 3'b1zz
                  : count <= #(`REG_DELAY) 16'h0000;
        3'b010, 3'b011, 3'b01x, 3'b11z
                  : count <= #(`REG_DELAY) load_data;
        3'b001 : count <= #(`REG_DELAY) count + 16'h0001;
        3'b000 : count <= #(`REG_DELAY) count - 16'h0001;
        default : count <= #(`REG_DELAY) 16'bx;
    endcase
endmodule
```

Example 10-8 Counter Using *if*

```
`define REG_DELAY 1
module countc(clock, reset, load, up, load_data, count);
input clock, reset, load, up;
input [15:0] load_data;
output [15:0] count;
reg [15:0] count;

always @(posedge clock)
    if (reset)
        count <= 16'h0000;
    else
        if (load)
            count <= #(`REG_DELAY) load_data;
        else
            if (up)
                count <= #(`REG_DELAY) count + 16'h0001;
            else
                count <= #(`REG_DELAY) count - 16'h0001;
endmodule
```

LOOPS

The only loop that has been used so far in this book is the *always* loop. The *always* loop combines a starting point for execution with an infinite loop. The rest of the looping constructs do not imply starting places for execution. They must be contained in a procedural block of code so they have a starting point.

The *forever* Loop

The simplest loop is the *forever* loop. The *forever* loop is an infinite loop. The construct *initial forever* is almost identical in behavior to *always*. *The forever* loop contains a single statement. If you want multiple statements in the *forever* loop, use a *begin-end* block. Like the *always* statement, *a forever* loop without delay or event controls would be a zero-delay loop and inhibits simulation time from advancing. Even though the *forever* loop is an infinite loop, there are two ways to stop it. The first way to stop *a forever* loop is with a *$finish* that ends simulation. The second way is to use the *disable* statement, which will be presented later in this chapter.

Example 10-9 and Example 10-10 compare the *forever* and *always* loops.

Example 10-9 Oscillator Using *always*

```
module osc1(clock);
output clock;
reg clock;
initial begin
    clock = 0;
    end
always begin
    #50 clock = ~clock;
    end
endmodule
```

Here is another version of the oscillator module using the *forever* loop.

Example 10-10 Oscillator Using *forever*

```
module osc2(clock);
output clock;
reg clock;
initial begin
    clock = 0;
    forever #50 clock = ~clock;
    end
endmodule
```

While the *initial forever* and the *always* loop are quite similar. Most synthesis tools do not synthesize the *forever* loop. The forever loop should only be used in behavioral models not intended for logic implementation or test benches.

The *repeat* Loop

The *repeat* loop is the next simplest of the looping constructs in Verilog. The *repeat* loop takes some expression that will evaluate to an integer, and repeats the loop that many times. The *repeat* loop is similar to the *forever* loop in that it uses a single statement or a *begin-end* block.

Example 10-11 is a slight modification of the *hello* module in which the module prints "Hello Verilog" five times.

Example 10-11 Repeating "Hello Verilog"

```
module hellor;
initial repeat(5) $display("Hello Verilog");
endmodule
```

The *repeat* loop has uses in state machines, such as the simple shifter state machine shown in Example 10-12.

Example 10-12 Using *repeat* in a State Machine

```
`define REG_DELAY 1
module shift1(clock,start,data,s,out,done);
input clock,start;
input [15:0] data;
input [3:0] s;
output [15:0] out;
reg [15:0] out;
output done;
reg done;
always @(posedge clock)
    if (start) begin
        done <= #(`REG_DELAY) 1'b0;
        out <= #(`REG_DELAY) data;
        repeat (s) // number of times to shift
            @(posedge clock) out <= #(`REG_DELAY) out << 1;
        @(posedge clock) done <= #(`REG_DELAY) 1'b1;
    end
endmodule
```

Please note that this is not the most simulation-efficient way to implement this shifter. Chapter 17 discusses performance issues and other shifter models will be presented.

Depending on the usage and synthesis tool, the *repeat* loop may be synthesizable.

The *while* Loop

The *while* loop is executed as long as its condition is true. Example 10-13 counts the number of 1s in an input signal.

Example 10-13 A *while* Loop

```
`define REG_DELAY 1
module onecount(clock,start,data,count,done);
input clock,start;
input [15:0] data;
reg [15:0] temp_data;
output [3:0] count;
reg [3:0] count;
output done;
reg done;
always @(posedge clock)
  if (start) begin
    done <= #(`REG_DELAY) 1'b0;
    count <= #(`REG_DELAY) 1'b0;
    temp_data = data;
    while(temp_data) begin// continue as long as non zero
      @(posedge clock)
      if (temp_data[0]) count <= #(`REG_DELAY) count + 1;
      temp_data = temp_data >> 1;
    end
    @(posedge clock)
    done <= #(`REG_DELAY) 1'b1;
  end
endmodule
```

Depending on the usage and synthesis tool, the *while* loop may be synthesizable.

The for Loop

The expression for the *for* loop is a bit more complicated. The *for* loop uses three expressions separated by semicolons to control the loop. The first expression (the initialization expression) is executed once before entering the loop the first time. The second expression (the loop condition) is evaluated to determine if the contents of the loop should be executed. If the loop condition expression is true, the loop is entered. The final expression (the increment expression) is evaluated at the end of the loop.

Example 10-14 is another version of the *hello* module with a *for* loop. Notice the expression $i=i+1$ in the loop. Those familiar with programming in C might try the ++ unary operator. A quick review of Chapter 8 will remind you that Verilog, unlike C, does not have a ++ operator.

Example 10-14 A Simple *for* loop

```
module hellof;
integer i;
initial for(i=0; i<5; i=i+1)
    $display("Hello Verilog %0d", i);
endmodule
```

Because the expressions in the *for* loop all don't need to reference the same register or variable, the *for* loop in Example 10-15 is also legal.

Example 10-15 A *for* Loop with Expressions Not Referencing the Same Variable

```
module for2;
reg [7:0] a,b,c,d;
initial begin
    c = 9;   // If c==0 the loop will not enter.
    b = 3;
    d = 0;
    for(a = b; c; d=a) begin
        a = a + d;
        c = c - b;
    end
    $display("a= %d, b=%d, c= %d, d=%d, OK?", a, b, c, d) ;
end
endmodule
```

Depending on the usage and synthesis tool, the *for* loop may be synthesizable.

With the high level code you have just learned, you can now start to model more complex behavior. Take a break from the reading and try some exercises to practice what you have just learned.

Exercise 4 Using Expressions and case

In this exercise you will model a simple arithmetic logic unit (ALU). This will give you a chance to practice using text macros, the *always* statement, the *case* statement, and some expressions. The ALU has sixteen functions. These are described below. The ALU has three inputs: The *a* and *b* operands (16 bits), and the function select input (4 bits). The ALU has four outputs: *aluout* (the 16-bit result), a *zero* flag; a *parity* flag, and a *carry* flag. The ALU is combinatorial. The *carry* flag is the seventeenth bit of the result of addition, subtraction, shifts, and rotates. All of the operations generate a result in the *carry*. The *zero* flag is set if all the low 16 bits of *aluout* are 0. The *parity* flag is set if an odd number of bits is set in *aluout.*

Here are the opcodes for the ALU:

```
`define move  4'b0000 /* out = b */
`define comp  4'b0001 /* out = complement(b) */
`define and   4'b0010 /* out = a AND b */
`define or    4'b0011 /* out = a OR b */
`define xor   4'b0100 /* out = a XOR b */
`define add   4'b0101 /* out = a PLUS b */
`define incr  4'b0110 /* out = b PLUS 1*/
`define sub   4'b0111 /* out = a MINUS b */
`define rotl  4'b1000 /* out = rotate b left*/
`define lshl  4'b1001
                  /* out = logical shift left*/
`define rotr  4'b1010
            /* out = rotate b right one bit */
`define lshr  4'b1011
      /* out = logical shift b right one bit*/
`define xnor  4'b1100 /* out = a XNOR b */
`define nor   4'b1101 /* out = a NOR b */
`define decr  4'b1110 /* out = b MINUS 1 */
`define nand  4'b1111 /* out = a NAND b */
```

The operations are detailed in Table 10-2, Figure 10-1, Figure 10-2, Figure 10-3, and Figure 10-4.

Table 10-2 ALU Exercise: Explanation of Opcodes

Opcode	Result	Carry	Explanation
move	out=b	0	Data pass-through
comp	bitwise complement of b	1	Complement
and	a AND b	0	Bitwise AND
or	a OR b	0	Bitwise OR
xor	a XOR b	0	Bitwise exclusive OR
add	a PLUS b	carry	Addition
incr	b + 1	carry	Increment
sub	a - b	1 = no borrow 0 = borrow	Subtraction with borrow
rotl	see Figure 10-1	b[15]	Rotate left
lshl	see Figure 10-2	b[15]	Logical shift left
rotr	see Figure 10-3	b[0]	Rotate right
lshr	see Figure 10-4	0	Logical shift right
xnor	a XNOR b	1	Bitwise exclusive NOR
nor	a NOR b	1	Bitwise NOR
decr	b - 1	0 = no borrow	Decrement b
nand	a nand b	1	Bitwise NAND
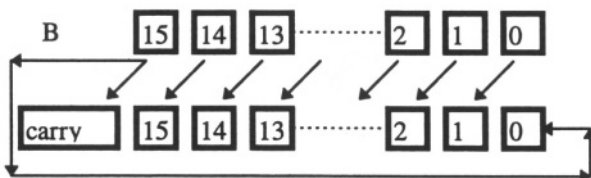
Figure 10-1 Rotate Left
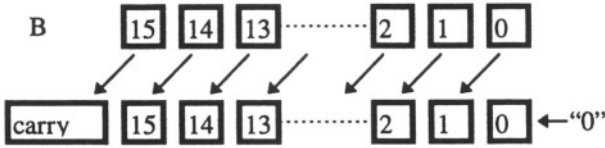
Figure 10-2 Logical Shift Left with 0 Fill
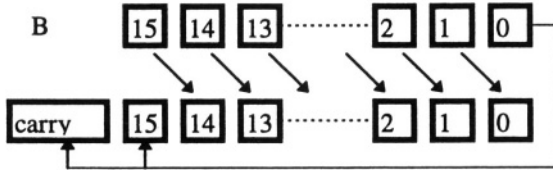
Figure 10-3 Rotate Right
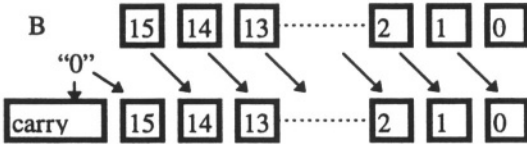
Figure 10-4 Logical Shift Right with 0 Fill

Example 10-16 illustrates a test bench for the ALU.

Example 10-16 Test Bench for the ALU

```
module test_alu;
reg [16+16+4+16+3:1] stim_res [1:100]; /* storage for stimulus
and response */
wire [15:0] aluout;  /* declare wires for results */
wire   zero,parity,carry;
integer pattern ;

reg [15:0] a,b ; /* declare registers for stimulus */
reg [3:0]  f;

reg [15:0] aluout_comp; /* declare registers to compare to */
reg zero_comp, parity_comp, carry_comp;

/* instantiate device under test */

alu   dut(a,b,f,aluout,zero,parity,carry);

initial begin
    pattern = 1 ;
    $readmemh("alu_test.vec", stim_res);
    forever  run_test;
    end

task run_test;
begin
  {a,b,f,aluout_comp,zero_comp,parity_comp,carry_comp}   =
stim_res[pattern];
  if (carry_comp === 1'bx) $finish;
  #100

  if (aluout !== aluout_comp) begin
    $display(
"compare error on aluout, pattern %d was %b should be %b.",
        pattern, aluout,aluout_comp);
    $finish;
  end

  if (zero !== zero_comp) begin
    $display(
"compare error on zero, pattern %d was %b should be %b.",
        pattern, zero, zero_comp);
    $finish;
  end

  if (parity !== parity_comp) begin
    $display(
"compare error on parity, pattern %d was %b should be %b.",
        pattern, parity,parity_comp);
    $finish;
  end

  if (carry !== carry_comp) begin
```

```
    $display(
"compare error on carry, pattern %d was %b should be %b.",
        pattern, carry,carry_comp);
    $finish;
  end

  pattern = pattern + 1;
end
endtask
endmodule
```

The test bench reads in an external set of vectors to apply and expected values to compare against the ALU outputs. These vectors are in a form readable by Verilog and are read from the file *alu_test.vec,* as is specified in the line with the *$readmemh* command. It is difficult to read these vectors because their bytes are not neatly aligned. Rather than go to the trouble of deciphering them, it is much easier to add *$display* statements into the test bench or ALU to evaluate what is happening with the vectors.

Please refer back to Chapter 8 on operators to create the functions for the alu. There are a few tricks in the expressions you will need to figure out to pass the test patterns. Each of the expressions is simple and can be expressed in a single short equation.

Enter the vectors shown in Figure 10-5 into a file called *alu_test.vec,* or better still, copy them off the disk enclosed with this book.

```
00000000000004
5807c3b2a9bba3
1d801d8001d802
091a7fffb891a8
0000891a0f6e5b
1bba091a1091a2
76e5891a1ffff8
091a091a200004
0000891a3091a8
0000c3b2c07659
47884788047880
0000c3b2c87653
48094809048090
0000c3b2d61d91
4c894c8904c890
0000c3b2da1d92
5308d308853088
0765e1d9619431
076521d96d802b
00009943719428
091a00007ffff9
19436eee86eee8
```

Figure 10-5 ALU Test Vector File *alu_test.vec*

When running this exercise, do not instruct Verilog to compile the file *alu_test.vec*.
This file cannot be compiled by Verilog. It is read in from the test bench with the
$readmemh command. This test bench uses a *task,* which is explained in the next
chapter. The *$readmemh* command will be explained in Chapter 18 (regarding test
benches).

11 TASKS AND FUNCTIONS

Chapter 3 discusses structural modeling. Structural modeling is the way to create a 'hardware' hierarchy. Tasks and functions create a 'software' hierarchy similar to subroutines or procedures in a programming language. Tasks and functions may be synthesizable depending on their contents and usage.

TASKS

In Verilog, you may use a *task* to encapsulate a behavior. A *task* is defined in a module and is invoked when its name is called in procedural code. A *task* has access to all the data objects (nets registers, integers, reals, etc.), so it does not need to have inputs and outputs, though it can have inputs, outputs and inouts. Tasks may take more than zero time to complete; they can have delays, *wait* statements, and event controls in them. Tasks can be used to apply stimulus, simulate bus cycles, display contents of memories, and many other things. Tasks contain a single statement. If you want a task to have more than a single statement, use a *begin-end* block or a *fork-join* block.

Example 11-1 is the *hello* module modified to use a *task*.

Example 11-1 Hello Verilog Tasks

```
module hellot;
initial begin
    say_hello;
    say_hello;
end

task say_hello;
    $display("Hello Verilog Tasks!");
endtask

endmodule
```

The *task* in Example 11-1, *say_hello,* is executed when its name (*say_hello*) is encountered in the *initial* block. This *task* has no inputs or outputs and uses no data. The *test_alu* module in the previous exercise had a *task* called *run_test.* The *run_test task* did not have any inputs or outputs, but directly accessed and modified the values on the nets and registers in the module. When the *run_test task* executed its procedural assignment into the stimulus registers, the inputs were applied.

When a *task* has *input, output,* or *inout* ports, the port list is not declared the same way as a module. Instead, the port order is determined by the port declarations.

Example 11-2 is a simple example of a *task* that has *input* and *output* ports and references a register that is not in the *task*.

Example 11-2 *task* with Inputs, Outputs, and External References

```
module task1;
integer a, b, c, d;
initial begin
    a=3;
    b=4;
    d=12;
    add(a,c,b);
    $display(" final value for c = %0d",c);
end

task add;
input  [31:0]  in1;
output [31:0]  out;
input  [31:0]  in2;
    out = in1 + in2 + d;
endtask

endmodule
```

Example 11-2 shows a *task* with inputs and outputs. Notice that the order of the inputs and outputs is determined by the order in which they are declared because there is no port list. Also note that the ports are declared to be 32 bits wide (which is the size of an integer), so the ports match the integers declared in the module. If the size of the ports is not declared, the port size defaults to 1 bit wide. If you run this module, the final value for *c* (as you might expect) is 19.

Example 11-3 shows both the use of inouts in *tasks* and the effect of the port range declaration. The *task* increment has only a 3-bit *inout* port and therefore only outputs a number between 0 and 7.

Example 11-3 Effect of *task* Port Size

```
module task2;
integer a;
initial begin
    a=0;
    increment(a) ;
    $display("a=%0d",a);
    increment(a);
    $display("a=%0d",a);
    a=7;
    increment(a);
    $display("a=%0d",a);
end

task increment;
inout  [2:0] x;
    x = x + 1;
endtask

endmodule
```

Tasks create a local name space. You may declare *regs, integers, reals,* and other data objects inside a *task* and not interfere with similarly named items in a module. Anything declared in a *task* is accessible from the module that contains the *task* by creating a hierarchical name to the data. Example 11-4 shows a *task* with a local integer. The local integer is both written and read from outside the *task*.

Example 11-4 Accessing a *task* Local Variable from Outside the *task*

```
module task3;

initial begin
    $display("total=%0d",count.total);
    count.total =0;
    count;
    count;
    $display("total=%0d",count.total);
  end

task count;
integer total;
  total = total + 1;
endtask

endmodule
```

You have already seen how to access a value from the parent module in a *task*. You have also seen how to access a *task* local value in the parent module. What happens when the *task* and the parent module both have an item with the same name? Both

items have unique hierarchical names that allow independent access. Example 11-5 shows a module and a *task* that both contain integers named *a*. The example demonstrates both the upward and downward references to the two unique integers. The only trick to the upward reference is that you must know the instance name for the module. For this example, the module *task4* is not instantiated, so the instance name is *task4*. To review hierarchical names, see Chapter 3.

Example 11-5 *task* **Local and Module Items with the Same Name**

```
module task4;
integer a;
initial begin
    a=12;  // set top level a
    t4;   // run the task
    $display("top a is %0d",a);
    $display("lower a is %0d",t4.a);
  end

task t4;
integer a;
  a = task4.a * 2;
endtask
endraodule
```

Automatic Tasks

Local data can be declared in a *task*. The only problem with local data in a *task* is with multiple invocations of a *task*. If a *task* is triggered while it is still running (for example, if a *task* is triggered from more than one place or a re-entrant *task*), this is legal, but there is only one copy of the local data, so the multiple invocations will be sharing data.

IEEE1364-2001 adds a new keyword to task declarations *automatic*. Automatic tasks do not have static data, the data is allocated when the task is called. Therefore Example 11-4 and Example 11-5 would not work with automatic tasks.

Example 11-6 Shows a re-entrant task. If *task reentT* is instead declared *as task automatic reentT,* each invocation of the task will get a unique copy of the data.

Example 11-6 Re-Entrant Task

```
module reent;
reg [2:0] a,b;
initial begin
  a = 3'b001;
  b = 3'b101;
  reentT(a);
  $display($time, " a = %d",a);
end

initial begin
  #5 reentT(b);
  $display($time, " b = %d",b);
end

task reentT;
inout [2:0] x;
  #20 x = x + 1;
endtask

endmodule
```

Example 11-6 shows a task that is invoked from more than one place at the same time. In this example the task will be called with *b* while it is operating on *a.* The results are quite deterministic; At time 0 the task is called with a(1) and it delays until time 20, at time 0, *x* takes the value 1. At time 5 the task is called with *b* (5). At time 5, *x* becomes 5. At time 20, *x* is incremented (6) and returned to a. At time 25, *x* is incremented again (7) and returned to *b.*

Common Uses for Tasks

Example 11-7 illustrates a *task* that might be used in a processor model to model a read cycle. This *task* has ports and references data directly in the module. This *task* also takes several clock cycles to complete, unlike the other *tasks* that take zero time.

Example 11-7 Read Cycle *task*

```
module readcycle(clock,data_ready,datain,address,read);
input clock,data_ready;
input [31:0] datain;
output [31:0] address;
reg [31:0] address;
output read;
reg read;

reg [31:0] data;

initial begin
    do_read('h00001234,data);
    do_read('habcdbabe, data);
end

task do_read;
input [31:0] location;
output [31:0] bus_value;
begin
    @(posedge clock) address = location;
    @(posedge clock) read = 1;
    while( ! data_ready)
        @(posedge clock) ;
    bus_value = datain;
    read = 0;
end
endtask

endmodule
```

The *readcycle* module is a correct, complete module. But it is not necessarily complete or correct to model a processor in this way. Instead, the *do_read task* would be within a larger module with several other *tasks* for other processor cycles. The *do_read task* directly references the incoming signals *clock, data_ready,* and *data_in,* and directly changes the value of the *address* register.

Another common use for *tasks* is to have *tasks* in a module that are only used interactively for debugging. These *tasks* can be activated interactively during a debugging session to display critical information or to set up particular values. A *task* to display the contents of a memory is shown in the parameterized RAM example in Chapter 14.

Finally tasks may be imbedded into your model or test bench to aid debugging. Most simulators provide an interface to allow calling tasks from the command line. An example of an embedded debug *task* can be found in the ram model in Chapter 14. It is often a desirable to see the contents of a ram when debugging. The ram model contains a *task* called dump that may be called during simulation.

FUNCTIONS

Functions differ from tasks in three important ways:

1. Functions must return a value.

2. Functions must take zero time.

3. Functions cannot contain delay or event controls.

A function returns a value by assigning a value back to a pseudo-variable represented by the function name.

When you declare a function, you must also declare the type and size of the return value. If the size of a function is not declared, it defaults to 1 bit. The size of a function is declared like any other range declaration, so the number of bits returned can be from one to one million.

Example 11-8 illustrates a function that returns an integer and counts the number of bits that are set in a 32-bit input value.

Example 11-8 Count Bits Function

```
module cbits;

initial begin
  $display("the answer is %d",count_bits(87));
  $display("the answer is %d",count_bits('h12345678));
  $display("the answer is %d",count_bits('hffff_ffff));
end

function integer count_bits;
input [31:0] a;
begin
  count_bits = 0;
  while(a) begin
    if( a[0] ) count_bits = count_bits + 1;
    a = a >> 1 ;
  end
end
endfunction

endmodule
```

The *count_bits* function not only assigns a value to *count_bits* to set the return value, but the function uses the pseudo-variable *count_bits* as a temporary variable during its calculation.

A function can also be called in a continuous assignment. Whenever any one of the inputs to the function changes, the function is called. This is consistent with the behavior of the continuous assignment because a continuous assignment is reevaluated whenever anything on the RHS changes. Example 11-9 provides an example of a mux implemented with a function and a continuous assignment.

Example 11-9 Mux with Function and Continuous Assignment

```
module  muxfunc(y,a,b,c,d,sel);
output  [7:0]  y;
input  [7:0]  a,b,c,d;
input  [1:0]  sel;

assign y = muxfunct(sel,a,b,c,d) ;

function  [7:0]  muxfunct;
input  [1:0]  sel ;
input  [7:0]  a,b,c,d;
  case(sel)
    2'b00 : muxfunct = a;
    2'b01 : muxfunct = b;
    2'b10 : muxfunct = c;
    2'b11 : muxfunct = d;
    default : muxfunct = 8'bx;
  endcase
endfunction
endmodule
```

A function generally returns only one value. But there is a trick to make a function return more than one value: Use the concatenation. Example 11-10 presents an 8-bit integer divide function that returns both the quotient and remainder of an 8-bit division.

Example 11-10 Divide Function Returning Two 8-Bit Values

```
module divfunc;
reg [7:0] a, b, q, r;
initial begin
  a = 5; b = 3;
  doit;
  a = 187; b = 3;
  doit;
  a = 255; b = 18;
  doit;
end

task doit;
begin
  {q,r} = div(a,b);
  $display(
    "%d goes into %d %d times with a remainder of %d",
    b, a, q, r) ;
end
endtask

function [15:0] div;
input  [7:0] dividend,  divisor;
reg [7:0] quotient, remainder;
  begin
    quotient = dividend / divisor;
    remainder = dividend % divisor;
    div = {quotient,remainder};
  end
endfunction

endmodule
```

This module also has a task that calls the function and displays the results to keep the code from getting cluttered by excessive *$display* statements. The results are shown in Example 11-10 Results.

Example 11-10 Results (for *divfunc.v*)

```
 3 goes into  5  1 times with a remainder of  2
 3 goes into 187 62 times with a remainder of  1
18 goes into 255 14 times with a remainder of  3
```

Functions and Integers

The declaration of a function or the inputs to a functions can be declared as integer. Example 11-11 shows a function with integers and the results.

Example 11-11 Function with Integers

```
module add;
initial begin
  $display("1 + 2 is %0d", add(1,2));
  $display("47 + 32'hffff_ffff is %0d",
add(47,32'hffff_ffff));
end

function integer add;
input a, b;
integer a, b;
  add = a + b;
endfunction

endmodule
```

The results are shown below in Example 11-11 results. The 2's complement of -1 is 32'hffff.

Example 11-11 Results

```
1 + 2 is 3
47  + 32'hffff_ffff  is  46
```

Automatic Functions

IEEE 1364-2001 adds a new keyword to function declarations *automatic*. Automatic functions do not have static data, the data is allocated when the function is called. Therefore with an automatic function it is now possible to create recursive functions. Example 11-12 shows a simple exmaple.

Example 11-12 Automatic Recursive Function

```
function automatic integer factorial;
input I;
  if( I==1)
    factorial = 1;
  else
    factorial = I * factorial(I-1);
endfuction
```

Exercise 5 Functions and Continuous Assignments

Now that you know about functions and continuous assignments, modify your result from the previous exercise so a function calculates *carry* and *aluout.* Assign the values to *aluout* and *carry* from the function in a continuous assignment. Use two more continuous assignments; One to calculate *zero* and one to calculate *parity.* Test your new ALU using the same test bench from the previous exercise.

12 ADVANCED PROCEDURAL MODELING

USING THE EVENT DATA TYPE

In Chapter 1, the *phone* example used the *wait for event* (@) operator and the *signal event* (->) operator. The *signal event* (->) is an operator that only works with the *event* data type and the *wait for event* (@) operator. Events have no value or duration. All they do is indicate something happened. Example 12-1 shows how the *signal event* operator (->) works.

Example 12-1 Using the *event* Data Type

```
module  show_event;
event the_event, something_else;

always @ the_event begin
    $display("The event happened at time %0d", $time);
    -> something_else;
end
```

```
always @ something_else $display("something else happened");

initial begin
    # 2 -> the_event;
    # 5 -> something_else;
    # 1 -> the_event;
end
endmodule
```

At time 0, both *always* blocks in Example 12-1 start and wait for their events. At time 2, the *initial* block signals *the_event*; the first *always* block continues, prints its message, and signals the other *always* block; and the second *always* block also prints its message. At time 7, the *initial* statement signals something else and the second block prints its message again. Finally, at time 8, the *initial* block signals the first *always* block, which signals the second *always* block, and both messages print.

The example shown in Example 12-1 has no practical application. What is a good use for the *signal event* operator (->) ? Suppose you are modeling a processor that has several things that it does, such as *fetch, execute,* and *store.* You could model each part of the behavior with a separate *always* block. You could also use the signal *data type* to model conditions such as *reset,* errors, or completion. Example 12-2 is a partial example of a processor model.

Example 12-2 Using Events To Simplify Modeling

```
module event_processor (clock,  ....);
input clock,  ...;
event fetch, execute, store, reset, error, halt;

initial -> reset;

always @ (reset or fetch) begin
    @(posedge clock)  ...
    ... // code that does a fetch cycle
    @(posedge clock)...
    -> execute
end

always @ execute begin
    @(posedge clock)  ...
    ... // code that does an execute cycle
    @(posedge clock)...
    if ( some condition ) -> store;
    if (something bad ) -> error;
    if (opcode == halt_code) -> halt;
end
```

```
always @ error begin
    $display("processor error occurred");
    $stop;
end
```

The event data type is generally not used for synthesizable code. One of the best places to use the event data type is in testbenches. Test benches for complex systems often need to go through reset or pre-load sequences before a test can be applied. The reset or pre-load code could be common to many tests and an event may be used to signal when the actual test sequence may begin.

PROCEDURAL CONTINUOUS ASSIGNMENTS

So far you have learned how to use the procedural assignment (which is like a typical assignment in a programming language) and the continuous assignment (which acts like combinatorial logic). The procedural continuous assignment has unique characteristics, and it shares characteristics with each of the other two assignments, but is identical to neither. The procedural continuous assignment overrides the normal behavior of a reg.

Example 12-3 illustrates a simple flip-flop module.

Example 12-3 A Simple Flip-Flop

```
module ff1(q,clk,d);
output q;
reg q;
input clk,d;
always @(posedge clk)
    q <= #(`REG_DELAY) d;
endmodule
```

If we want to add an asynchronous reset to the flip-flop, we could modify the first *always* block to be sensitive to changes on both *clk* and *reset,* or we could add a second *always* block, as seen in Example 12-4.

Example 12-4 A Flip-Flop with a Bad Reset

```
module ff2bad(q,clk,d,reset);
output q;
reg q;
input clk, d, reset;
always @(posedge clk)
    q <= #(`REG_DELAY) d;
always @reset
    if( reset ) q <=#(`REG_DELAY) 0;
endmodule
```

The module in Example 12-4 looks like it might work. But what happens if *reset* is high for many clock cycles? The *q=0* statement executes when *reset* first occurs, but if the clock signal arrives, the *q=d* statement can also execute. Thus, the flip-flop does not stay reset because both the procedural assignments have the same precedence.

The procedural continuous assignment has higher precedence than the procedural assignment, so this module can be fixed as shown in Example 12-5.

Example 12-5 A Flip-Flop with Reset

```
module ff2(q,clk,d,reset);
output q;
reg q;
input clk, d, reset;
always @(posedge clk)
    q <= #(`REG_DELAY) d;
always @reset
    if( reset )
        assign q=0;
    else
        deassign q;
endmodule
```

As you can see, the procedural continuous assignment (PCA) consists of two statements: The *assign* and *deassign.* Whenever *reset* changes, the *if* evaluates the state of *reset.* If *reset* is 1, then the *assign* statement is executed and the value of *q* is overridden to 0. Even if the statement *q-d* is executed, the value of *q* remains 0. When *reset* changes and the value is not 1, the *deassign* statement is executed and *q* reverts to its normal behavior, and the value of *q* changes the next time the *q=d* statement is executed.

To further examine the behavior of the PCA, *set* can be added to the module, as shown in Example 12-6.

Example 12-6 A Flip-Flop with Incorrect Set and Reset

```
module ff3bad(q,clk,d,reset,set) ;
output q;
reg q;
input clk, d, reset, set;
always @(posedge clk)
    q <= #(`REG_DELAY) d;
always @reset
    if( reset )
        assign q=0;
    else
        deassign q;
always @set
    if( set )
        assign q=1;
    else
        deassign q;
endmodule
```

With this module, it was easy to add the *set* condition by duplicating and modifying the *reset* code. But there is a problem with this module: What happens when both *set* and *reset* are 1? That is an illegal condition. When *set* and *reset* are both 1, and one of them goes to 0, a *deassign* statement executes. Once a *deassign* statement is executed, the behavior of *q* reverts to normal (since the PCA is no longer overriding the procedural assignment), and the *q=d* statement will have effect.

Note this sequence of events: *d* is 0; *clock* is 0; *reset* is 1; therefore *q* is 0. Next, *set* goes to 1, so *q* goes to 1. Finally, *reset* goes to 0. The *deassign* statement is executed, but *q* stays 1, its former value. (Remember, *set* is still 1.) Now the clock rises and the *q=d* statement executes. Because the *deassign* statement was executed, there is no override in effect for *q* and *q* goes to 0 even though *set* is 1. Therefore Example 12-6 is a bad model.

Example 12-7 is the final flip-flop model with *set* and *reset* working properly.

Example 12-7 A Flip-Flop with Correct *set* and *reset*

```
module ff3(q,clk,d,reset,set);
output q;
reg q;
input clk, d, reset, set;
always @(posedge clk)
    q <= #(`REG_DELAY) d;
always @(reset or set)
    case( {reset,set} )
        2'b00: deassign q;
        2'b10: assign q=0;
```

```
        2'b01: assign q=1;
        default: assign q=1'bx;
    endcase
endmodule
```

You have just seen only one part of the behavior of the procedural continuous assignment (PCA), the overriding of the normal behavior of a reg.

Consider, for a moment, the following mux-like module.

Example 12-8 Incorrect Mux

```
module mux1bad(y,a,b,c,d,sel);
input a,b,c,d;
input [1:0] sel;
output y;
reg y;
always  @(sel)
    case(sel)
        2'b00 : y = a;
        2'b01 : y = b;
        2'b10 : y = c;
        2'b11 : y = d;
    endcase
endmodule
```

So what is wrong with the mux in Example 12-8? The output *y* will only change when *sel* changes. If *sel* is 0 and *a* changes, the output *y* will not change. Thus, this is not a good mux. One way to fix this mux would be to make the *always* block sensitive to all the inputs changing. You could do this by changing *@(sel)* to *@(sel or a or b or c or d)*. This section of the book is about the PCA, so Example 12-9 shows how to fix this mux using the PCA.

Example 12-9 Mux with PCA

```
module mux2(y,a,b,c,d,sel);
input a,b,c,d;
input [1:0] sel;
output y;
reg y;
always @(sel)
    case(sel)
        2'b00 : assign y = a;
        2'b01 : assign y = b;
        2'b10 : assign y = c;
        2'b11 : assign y = d;
    endcase
```

```
endmodule
```

Using the PCA, whenever *sel* changes, a different assignment is executed. Just as with the continuous assignment, the left-hand side (LHS) of an active PCA gets a new value whenever the right-hand side (RHS) changes. Now this module behaves like a mux.

To summarize the PCA: It overrides the normal behavior of a reg. Only one PCA can be active on a reg at a time. The last PCA executed is the PCA in effect. The *deassign* statement stops the effect of the PCA and the next procedural assignment to the reg takes effect normally.

With the introduction of the PCA you have seen all three types of assignments summarized in Table 12-1:

Table 12-1 Summary of Assignment Types

Assignment Type	LHS	When Evaluated	Where in Module
Procedural assignment	reg	When encountered	In procedural block
Continuous assignment	net	Whenever RHS changes	On its own
Procedural continuous assignment	reg	When encountered, then whenever RHS changes, Until deassign	In procedural block

LHS = Left Hand Side (destination of assignment)

RHS = Right Hand Side (operands in assignment)

The procedural continuous assignment is generally not used in synthesizable code. The previous examples were used to illustrate the behavior of the procedural continuous assignment. Example 12-10 has nothing to do with the procedural continuous assignment but illustrates the proper coding technique for a synthesizable flip-flop.

Example 12-10 Proper Synthesizable Flip-Plop

```
module ff_syn(q,clk,d,reset, set) ;
output q;
reg q;
input clk, d, reset, set;
always @(posedge clk or posedge reset or posedge set)
    if(reset)
        q <= #(`REG_DELAY) 0;
     else
        if(set)
             q  <=  #  (`REG_DELAY)  1;
        else
             q  <=  #  (`REG_DELAY)  d;
endmodule
```

A REMINDER ABOUT PORTS AND REGS

In Chapter 4, you learned that only output ports can be regs, and that *inout* ports must be nets. Now that you are about to try to use *inout* ports, this is worth repeating because you will generate an error if you try to declare an input or *inout* port as a reg. Figure 12-1 shows the possible relationships of ports and regs. In high level modeling you will often want to declare an *inout* port as a reg, but this will not work. Actually, you only want to declare the output side of an *inout* port as a reg.
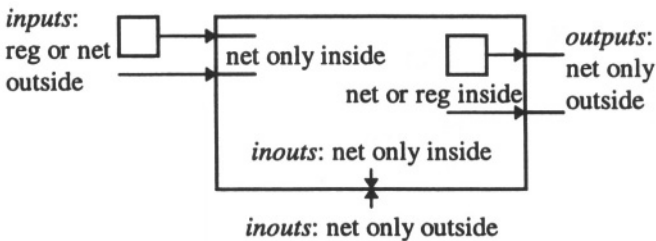
Figure 12-1 Relationships of ports and regs

MODELING WITH *INOUT* PORTS

Because *inout* ports cannot be regs, how is it possible to do high level modeling with *inouts*? The key is to have a reg for the output and a continuous assignment to the port. When sampling the input side of the *inout,* put a *z* in the reg, and read the port. When driving a value out, place the value in the output reg.

Example 12-11 *inout* Port Connected to a Reg

```
module io1(data, read, write);
inout data;
input read, write;
reg port_reg, internal_reg;

always @read
    if( read )
        port_reg = internal_reg;
    else
        port_reg = 1'bz ;

always @(posedge write)
    internal_reg = data;

assign data = port_reg ;

endmodule
```

Example 12-11 shows an output reg directly connected to an *inout* port through a continuous assignment.

Putting a *z* in the output reg is like turning off the driver on the port. This model can be simplified by making the continuous assignment control whether or not the reg is connected. Example 12-12 shows how the continuous assignment was changed so the reg can be disconnected from the *inout* port.

Example 12-12 Reg with Controllable Connection to *inout* Port.

```
module io2(data, read, write);
inout [15:0] data;
input read, write;
reg [15:0] internal_reg;

always @(posedge write)
    internal_reg = data;

assign data = read ? internal_reg: 16'bz ;

endmodule
```

The *io2* module illustrates how the *io1* module could easily be expanded to be more than 1 bit, and how the module could be simplified by using a continuous assignment to turn off a reg from driving an *inout* port.

NAMED BLOCKS

In Chapter 4, the *begin-end* and *fork-join* blocks were introduced. Both of these blocks can also be named. Naming a block allows the block to have local variables. Named blocks can be referenced with the *disable* statement to stop the execution in the block. (The disable statement is introduced in the next section.) To name a block, place a colon and the name for the block directly after the *begin* or *fork* statement. Example 12-13 shows both named *begin-end* and *fork-join* blocks with local regs.

Example 12-13 Named Blocks

```
module nameblock;
reg a;
initial begin : b1
    reg a;
    a = 1;
    end
initial fork :b2
    reg a;
    a = 0;
    join
endmodule
```

In this module, the three regs are unique regs. The regs each have a unique hierarchical name. The top level *a* is just named *a*; the other two regs are *b1.a* and *b2.a.* The block names can be any legal Verilog identifier. The block names *b1* and *b2* are used for simplicity.

THE *DISABLE* STATEMENT

The *disable* statement is a way of stopping execution in a named block or task. The *disable* statement is useful for handling time-outs and reset conditions. The disable can even be used within a named block or task to stop itself. To use the *disable* statement, insert the *disable* keyword and the name of the block or task. Example 12-14 illustrates some examples.

Example 12-14 The *disable* Statement

```
module disable1;
initial begin
  do_it;
  $display("Finished do it at time %0d",$time);
end

initial begin
  #55 disable do_it;
end

task do_it;
  forever
    #10 $display("doing it at time %0d",$time);
endtask

endmodule
```

The code in Example 12-14 finishes at time 55. Normally the task *do_it* would never finish due to the *forever* loop in it. The *disable* of *do_it* causes the task to immediately exit, so at time 55 the task exits and the message "Finished do it at..." prints. The output from Example 12-14 is shown below in Example 12-14 Results.

Example 12-14 Results (for *disable1.v*)

```
Compiling source file "disable1.v"
Highest level modules:
disable1

doing it at time 10
doing it at time 20
doing it at time 30
doing it at time 40
doing it at time 50
Finished do it at time 55
```

If you disable a *begin-end* block inside an *always* statement, what happens? The *begin-end* block immediately finishes. Because an *always* statement is a loop that starts again from the top when its statement finishes, the *begin-end* block starts again from the top. This behavior makes the *disable* an easy way to implement a reset for large segments of high level code. Example 12-15 shows a partial model of a complex CPU in which *disable* is used for *reset*.

Example 12-15 *disable* **Used To Model Reset**

```
module cpu;
reg reset, clock;

initial begin
    clock = 0;
    reset = 1;
    #10 reset = 0;
    #10 reset = 1;
    #10 reset = 0;
    #20 $finish;
end

always #1 clock = ~ clock;

always begin : fetch
    @(posedge clock)
    if (reset) disable fetch;
    $display("fetch process ready");
    // there would be code here for
    // the ongoing fetch process
    #10000 $display("fetch process done");
    // in a real processor the fetch process
    // might be an infinite loop
end


always begin : execute
    @(posedge clock)
    if (reset) disable execute;
    $display("execute process ready");
    // there would be code here for
    // the ongoing execute process
    #10000 $display("execute process done");
    // in a real processor the execute process
    // might be an infinite loop
end

always @( posedge reset)
    begin
        disable fetch;
        disable execute;
end

endmodule
```

The last *always* block that checks for *reset* handles the asynchronous nature of *reset.* Thus, the other blocks do not need to check for *reset* through their code. The *fetch* and *execute* blocks check for *reset,* and then disable themselves so that they will not continue if *reset* is still asserted. It is important that there is a delay or event control between the start of an *always* loop and the statement that self-disables the block. If there is no delay or event control, the *always* loop may become a zero-

delay infinite loop under the condition that it self-disables. Zero-delay *always* loops stop the progression of simulation time and your simulation freezes or hangs.

WHEN IS A SIMULATION DONE?

There are three ways a simulation can terminate:

1. When the simulation runs out of events to process.

2. When the simulation encounters a *$finish* command.

3. Through user intervention.

A simulation such as the *hello* simulation finishes when it has evaluated all available high level statements, or gates to evaluate. Most simulations include clocks that never terminate, so it is unlikely that many simulations will be terminated by running out of events to process.

The simplest way to ensure a simulation will terminate in a timely fashion is to include a *$finish* in the test bench. This is often done with a statement similar to the one shown in Example 12-16.

Example 12-16 Controlling When a Simulation Finishes

```
initial #1000 $finish;
```

With this statement, we know the simulation will terminate at time 1000. The only problem with this method is that if more stimulus is added, the simulation might terminate before the end of the stimulus. So if you use a separate *initial* block like this to end your simulation, be sure to calculate how far you want it to run before it terminates.

Often the *$finish* statement is included at the end of the stimulus (or elsewhere in the test bench) to check for the final test having been applied. If the correct conditions to terminate the simulation never occur (for example, if your code includes a zero-delay *always* loop), the simulation may never end on its own.

If the simulation does not terminate on its own by running out of events to process or by encountering a *$finish* statement, the only other way to stop Verilog is through user intervention. To manually stop Verilog, press *Control-C* (in most systems), or click the *stop* or *interrupt* button if you are using a graphical user interface. Once you have interrupted Verilog, you can issue the *$finish;* command manually on a

command line, or issue the *exit* command from a menu of a graphical interface
system.

13 USER-DEFINED PRIMITIVES

Although the aim of this book is to teach high level modeling in Verilog, the book would not be complete without mentioning user-defined primitives (UDPs). A UDP describes a piece of logic with a truth table. UDPs can be either combinatorial or sequential. As you may recall, the Verilog primitive set does not include any muxes, AND-OR-INVERT gates, or flip-flops. You can model all of these simple functions with UDPs.

Why use UDPs? For some types of logic, a truth table may be the easiest way to model them. The main reason UDPs are used is performance: Verilog evaluates UDPs quickly, and UDPs take up only a small amount of memory. For example, a mux is typically modeled with an inverter, two AND gates, and an OR gate. These four gates can be replaced with a single UDP. Flip-flops, which take even more gates to model than a mux, can be replaced by a UDP. The most common use for UDPs is in modeling a library of ASIC cells or standard components.

COMBINATORIAL UDPS

The simplest thing you might want to model with a UDP is a mux. If you want to model a mux, is the mux optimistic or pessimistic? If the two inputs are 1, and the select is unknown, is the output 1 or *x*? Because it is your UDP, you get to choose which result you want.

Optimistic Mux

This mux is optimistic because if the inputs are the same and the select is unknown, the input still propagates.

Example 13-1 Optimistic Mux UDP

```
primitive omux( y, sel, a, b );
output y;
input sel, a, b;
table
//s a b : y;
    0 0 ? : 0;
    0 1 ? : 1;
    1 ? 0 : 0;
    1 ? 1 : 1;
    x 0 0 : 0; // This is an optimistic line.
    X 1 1 : 1; // This one too!
endtable
endprimitive
```

Pessimistic Mux

This mux does not include lines for an unknown select. Therefore it is more pessimistic than the mux in Example 13-1 because the output will be unknown.

Example 13-2 Pessimistic Mux UDP

```
primitive pmux( y, sel, a, b );
output y;
input sel, a, b;
table
// s a b : y;
   0 0 ? : 0;
   0 1 ? : 1;
   1 ? 0 : 0;
   1 ? 1 : 1;
endtable
endprimitive
```

The Gritty Details

UDPs always have scalar inputs (1 bit wide). Combinatorial UDPs can have up to ten inputs (you would not want to create a table larger than that anyway). UDPs in original Verilog have only one output. The current IEEE 1364 standard provides for multiple-output UDPs, but this is not common yet. Just as with the built-in primitives, the output port must be the first port in the port list.

UDPs start to look a bit like modules except that *module-endmodule* is replaced by *primitive-endprimitive.* The other major difference between UDPs and modules is that instead of the usual module contents, there is only the *table-endtable* construct in the body of the UDP. Table 13-1 shows the symbols used so far, and what they mean.

Table 13-1 Basic UDP Table Symbols

Symbol	Description
0	Logic 0
1	Logic 1
x	Unknown
?	Match 0, 1, or x

If a set of inputs is not covered by a line in the table, the output will be unknown. One of the most common errors in creating UDPs is to omit one or more combinations of inputs, with the result that the output unexpectedly becomes unknown. Verilog produces an error message if two lines in a UDP describe the same condition with different outputs. Verilog issues a warning if two lines contain the same input and output description.

The order of the columns in the table entries is determined by the port list, not by the order that the ports are declared in the input statement(s). Although the comment just after the *table* keyword is not required, it is good modeling practice to label the columns of the table. As you can see from the examples, each line of the table contains the set of inputs, followed by a colon, then the output, and ends with a semicolon. White space is not required in the table so the code in Example 13-3 is legal but difficult to read.

Example 13-3 One-Line UDP

```
primitive umux( y, sel, a, b ); output y; input sel, a, b;
table  00?:0;01?:1;1?0:0;1?1:1;   endtable
endprimitive
```

SEQUENTIAL UDPS

Because Verilog does not include a set of built-in latches and flip-flops, latches and flip-flops are commonly modeled with UDPs, as shown below in Example 13-4 and Example 13-5.

Example 13-4 Level-Sensitive D Latch

```
primitive dlatch(q, ena, d);
output q;
reg q;
input ena, d;
table
// e d : q : q+
   1 0 : ? : 0;
   1 1 : ? : 1;
   0 ? : ? : -; //no change on low enable
endtable
endprimitive
```

Example 13-5 Edge-Sensitive D Flip-Flop

```
primitive  dff(q,clk,d);
output q;
reg q;
input clk, d;
table
// c d : q : q+
   r 0 : ? : 0;
   r 1 : ? : 1;
   f ? : ? : -; //no change on falling clock
   ? * : ? : -; //no change on steady clock
endtable
endprimitive
```

The sequential UDP differs slightly from the combinatorial UDP. First, the output is declared as a *reg* to alert Verilog that this is a sequential UDP, not a combinatorial one. Next, notice that the table has an extra column, separated by colons. This field represents the current state of the output. The final column is the next output. There are also a few new symbols used in the table, as listed in Table 13-2.

Table 13-2 Symbols for Sequential UDP Tables

Symbol	Description
r	Rising a change from 0 to 1
f	Falling a change from 1 to 0
*	Any change
-	The output remains unchanged

The *r* and *f* are quick ways to show a transition. The primitive in Example 13-6 describes exactly the same table as for the flip-flop in Example 13-5.

Example 13-6 Flip Flop Using Explicit Edge Definitions

```
primitive dff1(q,clk,d);
output q;
reg q;
input clk, d;
table
// clk d : q : q+
    (01) 0 : ? : 0;
    (01) 1 : ? : 1;
    (10) ? : ? : -; //no change on falling clock
    ? (??) : ? : -; //no change steady clock, data changing
endtable
endprimitive
```

Any transition on any of the input signals may be used. A UDP may be edge-sensitive to a clock, but level-sensitive to set and reset signals. In a sequential UDP, each row of the table may only contain one transition. There is no way to specify edges on two signals at the same time.

One of the most common problems with sequential UDPs is that the output can unexpectedly go to *x.* This is usually caused by omitting a condition from the UDP's table. In a sequential UDP, all transitions that do not affect the output must be specified, otherwise the output will go to *x.* That is why (in the D flip-flop examples) there is an entry for when the clock is steady and the data change. If that line were omitted, when the clock is steady and data changed, the output would go to *x.* With an edge-sensitive UDP, the output must be specified for all edges of all inputs, or the output will become unknown. In the *dff* and *dff1* primitives, the transitions of clock from 0 to *x, x* to 1,1 to *x,* and *x* to 0 are not specified, so any of these will set the output to *x.*

Sequential UDPs are limited to nine inputs (instead of the ten allowed by combinatorial UDPs) because the previous output is counted as one of the inputs for the table.

A sequential UDP may have its output initialized by adding an *initial* statement to the UDP. The *initial* statement in a UDP may only set the output of the UDP to a constant. The procedural assignment setting the register to a constant is the only statement allowed in an *initial* block in a UDP.

Example 13-7 *Initial* Block in a UDP

```
primitive dff2(q,clk,d);
output q;
reg q;
input clk, d;
initial q = 0;
table
// c d : q : q+
   P 0 : ? : 0;
   P 1 : ? : 1;
   n ? : ? : -; //no change on falling clock
   ? * : ? : -; //no change on steady clock
endtable
endprimitive
```

UDP INSTANCES

UDP instances look exactly like instances of built-in primitives. UDP instances optionally have instance names and delay specifiers. See Appendix A for more details on primitive instances.

Table 13-3 summarizes the differences between built-in primitive instances, user-defined primitive instances, and module instances.

Table 13-3 Summary of Instance Types

Instance Type	Instance Name	Port Order	Port Size	Delay Specification	Name Of Device
Built-in primitive	Optional	Output first	1 bit	Optional	Built in reserved names
UDP	Optional	Output first	1 bit	Optional	User-defined
Module	Required	User-defined	User-defined	None	User-defined

THE FINAL DETAILS

UDPs may not use *z* (high impedance) in their input or output specifications. UDPs with more inputs are more error prone and use more memory, so it is not recommended to use UDPs with more than six inputs.

Table 13-4 provides a list of all the UDP table symbols used in Verilog.

Table 13-4 Complete List of UDP Table Symbols

Symbol	Where	Meaning
0	Input or output	Logic 0
1	Input or output	Logic 1
x	Input or output	Unknown
?	Input only	Any of 0 1 or x
b	Input only	Either 1 or 0
(mn)	Input only	A change from m to n
r	Input only	Rising edge (01)
p	Input only	Possible rising edge (01) (0x) (x1)
f	Input only	Falling edge (10)
n	Input only	Possible negative edge (10) (1x) (x0)
*	Input only	Any edge (??)
–	Sequential output	No change

Exercise 6 Using UDPs

Start with the 8-bit adder from exercises 2 and 3, repeated in Figure 13-1. This 1 bit adder used several built-in primitives. All of those gates can be replaced with two UDPs. Design two combinatorial UDPs. The first UDP should have three inputs, and generate a sum. The second UDP should have the same three inputs and generate a carry. Now create an adder module just like the one from exercise 2, but instead of the gate instances, you will have instances of your two new primitives as shown in Figure 13-2. Simulate the new adder with the 2-, 4-, and 8-bit adders from exercise 2 and the test bench. You should get the same results.
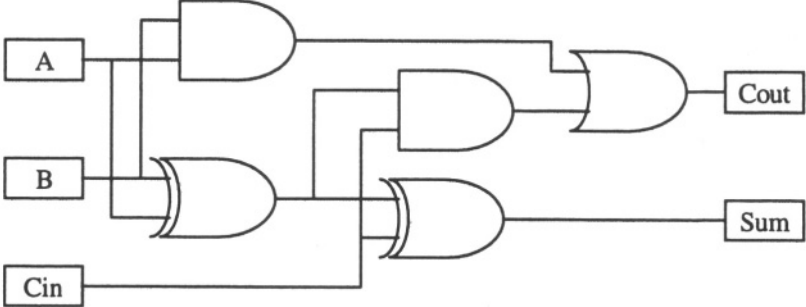
Figure 13-1 Adder Using Five Built-in Primitives
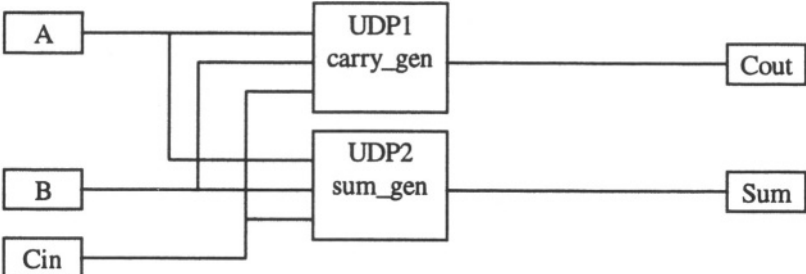
Figure 13-2 Adder Using Two UDPs

This Page Intentionally Left Blank

14 PARAMETERIZED MODULES

One big advantage of designing with Verilog is the ability to parameterize modules. You can design a generic adder and decide how many bits you need later. You can use the same parameterized adder as a 5-bit adder in one place and as a 64-bit adder elsewhere. Parameters are often used to describe the word size of a module, the number of words in a memory, or even delays. Delays are more commonly set up with a *specify* block that can be annotated with actual delays from an SDF file.

There are two Verilog keywords associated with parameters: *parameter* and *defparam*. When you define a parameterizable module, use the *parameter* keyword. When you instantiate a parameterized module, you can use the *defparam* statement to override the default value of parameters in the module.

parameters are used as constants. The only legal way to change the value of a constant is with the *defparam* statement, *parameters* and *defparam* statements are evaluated only during compilation. *parameters* cannot be changed during simulation time. The values are set at compile time. *parameters* are by default 32-bit values. It is not common, but the size of a parameter may be changed by specifying a range in the same way *wire* and *register* ranges are declared.

Various formats of the *parameter* statement are shown in Example 14-1.

Example 14-1 *parameter* Statements

```
parameter <parameter name> = <default value> ;
parameter size = 4;
parameter [3:0] width = 2;
parameter [1:11*8] hello_message = "hello word";
```

The best way to learn parameters is to see them in use.

N-BIT MUX

Example 14-2 illustrates a module of a n-bit wide 4-to-1 mux. It has a default word size of 32 bits.

Example 14-2 *n*-Bit Wide 4-to-1 Mux

```
module nmux4(a,b,c,d,sel,y);
// Parameterized n bit wide 4 to 1 mux.
parameter size = 32; // default to 32 bits
input [size-1 : 0] a,b,c,d;
input [1:0] sel;
output [size-1 :0] y;
reg [size-1 :0] y;

always @ (a or b or c or d or sel)
    case (sel)
        0 : y = a;
        1 : y = b;
        2 : y = c;
        3 : y = d;
        default : y = 'bx;  // will automatically size to fit
    endcase

endmodule
```

N-BIT ADDER

Example 14-3 shows a model of a simple parameterized adder. The adder has two inputs and an output that defaults to 32 bits.

Example 14-3 Parameterized Width Adder

```
module nadder(cout,sum, a,b,cin);
// Parameterized n bit wide  behavioral adder
parameter size = 32; // default to 32 bits
input [size-1 : 0] a,b;
input cin;
output [size-1 :0] sum;
output cout;

assign {cout,sum} = a + b + cin;

endmodule
```

N BY *M* MUX

Example 14-4 illustrates a more complicated model: A mux that has both parameterized width and a parameterized number of inputs. Because Verilog does not have an easy way to model a parameterized number of inputs, this model takes all the inputs as one long vector.

Example 14-4 Mux with Parameterized Width and Number of Inputs

```
module muxMto1 (Z,  SEL, D);

parameter N = 8;  // number of bits wide
parameter M = 4;  // number of inputs
parameter S = 2;  // number of select lines

parameter W = M * N;
`define   DTOTAL   W-1:0
`define   DWIDTH   N-1:0
`define   SELW     S-1:0
`define   WORDS    M-1:0

input   [`DTOTAL]  D;
input   [`SELW]    SEL;
output  [`DWIDTH]  Z;

integer i;
reg[`DWIDTH]   tmp, Z;  // tmp will be use to minimize events

always @(SEL or D) begin
  for(i=0; i < N; i = i + 1)  // for bits in the width
    tmp[i] = D[N*SEL + i];
  Z = tmp;
 end

endmodule
```

N BY *M* RAM

Example 14-5 shows a simple memory model with parameterized width, a number of words (address bits), and delays. This is an extremely simple memory model and does not completely model all the behaviors of a static RAM.

Example 14-5 Parameterized RAM

```
/* module for a simple parameterized ram */
module ram(data, address, read, write);
parameter width = 16 ; //default word size
parameter abits = 8 ; // default number of address bits

parameter twdh = 10;  // write data hold
parameter trd = 25; // rd to output delay
input [abits-1 : 0] address;
inout [width-1 : 0] data;
input read, write;

// declare the internal storage

reg [width-1 : 0] imem [0 : (1<<abits)-1] ;
/*
   Simple behavior of a static type ram
   write occurs when write is 1.
   to act more like a real static ram, if the data changes
   while write is asserted the data in the memory is changed
*/
always @(write or data)
    if(write)
        # twdh imem[address] = data ;

assign #trd data = read ? imem[address] : 'bz;

/*
   convenience task for displaying the contents of the memory
   during interactive debug
*/

task dump;
  input [31:0] low, high;
  integer i;
  begin
    for( i=low; i <= high; i=i+1)
        $display("imem[%h] = %h",i,imem[i]);
    $stop;
  end
endtask

endmodule
```

USING PARAMETERIZED MODULES

The *parameter* statements in parameterized modules set default values for the parameters. The *parameter* default values may be overridden on an instance-by-instance basis with the *defparam* statement. The format for the *defparam* statement is shown in Example 14-6.

Parameter Passing by Name

Example 14-6 The *defparam* Statement

```
defparam <instance name>.<parametername> = <value>
```

Example 14-7 uses the parameterized adder and creates three instances. One instance is 5 bits wide, one instance is the default 32 bits wide, and one instance is expanded to 128 bits wide.

Example 14-7 Using Parameterized Modules

```
module use_defparam;
wire [4:0] a5, b5, c5;  // some 5 bit wires
wire [31:0] a32, b32, c32; // some 32 bit wires
wire [127:0] biga, bigb, bigc; // some 128 bit wires
wire x,y,z;

// create some instances of the n bit adder

nadder a1 (z,a5,b5,c5,x);
nadder a2 (z,a32,b32,c32,x);
nadder a3 (z,biga,bigb,bigc,x);

defparam a1.size = 5;
defparam a3.size = 128;

endmodule
```

Parameter Passing by Order

Parameters can be passed by order directly in the statement creating the instance. Example 14-8 shows the syntax for parameter passing by order. The parameter passing by order syntax is shorter and more compact for simple modules with few parameters. The *ram* module in Example 14-5 has 4 parameters. Using the parameter passing by order you can pass some or all of the parameters but there is

no way to pass just the later parameters. For example if we had only wanted to pass *trd* to the *ram* module we would still need to pass the first three parameters.

Example 14-8 Parameter Passing by Order

```
module use_defparam_order;
wire [4:0] a5, b5, c5;  // some 5 bit wires
wire [31:0] a32, b32, c32; // some 32 bit wires
wire [127:0] biga, bigb, bigc; // some 128 bit wires
wire x,y,z, read, write;
wire [31:0] data;
wire [3:0] address;

// create some instances of the n bit adder

nadder #(5) a1 (z,a5,b5,c5,x);
nadder a2 (z,a32,b32,c32,x);
nadder #(128) a3 (z,biga,bigb,bigc,x);

// The ram example has 4 parameters

ram #(32,4,2,7) r1(address, data, read,write);

endmodule
```

Parameter Passing by Named List

IEEE 1364-2001 adds a new syntax for passing parameters by name, the syntax is quite similar to the module port connect by name syntax. The parameter passing by named list is compact and passes the parameters in the module instance statement; it eliminates the need for passing all the parameters if you only want to pass one of the latter parameters.

Example 14-9 Parameter Passing by Named List

```
module use_defparam_named_list;
wire [4:0] a5, b5, c5;  // some 5 bit wires
wire [31:0] a32, b32, c32; // some 32 bit wires
wire [127:0] biga, bigb, bigc; // some 128 bit wires
wire x,y,z, read, write;
wire [15:0] data;
wire [7:0] address;

// create some instances of the n bit adder

nadder #(.size{5)) a1 (z,a5,b5,c5,x);
nadder a2 (z,a32,b32,c32,x);
nadder #(.size(128)) a3 (z,biga,bigb,bigc,x);

// The ram example has 4 parameters

ram #(.twdh(2),.trd(7)) r1(address, data, read,write);

endmodule
```

Values of parameters in module instances

A parameter that is not passed will have its default value. If a parameter value is an equation such as *W* in Example 14-4, the value is recalculated if any of the values in the equation are passed. It is possible to bypass the equation by passing a value to a calculated parameter.

This Page Intentionally Left Blank

15 STATE MACHINES

STATE MACHINE TYPES

There are two types of state machines: Mealy machines and Moore machines. You can model both types of machines in Verilog. The difference between Mealy and Moore machines is in how outputs are generated. In a Moore machine, the outputs are a function of the current state. This implies that the outputs from the Moore machine are synchronous to the state changes. In a Mealy machine, the outputs are a function of both the state and the inputs.

A state machine can be broken down into three parts: The state register, the next-state logic, and the output logic.

A state machine can be depicted as shown in Figure 15-1.
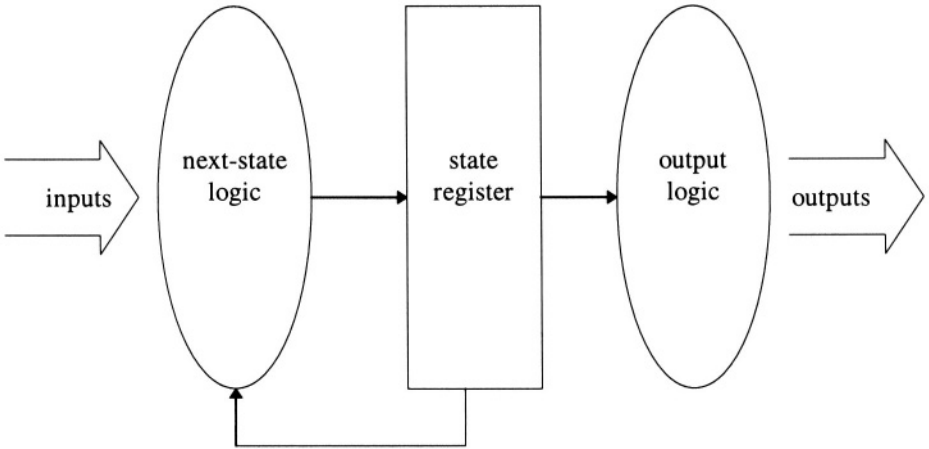
Figure 15-1 Moore State Machine

Figure 15-1 can be modified to bring the inputs through to the output logic, thus creating a Mealy state machine, as shown in Figure 15-2.
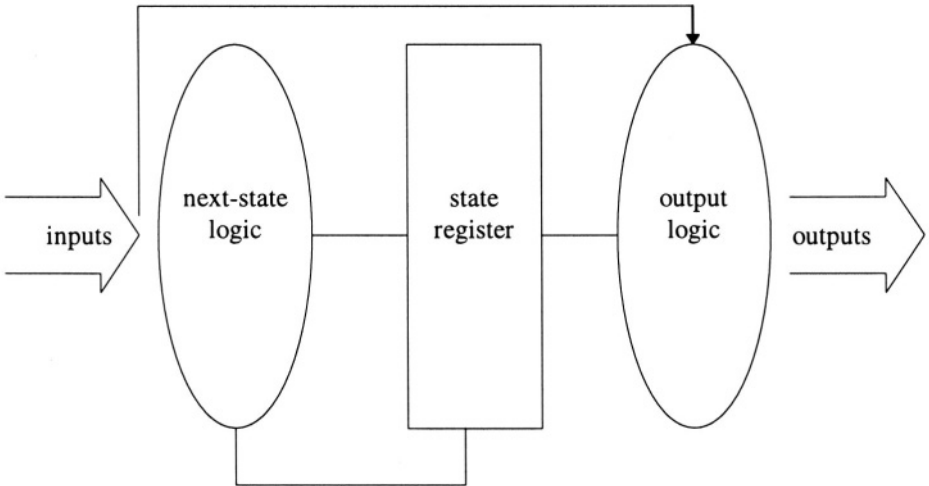
Figure 15-2 Mealy State Machine

To model a state machine in Verilog, you must model each of the three parts of the state machine.

STATE MACHINE MODELING STYLE

Because a state machine is made up of three parts, you have a choice whether to model each section independently, or to try to combine the parts into one section of the model.

Table 15-1 shows some interesting combinations:

Table 15-1 State Machine Styles

Style	State Register	Next-State Logic	Output Logic
1	Separate	Separate	Separate
2	Combined	Combined	Separate
3	Separate	Combined	Combined
4	Combined	Combined	Combined
5	Combined	Separate	Combined

In the first style, each of the functional blocks is modeled in a separate *always* block. The state register logic is modeled as an *always* block, and the next-state logic and output logic are separate *always* blocks, representing combinatorial logic. Because the output section can be modeled as either sensitive only to changes on state or also sensitive to changes on inputs, you can use this to model both Mealy and Moore machines. This style is the most modular; it may take a few more lines of Verilog code, though it may be the easiest to maintain.

The second style combines the next-state logic and the state register. This style is a good style to use because the next-state logic and state register are strongly related. This style is more compact than the first, and may even be more efficient because the next-state logic is only evaluated on clock edges, rather than whenever an input changes. If your state machine has many inputs that change frequently, this may be a better style to use than the first. This style has the output as a separate section so you can use this style to model both Mealy and Moore machines.

The third style leaves the state register in a separate *always* block, while combining the next-state logic and the output logic. Because the next-state logic and the output logic may be combinatorial, combining them still allows for modeling both Mealy and Moore machines. However, this grouping does not tend to help the readability of your code; styles 1 and 2 are easier to model and maintain.

The fourth style combines everything into one *always* block. The *always* block is sensitive only to the clock for the state register, so this implies the outputs only change with the state. Thus this style will only create Moore machines.

The fifth and final style combines the state register and output logic into one *always* block, so again, this only creates Moore machines.

To demonstrate all these styles, with Moore and Mealy variations on them, we will use a simple example. This example won't be the traffic light controller, vending machine, or a completely trivial state machine, but instead an automatic food cooker. This cooker has a supply of food that it can load into its heater when requested. The cooker then unloads the food when the cooking is done.

Besides *clock,* the inputs to this state machine are *start* (which starts a load/cook/unload cycle); *temp_ok* (a temperature sensor that detects when the heater is done preheating); *done* (a signal from a timer or sensor that detects when the cooking cycle is complete); and *quiet* (a final input that selects if the cooker should beep when the food is ready).

The outputs from the machine are *load* (a signal that sends food into the cooker); *heat* (a signal that turns on the heating element, which has its a built-in temperature control); *unload* (a signal that removes the food from the cooker and presents it to the diner); and *beep* (a signal that alerts the diner when the food is done).

Example 15-1 Style 1 Moore State Machine

```verilog
module auto_oven_style_1_moore(clock, start, temp_ok, done,
  quiet, load, heat, unload, beep);
input clock, start, temp_ok, done, quiet;
output load, heat, unload, beep;
reg load, heat, unload, beep;
reg [2:0] state, next_state;
`define IDLE     'b000
`define PREHEAT  'b001
`define LOAD     'b010
`define COOK     'b011
`define EMPTY    'b100

// State register block
always @(posedge clock)
  state <= #(`REG_DELAY) next_state;
// next state logic
always @(state or start or temp_ok or done) begin
  next_state = state; // default to stay in current state
  case (state)
    `IDLE: if (start) next_state=`PREHEAT;
    `PREHEAT: if(temp_ok) next_state = `LOAD;
    `LOAD: next_state = `COOK;
    `COOK: if (done) next_state=`EMPTY;
    `EMPTY: next_state = `IDLE;
   default: next_state = `IDLE;
  endcase
end

// Output logic
always @(state) begin
  if(state == `LOAD) load = 1; else load = 0;
  if(state == `EMPTY) unload =1; else unload = 0;
  if(state == `EMPTY && quiet == 0) beep =1; else beep = 0;
  if(state == `PREHEAT ||
     state == `LOAD ||
     state == `COOK) heat = 1; else heat =0;
end
endmodule
```

In style 1, as shown in Example 15-1, each section of the state machine is modeled with a separate *always* block. Style 1 can be used to represent either a Moore machine or a Mealy machine for our automatic oven. The difference between the two styles is seen in the different behavior of the *quiet* input and *beep* output. With the Moore machine the diner must wait through the entire EMPTY state for the beeper to be quiet. The Mealy version of this is for those diners who want the beeper to sound, and then jump up to turn it off, as shown in Example 15-2.

Example 15-2 Style 1 Mealy State Machine

```
module auto_oven_style_1_mealy(clock, start, temp_ok, done,
  quiet, load, heat, unload, beep);
input clock, start, temp_ok, done, quiet;
output load, heat, unload, beep;
reg load, heat, unload, beep;
reg [2:0] state, next_state;
`define IDLE      'b000
`define PREHEAT   'b001
`define LOAD      'b010
`define COOK      'b011
`define EMPTY     'b100

// State register block
always @(posedge clock)
  state <= #(`REG_DELAY) next_state;
// next state logic
always @(state or start or temp_ok or done) begin
  next_state = state; // default to stay in current state
  case (state)
    `IDLE: if (start) next_state=`PREHEAT;
    `PREHEAT: if(temp_ok) next_state = `LOAD;
    `LOAD: next_state = `COOK;
    `COOK: if (done) next_state=`EMPTY;
    `EMPTY: next_state = `IDLE;
    default: next_state = `IDLE;
  endcase
end

// Output logic
always @(state or quiet) begin
  if(state == `LOAD) load = 1; else load = 0;
  if(state == `EMPTY) unload =1; else unload = 0;
  if(state == `EMPTY && quiet == 0) beep =1; else beep = 0;
  if(state == `PREHEAT ||
     state == `LOAD ||
     state == `COOK) heat = 1; else heat =0;
end
endmodule
```

Style 2 can also be used to express both Moore and Mealy machines. Example 15-3 and Example 15-4 are just slightly shorter because the next-state logic and register are combined into one block of code. This also eliminates the need for the *next_state* temporary variable.

Example 15-3 Style 2 Moore Machine

```
module auto_oven_style_2_moore(clock, start, temp_ok, done,
  quiet, load, heat, unload, beep);
input clock, start, temp_ok, done, quiet;
```

```
output load, heat, unload, beep;
reg load, heat, unload, beep;
reg [2:0] state;
`define IDLE      'b000
`define PREHEAT  'b001
`define LOAD      'b010
`define COOK      'b011
`define EMPTY     'b100

// State register block
always @(posedge clock)begin
  case (state)
    `IDLE: if (start) state=`PREHEAT;
    `PREHEAT: if(temp_ok) state <= #(`REG_DELAY) `LOAD;
    `LOAD: state <= #(`REG_DELAY) `COOK;
    `COOK: if (done) state=`EMPTY;
    `EMPTY: state <= #(`REG_DELAY) `IDLE;
    default: state <= #(`REG_DELAY) `IDLE;
  endcase
end
// Output logic
always @(state) begin
  if(state == `LOAD) load = 1; else load = 0;
  if(state == `EMPTY) unload =1; else unload = 0;
  if(state == `EMPTY && quiet == 0) beep =1; else beep = 0;
  if(state == `PREHEAT ||
     state == `LOAD ||
     state == `COOK) heat = 1; else heat =0;
end
endmodule
```

Example 15-4 Style 2 Mealy Machine

```
module auto_oven_style_2_mealy(clock, start, temp_ok, done,
  quiet, load, heat, unload, beep);
input clock, start, temp_ok, done, quiet;
output load, heat, unload, beep;
reg load, heat, unload, beep;
reg [2:0] state;
`define IDLE      'b000
`define PREHEAT  'b001
`define LOAD      'b010
`define COOK      'b011
`define EMPTY     'b100

// State register block
always @(posedge clock)begin
  case (state)
    `IDLE: if (start) state=`PREHEAT;
    `PREHEAT: if(temp_ok) state <= #(`REG_DELAY) `LOAD;
    `LOAD: state <= #(`REG_DELAY) `COOK;
    `COOK: if (done) state=`EMPTY;
```

```
    `EMPTY: state <= #(`REG_DELAY) `IDLE;
    default: state <= #(`REG_DELAY) `IDLE;
  endcase
end
// Output logic
always @(state or quiet) begin
  if(state == `LOAD) load = 1; else load = 0 ;
  if(state == `EMPTY) unload =1; else unload = 0;
  if(state == `EMPTY && quiet == 0) beep =1; else beep = 0;
  if(state == `PREHEAT ||
     state == `LOAD |
     state == `COOK) heat = 1; else heat =0;
end
endmodule
```

Style 3 combines the next-state logic and output logic. In Example 15-5, this modeling will result in a Mealy machine. It is possible to use this style to describe a Moore machine.

Example 15-5 Style 3 Mealy Machine

```
module auto_oven_style_3_mealy(clock, start, temp_ok, done,
  quiet, load, heat, unload, beep);
input clock, start, temp_ok, done, quiet;
output load, heat, unload, beep;
reg load, heat, unload, beep;
reg [2:0] state, next_state;
`define IDLE     'b000
`define PREHEAT 'b001
`define LOAD     'b010
`define COOK     'b011
`define EMPTY    'b100

// State register block
always @(posedge clock)
  state <= #(`REG_DELAY) next_state;

// next state logic
always @(state or start or temp_ok or done or quiet) begin
  next_state = state; // default to stay in current state
  case (state)
    `IDLE: if (start) next_state=`PREHEAT;
    `PREHEAT: if(temp_ok) next_state = `LOAD;
    `LOAD: next_state = `COOK;
    `COOK: if (done) next_state=`EMPTY;
    `EMPTY: next_state = `IDLE;
    default: next_state = `IDLE;
  endcase

//output logic
  if(state == `LOAD) load = 1; else load = 0;
```

```
   if(state == `EMPTY) unload =1; else unload = 0;
   if(state == `EMPTY && quiet == 0) beep =1; else beep = 0;
   if(state == `PREHEAT ||
      state == `LOAD ||
      state == `COOK) heat = 1; else heat =0;
end

endmodule
```

Style 4 combines everything into one big block, which yields a Moore machine.

Example 15-6 Style 4 Moore Machine

```
module auto_oven_style_4_moore(clock, start, temp_ok, done,
  quiet, load, heat, unload, beep);
input clock, start, temp_ok, done, quiet;
output load, heat, unload, beep;
reg load, heat, unload, beep;
reg [2:0] state;
`define IDLE    'b000
`define PREHEAT 'b001
`define LOAD    'b010
`define COOK    'b011
`define EMPTY   'b100

// State register block
always @(posedge clock)begin
  case (state)
    `IDLE: if (start) state=`PREHEAT;
    `PREHEAT: if(temp_ok) state <= #(`REG_DELAY) `LOAD;
    `LOAD: state <= #(`REG_DELAY) `COOK;
    `COOK: if (done) state=`EMPTY;
    `EMPTY: state <= #(`REG_DELAY) `IDLE;
    default: state <= #(`REG_DELAY) `IDLE;
  endcase
  if(state == `LOAD) load <= #(`REG_DELAY) 1;
    else load <= #(`REG_DELAY) 0;
  if(state == `EMPTY) unload <= #(`REG_DELAY) 1;
    else unload <= #(`REG_DELAY) 0;
  if(state == `EMPTY && quiet == 0) beep <= #(`REG_DELAY) 1;
    else beep <= #(`REG_DELAY) 0;
  if(state == `PREHEAT ||
     state == `LOAD ||
     state == `COOK) heat <= #(`REG_DELAY) 1;
    else heat <= #(`REG_DELAY) 0;
end
endmodule
```

Style 5 combines the state register and output sections, and results in either a Moore machine or registered outputs.

Example 15-7 Style 5 Moore Machine

```
module auto_oven_style_5_moore(clock, start, temp_ok, done,
quiet, load, heat, unload, beep);
input clock, start, temp_ok, done, quiet;
output load, heat, unload, beep;
reg load, heat, unload, beep;
reg [2:0] state, next_state;
`define IDLE     'b000
`define PREHEAT  'b001
`define LOAD     'b010
`define COOK     'b011
`define EMPTY    'b100

// State register block
always @(posedge clock) begin
  state <= #(`REG_DELAY) next_state;
// Output logic
  if(state == `LOAD) load <= #(`REG_DELAY) 1;
    else load <= #(`REG_DELAY) 0;
  if(state == `EMPTY) unload <= #(`REG_DELAY) 1;
    else unload <= #(`REG_DELAY) 0;
  if(state == `EMPTY && quiet == 0) beep <= #(`REG_DELAY) 1;
    else beep <= #(`REG_DELAY) 0;
  if(state == `PREHEAT ||
     state == `LOAD ||
     state == `COOK) heat <= #(`REG_DELAY) 1;
    else heat <= #(`REG_DELAY) 0;

end

// next state logic
always @(state or start or temp_ok or done) begin
  next_state = state; // default to stay in current state
  case (state)
    `IDLE: if (start) next_state=`PREHEAT;
    `PREHEAT: if(temp_ok) next_state = `LOAD;
    `LOAD: next_state = `COOK;
    `COOK: if (done) next_state=`EMPTY;
    `EMPTY: next_state = `IDLE;
    default: next_state = `IDLE;
  endcase
end

endmodule
```

With all of these styles to choose from, which one is best? Which one will result in the smallest synthesized circuit? These are not easy questions to answer. Style 2 is both compact and allows for both Mealy and Moore machines, so this is a good all-around style to use. As for synthesized results, state encoding will have a greater effect on ultimate size than any of these variations in style.

STATE ENCODING METHODS

State encoding can have a great effect on circuit size and performance, and can also influence the amount of glitching produced by a circuit. With all that said, you should note that most synthesis tools can encode or re-encode states.

The state encoding used in the previous examples is a simple sequential numbering, as shown in Table 15-2.

Table 15-2 Sequential State Encoding

State	Code
IDLE	000
PREHEAT	001
LOAD	010
COOK	011
EMPTY	100

A common sense approach to state encoding might be to assume that the *heat* output needs to be on for the states PREHEAT, LOAD, and COOK. So the states could be encoded with one of the bits set for all of those states. This would have the effect of simplifying the output logic. The *heat* output is now simplified to *state[2]*.

Table 15-3 Mapping State Code To Simplify Outputs

State	Code
IDLE	000
PREHEAT	100
LOAD	111
COOK	110
EMPTY	001

Another approach that may minimize glitching is to "Gray code" the state encoding. "Gray code" is another method of binary counting; in Gray code during each transition, only one bit changes. This is easy for some state machines and difficult or impossible for state machines that branch in many directions. For the automatic oven, we could encode the states as shown in Table 15-4.

Table 15-4 Gray State Encoding

State	Code
IDLE	000
PREHEAT	100
LOAD	110
COOK	111
EMPTY	101

In this encoding between each state, only one bit changes (either sets or clears). The only transition in this model that violates this Gray code rule is the transition from EMPTY to IDLE, during which two of the bits clear.

For each of the encodings shown in Table 15-2, Table 15-3, and Table 15-4, only three flip-flops are used to encode the states. Because the state machine has five states, the minimum number of flip-flops to use is three. If you are not concerned about using the minimum number of flip-flops, there are other encodings you can use.

If you want to get the outputs out as quickly as possible, we can re-encode the states to an encoding of *output = state*. To start the *output = state* encoding, let's look at all the states in reference to the outputs, as shown in Table 15-5.

Table 15-5 States Compared with Outputs

State	LOAD	HEAT	UNLOAD	BEEP
IDLE	0	0	0	0
PREHEAT	0	1	0	0
LOAD	1	1	0	0
COOK	0	1	0	0
EMPTY	0	0	1	1

Now we can add a state encoding to Table 15-5, yielding the data in Table 15-6.

Table 15-6 Outputs as State Code

State	State Code	LOAD	HEAT	UNLOAD	BEEP
IDLE	00000	0	0	0	0
PREHEAT	01000	0	1	0	0
LOAD	11000	1	1	0	0
COOK	01010	0	1	0	0
EMPTY	00101	0	0	1	1

This state encoding has a bit for each output and extra bits for states that do not have unique outputs. The states PREHEAT and COOK both have the same outputs, so they need to have two different encodings. The last bit, used for the BEEP output, can be removed for optimization since it is the same as UNLOAD. This method of encoding uses four or five flip-flops, so it yields a larger circuit than do the other encoding methods.

A final state encoding method is called "one-hot." In this encoding method there is exactly one flip-flop set per state. This method may take even more flip-flops, but can sometimes produce faster circuits. One-hot state encoding is shown in Table 15-7.

Table 15-7 One-Hot State Encoding

State	Code
IDLE	10000
PREHEAT	01000
LOAD	00100
COOK	00010
EMPTY	00001

Which state encoding is best for a particular design depends on your design goals. Does the design need to be fast or small? Is glitching a concern? Is simultaneous switching a concern for power and glitching? These are the questions that most influence the choice of a state encoding.

DEFAULT CONDITIONS

Each of the state machine examples included a *default* clause. Those examples used 3 bits for state, but only five states were used. Thus it may be possible for the state machine to glitch into one of the three remaining illegal states. One other reason for

including the *default* clause is that when simulation starts, the state machine is in an unknown state, and the *default* clause gets it on track with the first clock.

The *default* clause may cause synthesis to generate more logic, so a trade-off must be made between the security of having a *default* clause and the potential size savings of not having it.

IMPLICIT STATE MACHINES

In all of the previous state machine examples, the three sections of the state machine were obvious (or explicit) in the coding style. The automatic oven design could also be coded as an implicit state machine. This style can be easier to code and maintain in an abstract model. In this style, only the behavior of the state machine is seen. The values of the outputs can also be seen, but the state register and next-state logic are implied.

Example 15-8 Implicit State Machine Style

```
module auto_oven_implicit(clock, start, temp_ok, done, quiet,
  load, heat, unload, beep);
input clock, start, temp_ok, done, quiet;
output load, heat, unload, beep;
reg load, heat, unload, beep;
initial begin // set all outputs to the off state
    load = 0;
    heat = 0;
    unload = 0;
    beep = 0;
end

always begin
    @(posedge clock) ; //stay IDLE at least one clock
    while( ! start) @(posedge clock) ;
                                    // do nothing until a start
    heat =1 ; // turn on heating element
    load = 1;
    @(posedge clock) ; //stay PREHEAT at least one clock
    while( ! temp_ok) @(posedge clock) ;  // wait to heat up
    load = 1;
    @(posedge clock) load = 0;
    @(posedge clock) ;
                    // stay in COOK at least one clock cycle
    while( ! done) @(posedge clock) ;
                // wait to finish cooking
    heat = 0;
    unload = 1;
    if (! quiet) beep = 1;
    @(posedge clock) unload = 0;
```

```
      beep =0;
end
endmodule
```

You have also already seen two other examples of implicit state machines. In chapter 6, when introducing the looping constructs, the modules *shift1* and *onecount* were simple, synthesizable implicit state machines.

REGISTERED AND UNREGISTERED OUTPUTS

The modeling of the output section of the state machine may infer that the outputs are either registered, or combinatorial. Registered outputs are available sooner after the clock and are less subject to glitching. One great disadvantage of registered outputs is that the machine becomes larger due to the extra flip-flops. Because the outputs are synchronous, this implies that machines with registered outputs are only Moore machines. Changing the Verilog for the output section to be registered instead of combinatorial is a simple matter of changing the *always* block to execute only on a clock, rather than a change on any input.

Example 15-9 Combinatorial Outputs

```
 always @(state or quiet) begin
  if(state == `LOAD) load = 1; else load = 0;
  if(state == `EMPTY) unload = 1; else unload = 0;
  if(state == `EMPTY && quiet == 0) beep = 1; else beep = 0 ;
  if(state == `PREHEAT ||
     state == `LOAD ||
     state == `COOK) heat = 1; else heat =0;
end
endmodule
```

Example 15-10 Registered Outputs

```
always @(posedge clock) begin
  if(state == ` LOAD)
    load <= #(` REG_DELAY) 1;
  else
    load <= #(`REG_DELAY) 0;
  if(state == `EMPTY)
    unload <= #(` REG_DELAY) 1;
  else
    unload <= #(` REG_DELAY) 0;
  if(state == ` EMPTY && quiet == 0)
    beep <= #(`REG_DELAY) 1;
  else
```

```
   beep <= #(`REG_DELAY) 0;
  if(state == `PREHEAT ||
     state == `LOAD ||
     state == `COOK)
    heat <= #(`REG_DELAY) 1;
  else
    heat <= #(`REG_DELAY) 0;
end
endmodule
```

Example 15-10 will produce its outputs delayed by one clock. This may work for some machines, but it may make others out of sync.

Since it is highly desirable to have registered outputs for high speed machines or to eliminate glitching, a modified Moore machine may be used to create registered outputs in sync with the state machine.
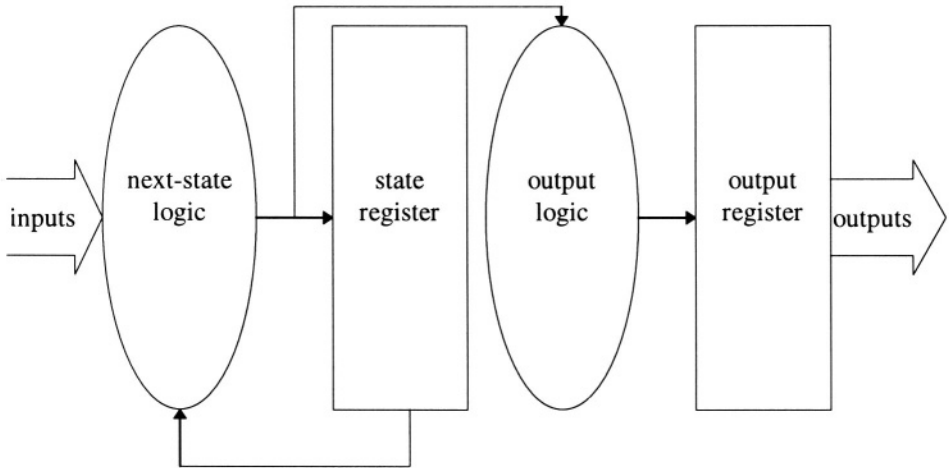
Figure 15-3 Modified Moore Machine

The modified Moore machine allows for registered outputs as shown in Example 15-11.

Example 15-11 Modified Moore Machine with Registered Outputs

```
always @(posedge clock) begin
  if(next_state == ` LOAD)
    load <= #(`REG_DELAY) 1;
  else
    load <= #(`REG_DELAY) 0;
  if(next_state == `EMPTY)
    unload <= #(`REG_DELAY) 1;
  else
    unload <= #(`REG_DELAY) 0;
  if(next_state == `EMPTY && quiet == 0)
    beep <= #(`REG_DELAY) 1;
  else
    beep <= #(`REG_DELAY) 0;
  if(next_state == `PREHEAT |
     next_state == `LOAD |
     next_state == `COOK)
    heat <= #(`REG_DELAY) 1;
  else
    heat <= #(`REG_DELAY) 0;
end
endmodule
```

FACTORS IN CHOOSING A STATE MACHINE MODELING STYLE

There are many ways to model a state machine. How do you know which style to use and how to encode your states? There are a number of different goals you may have for your state machine: Maximum frequency, area, clock-to-output delay, glitch-free outputs, minimal input setup time, minimum simultaneous switching, minimal power, or ease of maintenance. Each of these goals dictates a different style or encoding method.

For a state machine to have the highest possible frequency, the next-state logic must be as small as possible. In a logic type such as CMOS (where AND gates are fast), one-hot encoding may generate the fastest next-state logic because each state bit is usually set from the outputs of only AND gates. In other state encodings, there tend to be AND/OR networks driving each state bit.

For minimal clock-to-output delay, a state machine where the outputs come directly from flip-flops is best. The output-equals-state encoding or registered output style state machines both have outputs that come directly from flip-flops, making either of these styles the best choices for state machines requiring fast clock-to-output times.

For minimal simultaneous switching and minimal power in CMOS, a state machine with the states Gray-coded might be the best solution. The Gray-coded state machines only change 1 bit per transition.

If the state machine is visualized as a flow, the implicit state machine makes it easy to model and maintain. Inserting or removing behavior from this type of state machine is easiest.

If high speed or glitch free outputs are desired, a style 1 modified Moore Machine may be the best choice.

16 MODELING TIPS

Most circuits are either sequential or combinatorial. The Verilog constructs can be used to model both combinatorial and sequential circuits, and it is possible to create models in Verilog that are neither combinatorial or sequential. (However, if a model is neither sequential nor combinatorial it may not be possible to built it in actual hardware.) This chapter provides modeling rules for both combinatorial as well as sequential circuits.

In addition to presenting styles for combinatorial and sequential circuits, this chapter presents tips for synchronous and asynchronous circuits such as one-shots, and special-purpose models such as two-dimensional arrays.

MODELING COMBINATORIAL LOGIC

Combinatorial logic is always synthesizable. What Verilog constructs can you use for combinatorial logic? You can use nets (*wire, tri, tri1, tri0, trior, wand,* etc.) for modeling combinatorial logic. You can even use Verilog *regs* for modeling combinatorial logic!

Combinatorial Models Using Continuous Assignments

What constructs can you use to assign a value to a net? You can only use the *continuous assignment*. The continuous assignment always models combinatorial logic. (There is one exception, which is shown in Example 16-5.) Example 16-1, Example 16-2, Example 16-3 and Example 16-4 show simple, common combinatorial circuits.

Example 16-1 A 2-to-1 Mux Using Continuous Assignment

```
module mux2ca( y, sel, a, b) ;
output [3:0] y;
input [3:0] a, b;
input sel;

assign y = sel ? a : b;

endmodule
```

Example 16-1 shows a 2-to-l mux modeled with a continuous assignment. One advantage of this model is that it could be used to model a mux of any bus width. In this case, a 4-bit wide mux is shown.

Example 16-2 A 4-to-1 Mux Using Continuous Assignment

```
module mux4ca( y, sel, a, b, c, d) ;
output [3:0] y;
input [3:0] a, b, c, d;
input [1:0] sel;

assign y = sel[1] ? (sel[0] ? d : c)  :  (sel[0] ? b : a) ;

endmodule
```

Example 16-2 shows a simple 4-to-l mux using a continuous assignment, using nested ternary operators.

The continuous assignment for a 4-to-l mux has a more complex expression, and is a bit harder to read. An alternate expression for the 4-to-l mux is shown in Example 16-3. With both of these 4-to-l mux examples, if the *sel* input is unknown the output is *d*.

Example 16-3 Alternate 4-to-1 Mux Using Continuous Assignment

```
module mux4caa( y, sel, a, b, c, d);
output [3:0] y;
input [3:0] a, b, c, d;
input [1:0] sel;

assign y = (sel == 2'b00) ? a :
           (sel == 2'b01) ? b :
           (sel == 2'b10) ? c : d;

endmodule
```

Example 16-4 An 8-Bit Adder Using Continuous Assignment

```
module adder8ca(carry_out, sum, a, b, carry_in);
output carry_out;
output [7:0] sum;
input  [7:0] a, b ;
input  carry_in;

assign {carry_out,sum} = a + b + carry_in;
endmodule
```

The 8-bit adder from exercise 2 in chapter 3 could be simplified into a single continuous assignment, as shown in Example 16-4.

Example 16-5 Latch Using Continuous Assignment

```
module calatch(out, data, enable);
output out;
input data, enable;

assign out = enable ? out : data;

endmodule
```

The continuous assignment in Example 16-5 forms a level-sensitive latch, a mux feeding back into itself. Is this example combinatorial or sequential? (Answer: It's a bit of both!) Because this continuous assignment forms a feedback loop to itself, some synthesis tools may not allow it.

Combinatorial Models Using the *always* Block and *regs*

You can also use the *always* block to model combinatorial logic. If an *always* block runs in zero time, and runs whenever any of the inputs changes, it is modeling

combinatorial logic. The *reg* data types are used in *always* blocks, but *regs* do not always imply that a latch or flip-flop or some other sequential logic is being modeled. As you will see, a *reg* can be used for combinatorial modeling. The examples in this section illustrate the 2-to-1 mux, the 4-to-1 mux, and the 8-bit adder using *always* blocks and *regs*.

If you intend to model combinatorial logic using the *always* block, be sure to model all branches in your logic, or you may imply a latch. All *if* statements must have *else* clauses. All *case* statements must either have a *default* clause or all cases specified. Otherwise, a latch will be implied, and you will not have combinatorial logic.

Example 16-6 The 2-to-1 Mux Using *always*

```
module mux2r( y, sel, a, b) ;
output [3:0] y;
input [3:0] a, b;
input  sel;

reg [3:0] y;
always @(sel or a or b)
    if(sel)
        y = a;
    else
        y = b;

endmodule
```

Example 16-6 is functionally equivalent to Example 16-1. Both are synthesizable 2-to-1 muxes. Review Chapters 8 and 10, and you will find the models behave differently when *sel* is *x;* However synthesis will produce similar hardware for both models.

In Example 16-7 the *mux4* is modeled in a procedural block. You can use a *case* statement to improve readability. Notice how Example 16-7 is easier to read than Example 16-2, in which a continuous assignment was used.

Example 16-7 The 4-to-1 Mux Using *always*

```
module mux4r( y, sel, a, b, c, d) ;
output  [3:0]  y;
input [3:0] a, b, c, d;
input [1:0] sel;

reg  [3:0]  y;
always @(sel or a or b or c or d)
    begin
    case  (sel)
        0:  y = a;
        1:  y = b;
        2:  y = c;
        3:  y = d;
    endcase
end

endmodule
```

The adder in Example 16-8 is still combinatorial even though it uses an integer, the *repeat* loop, and the *if* statement. This is definitely not the best way to model an adder, but it shows how many constructs can be used to model a combinatorial circuit.

Example 16-8 The 8-Bit Adder Using *always*

```
module adder8r(carry_out,  sum,  a,  b,  carry_in);
output carry_out;
output [7:0]  sum;
input   [7:0]  a, b ;
input carry_in;

reg  [7:0]  sum;
reg carry_out;
integer temp;
always @(a or b or carry_in) begin
    temp = a;
    repeat  (b)  begin
        temp = temp + 1;
    end
    if( carry_in) temp = temp + 1;
    carry_out = temp[8];
    sum = temp[7:0] ;
end

endmodule
```

Example 16-9 is a simplified 8-bit adder using the *always* loop.

Example 16-9 Simplified 8-Bit Adder Using *always*

```
module adder8s(carry_out, sum, a, b, carry_in);
output carry_out;
output [7:0] sum;
input  [7:0] a, b ;
input carry_in;

reg [7:0] sum;
reg carry_out;
always @(a or b or carry_in)
   {carry_out, sum) = a + b + carry_in;
endmodule
```

Combinatorial Models Using Functions

Any combinatorial logic modeled with an *always* block can quickly be modified to be a *continuous assignment* and *function* combination. Example 16-7 can be rewritten as a continuous assignment with a function, as shown in Example 16-10.

Example 16-10 Mux with Continuous Assignment and Function

```
module mux4caf( y, sel, a, b, c, d) ;
output [3:0] y;
input [3:0] a, b, c, d;
input [1:0] sel;
assign y = muxf(a, b, c, d, sel);

function [3:0] muxf;
   input [3:0] a, b, c, d;
   input [1:0] sel;
     case (sel)
         0: muxf = a;
         1: muxf = b;
         2: muxf = c;
         3: muxf = d;
     endcase
endfunction

endmodule
```

Example 16-10 is the 4-to-l mux rewritten as a continuous assignment and function. With this simple example there is little advantage to choosing either modeling style. Some of the advantages of using the function and continuous assignment combination are as follows:

- Functions are always combinatorial, so there is no doubt about what type of logic is being modeled.
- The same function can be used more than once within the same module.
- If a behavioral model has blocks of similar logic, you can combine them into a common function that is called more than once.

In summary, the behavioral constructs to model combinatorial logic are the continuous assignment, the *always* block that triggers on any signal change, and the function.

MODELING SEQUENTIAL LOGIC

Sequential logic is most often modeled with *always* blocks. What other constructs can you use to model sequential logic? A flip-flop can be created with the built-in Verilog primitives or a sequential user-defined primitive (UDP). You can even use an *initial* block to model sequential logic.

Sequential Models Using *always*

A simple block of sequential code is shown in Example 16-11. Most simple sequential logic is just a block of code that is synchronous with a clock. This model is not a good model for synthesis due to the lack of reset and use of the *initial* statement. Chapter 17 will discuss the interaction of *initial,* reset and synthesis.

Example 16-11 Simple Counter

```
`define REG_DELAY 1
module counter(clock, out);
input clock;
output [7:0] out;
reg [7:0] out;
initial out <= #(`REG_DELAY) 0;
always @(posedge clock) out <= #(`REG_DELAY) out + 1;
endmodule
```

Sequential Models Using *initial*

Sequential logic does not have to be started with an *always* block. A counter can be modified by using only an *initial* statement and still function correctly. A counter using only an *initial* statement is shown in Example 16-12. It is important to note, even though this counter simulates identically to the previous example, It will not be synthesizable by most tools.

Example 16-12 A Counter without *always*

```
module counterf(clock, out);
input clock;
output [7:0] out;
reg [7:0] out;
initial
  begin
    out = 0;
    forever @(posedge clock) out <= #(`REG_DELAY) out + 1;
  end
endmodule
```

In Example 16-12, the *initial* block will never finish but the counter still functions correctly.

Sequential blocks are not limited to being synchronous to a clock. Sequential blocks can be used to generate clocks or other stimulus.

Example 16-13 Sequential Stimulus Block

```
module stimulus;
reg a, b, c;
initial
  begin
    a = 0; b = 0; c = 0;
    #100 a = 1;
    #100 b = 1;
    #500 c = 1;
  end
endmodule
```

Example 16-13 is considered to be a sequential block of code because it changes value in a sequence and time elapses between its statements.

Example 16-14 Clock Source

```
module clock_cource;
reg clock;
always
  begin
    #50 clock = 0;
    #50 clock = 1;
  end
endmodule
```

Example 16-14 is also sequential because it changes a value and time elapses.

A more complicated sequential block could use a combination of waiting for values to change and time delays.

Example 16-15 Memory Exerciser

```
module memex(clock, start, address, data,
                read, write, ready);
input clock, start, ready;
output [31:0] address;
inout  [31:0] data;
output read, write;
reg read, write, running, drive_data;
reg [31:0] data_reg, address;

initial
  begin
    running =0;
    drive_data = 0;
    read = 1'bz;
    write = 1 'bz;
    data_reg = 32'bz;
    address = 32'bz;
  end

assign data = drive_data ? data_reg : 32'bz;

always  @start
  if( start )
    @(posedge clock) begin
      running = 1;
      drive_data = 1;
      for( address = 0; address <= 32'hffff_ffff;
           address = address + 1)
        begin
          #10 data_reg = ~address;
          @(posedge clock) write = 1;
          # 40 write = 0;
          @(posedge ready) @(posedge clock) ;
        end
     drive_data = 0;
      for( address = 0; address <= 32'hffff_ffff;
           address = address + 1)
        begin
          @(posedge clock) read = 1;
          @(posedge  ready)
          if(data !== ~address)
            $display("memory error at location %h",  address);
          @(posedge clock) # 40 read = 0;
        end
      running = 0;
      read = 1'bz;
      write = 1'bz;
```

```
      address = 32'bz;
    end
endmodule
```

Example 16-15 is a memory exerciser. It waits for a start signal, then it waits for a clock. It cycles through each location in a memory and performs a write cycle and writes into each location data equal to the complement of the address. The write cycle is a sequence of events: First there is a time delay, then the data register is set. The model then waits for a *clock* and provides a pulse on the *write* signal. The memory responds with the *ready* signal when it is done with the write, so the model waits for the ready signal before continuing. The write sequence continues through all of the memory locations and then all of the location go through a read sequence.

Sequential Models Using Tasks

Most of the sequential behavior in the memory exerciser models either the read cycle or write cycle. The sequential code for the read and write cycles could be moved into tasks. With the complexity of the read and write cycles removed, the memory exerciser could then be expanded to run other memory test patterns. The new model for the memory exercise is shown in Example 16-16.

Example 16-16 Tasks for Sequential Code

```
module memex2(clock, start, address, data,
              read, write, ready);
input clock, start, ready);
output [31:0] address;
inout [31:0] data;
output read, write;
reg read, write, running, drive_data;
reg [31:0] data_reg, address;

initial
  begin
    running = 0;
    drive_data = 0;
    read = 1'bz;
    write = 1'bz;
    data_reg = 32'bz;
    address = 32'bz;
  end

assign data = drive_data ? data_reg : 32'bz;

always @start
  if( start )
```

```
    @(posedge clock) begin
      running = 1;
      drive_data = 1;
      for( address = 0; address == 32'hffff_ffff;
           address = address + 1)
           write_cycle(-address);
      drive_data = 0;
      for( address = 0; address == 32'hffff_ffff;
           address = address + 1)
        read_cycle( ~address);
      drive_data = 1;
      for( address = 0; address == 32'hffff_ffff;
           address = address + 1)
           write_cycle({32{address[0]}});
      drive_data = 0;
      for( address = 0; address == 32'hffff_ffff;
           address = address + 1)
        read_cycle({32{address[0]}});
      running = 0;
      read = 1'bz;
      write = 1'bz;
    end
task write_cycle;
input [31:0] data_in;
  begin
    #10 data_reg = data_in;
    @(posedge clock) write = 1;
    # 40 write = 0;
    @(posedge ready) @(posedge clock)  ;
  end
endtask

task read_cycle;
input [31:0] data_in;
  begin
    @(posedge clock) read = 1;
    @(posedge ready)
    if(data !== data_in)
      $display("memory error at location %h", address);
    @(posedge clock) # 40 read = 0;
  end
endtask

endmodule
```

Example 16-16 moves the read and write cycles into tasks. With the simplification that results from moving the complicated sequences into tasks, it is easy to expand the memory exerciser to write and test alternating patterns of 1s and 0s, in addition to the earlier test of using the complement of the address as the data.

In conclusion, *initial, always,* and *tasks* are used to model sequential code. Sequential code always uses *regs* and *begin-end* blocks. Combinatorial logic may also be modeled using all of these constructs, so the real distinction is that time advances in the block of code either by using the time delay operator (#) or by using the *wait for event* operator (@).

MODELING ASYNCHRONOUS CIRCUITS

The simplest asynchronous circuit is a one-shot. This section starts with a simple one-shot and expands into a full-blown asynchronous system and test bench.

Modeling a One-Shot

One-shots can be tricky to model. The *disable* statement in Verilog makes retriggerable one-shots easy to model. A basic one-shot with a positive edge trigger is shown in Example 16-17.

Example 16-17 Basic One-Shot

```
module oneshot(trigger,out);
input trigger;
output out;
reg out;
parameter timeConstant = 100;
initial out = 0;
always @(posedge trigger)
  begin
    out = 1;
    # timeConstant out = 0;
  end
endmodule
```

The one-shot in Example 16-17 may work for you, but it has a few potential problems. What happens if the trigger signal is held high? What happens if the trigger signal rises again during the time constant? For this one-shot, the answer to both of those questions is: Nothing happens. This one-shot starts timing at the first positive edge and then ignores the trigger signal until the time constant expires.

A retriggerable one-shot can be modeled. The retriggerable one-shot will still be edge sensitive, but if the trigger signal has a second rising edge during the time constant, the timing will start all over again. Thus, the output will fall only after the time constant expires after the last rising edge. The retriggerable one-shot is shown in Example 16-18.

Example 16-18 Retriggerable One-Shot

```
module roneshot(trigger,out);
input trigger;
output  out;
reg out;
event start;
parameter timeConstant = 100;
initial out = 0;

always @(posedge trigger)
  begin
    disable time_out;
    #0 -> start;
  end

always @start
  begin : time_out
    out = 1;
    # timeConstant out = 0;
  end
endmodule
```

The retriggerable one-shot makes use of a named *begin-end* block for the time constant. The named block can then be disabled with the *disable* statement. The first *always* block disables and then starts the *time_out* block to be sure that the entire time constant delay is used before *out* is set to 0. The *disable* will have no effect if the *time_out* block is not running, that is, in the case of a trigger that is not a retrigger. If the *time_out* block is running (that is, if it is a retrigger), the *disable* statement stops the block, *out* is not set to 0, and because the *time_out* block is in an *always* loop, the one-shot will be ready to run again. The signal of the event *start* will start the *time_out* block running. The *#0* allows the *always* to get to the *@start* so it can be retriggered.

Modeling Asynchronous Systems

You have already seen a model of a one-shot, which is the simplest of asynchronous circuits. This next set of examples will go through the entire design process of a slightly more complex asynchronous system. First, a simple behavioral model of the system will be developed. The behavioral model will use some of the less common behavioral constructs to develop a working model quickly. The working model will then be used to develop a comprehensive test bench. (A following chapter, Chapter 17, will go into more detail about test benches.) The comprehensive test bench will then be used to verify the detailed implementation model of the system. Because this is an asynchronous circuit, we cannot use synthesis tools to help with the implementation.

The system we will design is a simple self-arming alarm system. The alarm system has two inputs: *sense* and *disarm*. The alarm has two outputs, *armed* and *alarm*. The *sense* input is low whenever a door is open. When everything is secure, the *sense* input is high. The *disarm* input is momentarily low when the system should be disarmed. The *armed* output is high when the alarm is armed. The alarm output is high for five minutes after the alarm is armed and the *sense* input goes low. The alarm arms itself from the disarmed state when the *sense* input is high for five uninterrupted minutes. If the alarm output goes high, after five minutes, it goes low again, and waits five minutes for *sense* to go high before rearming.

This model will have three asynchronous features. The first is the five-minute timer, which itself is asynchronous because it is reset by the *sense* signal when the system waits for five minutes of no lows on the *sense* input. Secondly, the system is asynchronous because it will go from an armed state to an alarm state instantly when the *sense* input goes low. The final asynchronous feature is the *disarm* signal, which quiets the alarm output signal and sets the machine into a disarmed state instantly.
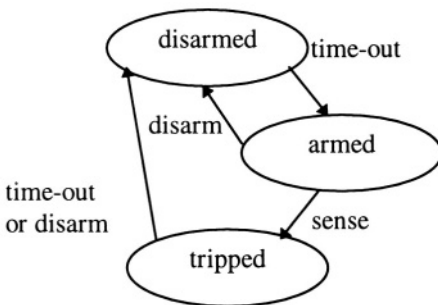
Figure 16-1 State Diagram for Alarm System

Using the state diagram shown in Figure 16-1 and the general textual description of the alarm, the alarm can be described in a Verilog behavioral model, as shown in Example 16-19.

Example 16-19 Behavioral Description of the Alarm

```
module alm(sense, disarm, armed, almOut);
input sense, disarm; // both active low
output armed, almOut;

// use parameters to describe some states
parameter S_disarmed = 0;
parameter S_armed = 1;
```

```verilog
parameter S_tripped = 2;

integer state;   // the state variable
initial state = S_disarmed;

reg timeout;     // the only other signal we should need.
initial timeout = 0;
event never;

// The alarm output is high when the alarm is tripped.
assign almOut = (state == S_tripped);

// The alarm is armed if in state armed or tripped
assign armed = (state == S_armed) || (state == S_tripped);

// all state changes are a result of the
// inputs or timeout signal
always @(sense or disarm or timeout)
  case(state)
    S_disarmed :
      casez({sense,disarm,  timeout})
        3'b111: state = S_armed;
      endcase

    S_armed :
      casez({sense,disarm, timeout})
        3'b?0?: state = S_disarmed;
        3'b01?: state = S_tripped;
      endcase

    S_tripped :
      casez({sense,disarm, timeout})
        3'b?0?: state = S_disarmed;
        3'b??1: state = S_disarmed;
      endcase
    endcase

// describe the asynchronous resettable time

// first the reset condition
always @(negedge sense or negedge disarm)
  if(state == S_disarmed || state == S_tripped)
    disable timer;

always begin : timer
  timeout = 0;
  wait(state == S_disarmed && sense == 1
       || state == S_tripped) // wait to start the timer
       # 5 timeout = 1;
       @never ; // wait forever after the timer times out;
 end

endmodule
```

The behavior of the alarm system can now be verified by creating a test bench and checking the behavior of the system. Once you have verified the system and test bench, it is time to describe the system's implementation more closely.

Example 16-20 Alarm Test Bench

```
module testAlm;

reg sense, dis;
alm dut(sense, dis, armed, almOut);

// sense and dis active low;
initial begin
  $monitor("%t: sense %b, disarm %b, armed %b, almOut %b",
        $time, sense, dis, armed, almOut);
  // first test that the alarm will disarm
  // and not arm too soon
  $display("initialize, and try disarm");
  sense = 1 ; dis = 0; // door closed, try to disarm
  #1 check(0,0);
  $display("Open and close door during auto arming period.");
  $display("Check that it does not arm too soon but does");
  $display("arm after 5 minutes of closed doors.");
  sense = 0 ; dis = 1; // door open not trying to disarm
  #10 check(0,0);
  sense = 1 ; // door closed
  #1 check(0,0);
  sense = 0 ; // door open
  #1 check(0,0);
  $display("Holding door open a long time.");
  #9 check(0,0);
  sense = 1 ; // door closed
  #4 check(0,0);

// allow a total of 6, the system should arm
  #2 check(1,0);

// try to disarm
  $display("Try to disarm.");
  sense = 1 ; dis = 0; // door closed, try to disarm
  #1 check(0,0);

// arm / disarm cycle works ok,
// allow it to rearm and see if it trips
  $display("arm/disarm cycle works, try to trip.");
  dis = 1;
  #4 check(0,0);

// allow a total of 6, the system should arm
  #2 check(1,0);
  $display("Trip and disarm.");
// trip the alarm and try to disarm
```

```
   sense = 0;
   #1 check(1,1);
   sense = 1;
   #1 check(1,1);
   dis = 0;
   #1 check(0,0);
   dis = 1;
   $display("rearm,  trip and test  timeout.");
   #4 check(0,0);
   #2 check(1,0);
// allow to rearm, trip check timeout
   sense = 0;
   #1 check(1,1);
   sense = 1;
   #1  check(1,1);
   #5  check(0,0);

   #100 $finish;
end

reg [ (8*5) :0] states[0:3];
initial begin
   states[0] = "IDLE";
   states[2] = "ARMED";
   states[3] = "ALARM";
end

task check;
input arm,out;
   if(armed!==arm | almOut !== out)
     begin
     $display(
       "ERROR\n%t: improper state, %b%b (%s), should be %s",
                 $time, armed, almOut,
                 states[{armed,almOut}], states[{arm,out}]);
     $stop;
     end
     else
     $display("%t: %s OK!",$time,states[{arm,out}]);
endtask

endmodule
```

The implementation is now possible because the desired behavior has been modeled and a comprehensive test bench has been developed. Because the final implementation will be based on some target technology, we will not attempt a final implementation here. What we will do is take a step towards implementation by using a combination of structural and behavioral modeling techniques. The timing element, a resistor, a capacitor, and a discharge transistor will be modeled behaviorally. Some SR flip-flops will be modeled structurally, and the "glue logic" that determines the functionality will be modeled with continuous assignments. The difficult part of the design will be creating the correct equations for the state

transitions. Because the state transition logic needs the most tuning to get it right, the continuous assignments will make modifying it easy. The continuous assignments will map easily into some final technology for final implementation.

Example 16-21 Partial Implementation of Alarm

```
`timescale 1s/1ms
module srff(sb, rb, q, qb);
input sb, rb;
output q, qb;
nand #0.001(q, sb, qb);
nand #0.001(qb, q, rb);
endmodule

module mytimer(start, out, disc);
// timer start active low, once released, delay will start
// output will rise after time constant
input start; // active low
output out;  // high when time expires
output disc; // high when discharging
parameter timeConstant = 5;
parameter disTime = 0.1;
reg cx; // abstraction of resistor - capacitor

buf #0.01 opa ( out, cx); // opamp or comparator
buf #0.01 opb ( down, cx); // opamp or comparator
nand #0.001 nq ( q, start, qb);
nand #0.001 nqb ( qb, down, q);
buf ( disc, q);

always @(posedge q) begin : discharge
  disable charge;
  # disTime cx = 0;
  end

always @(negedge q) begin : charge
  disable discharge;
  #timeConstant cx=1;
  end
endmodule

module alm(sense, dis, armed, almOut);
// sense active low
// dis disarm active low
input sense, dis;
output armed, almOut;
srff ff1(armb,r,armed, armedb);
srff ff2(tripb,r,almOut,almOutb);
srff ff3(almto, almclr, almrto, almrtob);
mytimer t2(tstartb, to, disc);

assign armb = ~(armedb & almOutb & sense & to);
```

```
assign tripb = ~(armed & almOutb & ~sense);
assign r= ~( ~dis | almrto);
assign tstartb = ~( almOutb & ~sense
                    || almOutb & ~dis || almrto);
assign almto = ~(armed & almOut & to & ~disc);
assign almclr = ~(armedb & almOutb & ~to & ~disc | ~dis);

endmodule
```

Example 16-21 shows a step toward the final implementation of the alarm. The model has enough structure that the final implementation can be easily mapped. The timer (which is partially analog) has a simple mapping of its elements into actual components. The voltage comparitors that determine the charge state of the capacitor are modeled as buffers. The resistor-capacitor timing element is modeled as behavioral code. The *set* and *reset* signals to each of the flip-flops are active low. Continuous assignments are used to model the logic that feeds the flip-flops. To make the equations as simple to write as possible, the bit-wise complement operator is used to invert each of the continuous assignments and make them active low. It is much easier to debug the circuit at this mixed behavioral and structural level than it would be to try and go directly from the behavioral model shown in Example 16-19 to some final implementation.

SPECIAL-PURPOSE MODELS

Two special-purpose modeling tips have demonstrated their usefulness over the years: multidimensional arrays and Z-detectors. The two-dimensional model can be expanded to three or more dimensions. The usefulness of the Z-detector will be explained as it is presented.

Two-Dimensional Arrays

Verilog supports only one-dimensional arrays of registers, integers, or times. What do you do if you need a two- (or more) dimensional array? There is an old programmer's trick that works well here. An index for a two-dimensional array can be computed as an index into a single-dimensional array. Example 16-22 shows a parameterized two-dimensional array of 32-bit registers.

Example 16-22 Two-Dimensional Array

```
module array2d;
parameter m = 10; // number of rows
parameter n = 12; // number of columns
reg [31:0] my_array[0:(m * n - 1)];

task store;
input [31:0] row, col, value;
  my_array[row * n + col] = value;
endtask

function [31:0] retrieve;
input [31:0] row, col;
  retrieve = my_array[row * n + col];
endfunction

endmodule
```

The *task* (for storing) and the *function* (for retrieving values from the two-dimensional array) are not required in a two-dimensional array. However, they are included in Example 16-22 to clarify how you index into the two-dimensional array.

Z-Detectors

Detecting Zs can have two uses. A common use for detecting Zs might be in a test bench that verifies when other modules should and should not be driving a signal. A more interesting use for detecting Zs would be to model special three-state logic. Some circuits use a special voltage that is applied to a pin to make the chip enter a programming, setup, or diagnostic mode. Because Verilog does not have a concept of voltage associated with a logic level, the value Z can be used to represent the special voltage when applying test vectors.

Example 16-23 Behavioral Z-Detector

```
module zdetb(in,out);
input in;
output out;
reg out;

always @in
  if( in === 1'bz)
    out = 1;
  else
    out = 0;
endmodule
```

Example 16-23 is a simple behavioral description for a Z-detector. If you were modeling a test bench that was going to check when a bus was being driven, the behavioral approach is the best solution. If you wanted to model the library cell that detects the special voltage on an input to put a chip in a special mode, a structural approach might be more appropriate.

Example 16-24 Structural Z-Detector

```
module zdets(in,out);
input in;
output out;
tri1 hi;
tri0 lo;
nmos(hi,in,1'b1);
nmos(lo, in,1'b1);
xor(out,hi,lo);
endmodule
```

Example 16-24 uses the properties of the *mos* switch primitives in Verilog to create a structural model for the Z-detector. The two unidirectional *nmos* transistors split the signal and feed it both to a *pulldown* and *pullup.* Rather than using *wire* type nets and *pullup* and *pulldown* primitives, the model can just use the *tri1* and *tri0* net types. When the input is driven, the same signal will drive both nets. When the input is not driven (Z), the two nets will go to different values, and the difference will be detected by the *xor* gate.

MULTIPLIER EXAMPLES

If your design requires a multiplier you will find there are many algorithms for multipliers. You must then choose how you want to implement your multiplier and what the trade-offs will be. There is a dizzying array of ways to build a multiplier. A few examples of multipliers and their trade-offs are presented in this section.

Example 16-25 An 8-by-8 Booth Multiplier

```
module boothmul (C, A, B);

   output [15:0] C; // definition of direction
   input   [7:0] A;
   input   [7:0] B;

   reg [15:0] STAGE0; // reduce width from stage to stage
   reg [15:2] STAGE1; // reduce width from stage to stage
   reg [15:4] STAGE2; // reduce width from stage to stage
   reg [15:6] STAGE3; // reduce width from stage to stage
```

```verilog
  wire [7:0]  NREG;    // start register N
  wire [15:0] MREG;    // start register M
  wire [15:0] M2REG;   // 2*M
  wire [15:0] NMREG;   // ~M+1
  wire [15:0] NM2REG;  // 2*(~M+1)

  assign NREG     = A;
  assign MREG     = {{8{B[7]}},B};

  // calculate the different operators. The values in the
registers
  // M2REG,NMREG,NM2REG are the results of the booth decoder.

  assign M2REG    = MREG<<1;
  assign NMREG    = ~MREG+1;
  assign NM2REG   = (~MREG+1)<<1;

  // build the references to the outputs

  assign C[0]  = STAGE0[0]    // 0 bits ready
  assign C[1]  = STAGE0[1]    // 1 bits ready
  assign C[2]  = STAGE1[2]    // 2 bits ready
  assign C[3]  = STAGE1[3]    // 3 bits ready
  assign C[4]  = STAGE2[4]    // 4 bits ready
  assign C[5]  = STAGE2[5]    // 5 bits ready
  assign C[6]  = STAGE3[6]    // 6 bits ready
  assign C[7]  = STAGE3[7]    // 7 bits ready
  assign C[8]  = STAGE3[8]    // 8 bits ready
  assign C[9]  = STAGE3[9];   // 9 bits ready
  assign C[10] = STAGE3[10];  // 10 bits ready
  assign C[11] = STAGE3[11];  // 11 bits ready
  assign C[12] = STAGE3[12];  // 12 bits ready
  assign C[13] = STAGE3[13];  // 13 bits ready
  assign C[14] = STAGE3[14];  // 14 bits ready
  assign C[15] = STAGE3[15];  // 15 bits ready

  always @(NREG or MREG)
  begin
  // stage zero is calculated by a subset of calculation
  //rules because m<-1> = 0

    case ({NREG[1],NREG[0]})
      2'b00 : assign STAGE0 = 0;       // clear stage register
      2'b01 : assign STAGE0 = MREG;    // MREG into MSB
      2'b10 : assign STAGE0 = NM2REG;  // NM2REG into MSB
      2'b11 : assign STAGE0 = NMREG;   // NMREG into MSB
    endcase

    case ({NREG[3],NREG[2],NREG[1]}}
      3'b000 : assign STAGE1 = STAGE0[15:2];
      3'b001 : assign STAGE1 = STAGE0[15:2]+MREG[13:0];
      3'b010 : assign STAGE1 = STAGE0[15:2]+MREG[13:0];
```

```
      3'b011 : assign STAGE1 = STAGE0[15:2]+M2REG[13:0];
      3'b100 : assign STAGE1 = STAGE0[15:2]+NM2REG[13:0];
      3'b101 : assign STAGE1 = STAGE0[15:2]+NMREG[13:0];
      3'b110 : assign STAGE1 = STAGE0[15:2]+NMREG[13:0];
      3'b111 : assign STAGE1 = STAGE0[15:2];
    endcase

    case ({NREG[5],NREG[4],NREG[3]})
      3'b000 : assign STAGE2 = STAGE1 [15:4];
      3'b001 : assign STAGE2 = STAGE1 [15:4]+MREG[11:0];
      3'b010 : assign STAGE2 = STAGE1 [15:4]+MREG[11:0];
      3'b011 : assign STAGE2 = STAGE1 [15:4]+M2REG[11:0];
      3'b100 : assign STAGE2 = STAGE1 [15:4]+NM2REG[11:0];
      3'b101 : assign STAGE2 = STAGE1 [15:4]+NMREG[11:0];
      3'b110 : assign STAGE2 = STAGE1 [15:4]+NMREG[11:0];
      3'b111 : assign STAGE2 = STAGE1 [15:4];
    endcase

    case ({NREG[7],NREG[6],NREG[5]})
      3'b000 : assign STAGE3 = STAGE2[15:6];
      3'b001 : assign STAGE3 = STAGE2[15:6]+MREG[9:0] ;
      3'b010 : assign STAGE3 = STAGE2[15:6]+MREG[9:0];
      3'b011 : assign STAGE3 = STAGE2[15:6]+M2REG[9:0];
      3'b100 : assign STAGE3 = STAGE2[15:6]+NM2REG[9:0];
      3'b101 : assign STAGE3 = STAGE2[15:6]+NMREG[9:0];
      3'b110 : assign STAGE3 = STAGE2[15:6]+NMREG[9:0];
      3'b111 : assign STAGE3 = STAGE2[15:6];
    endcase

  end
endmodule
```

A Booth multiplier like the one shown in Example 16-25 is good for a highly pipelined machine, with each of the stages being another level. However, this is a poor choice for a single-cycle multiplier.

The Booth multiplier can be modified to implement a Wallace multiplier, which is a fast way to design a single cycle multiplier as shown in Example 16-26.

Example 16-26 Wallace 8-by-8 Multiplier

```
module walmult8 (C,A,B);
  output [15:0] C;
  input  [7:0] A;
  input  [7:0] B;

  wire  [7:0] stage0_0 = {8{B[0]}} & A;
  wire  [7:0] stage0_1 = {8{B[1]}} & A;
  wire  [7:0] stage0_2 = {8{B[2]}} & A;
```

```verilog
  wire    [7:0]  stage0_3 = {8{B[3]}}  & A;
  wire    [7:0]  stage0_4 = {8{B[4]}}  & A;
  wire    [7:0]  stage0_5 = {8{B[5]}}  & A;
  wire    [7:0]  stage0_6 = {8{B[6]}}  & A;
  wire    [7:0]  stage0_7 = {8{B[7]}}  & A;

  wire    [9:0]  stage1_sum1;
  wire    [9:0]  stage1_carry1;
  wire    [9:0]  stage1_sum2;
  wire    [9:0]  stage1_carry2;

  wire    [10:0] stage2_sum1;
  wire    [10:0] stage2_carry1;
  wire    [12:0] stage2_sum2;
  wire    [12:0] stage2_carry2;

  wire    [14:0] stage3_sum;
  wire    [14:0] stage3_carry;

  wire    [15:0] stage4_sum;
  wire    [15:0] stage4_carry;

carry_save_add stage1_1(stage1_carry1, stage1_sum1,
                        {2'b0, stage0_0),
                        {1'b0, stage0_1, 1'b0},
                        {stage0_2, 2'b0});
carry_save_add stage1_2(stage1_carry2, stage1_sum2,
                        {2'b0, stage0_3},
                        {1'b0, stage0_4, 1'b0),
                        {stage0_5, 2'b0});

defparam stage2_1.WIDTH = 11;
carry_save_add stage2_l(stage2_carry1, stage2_sum1,
                        {1'b0, stage0_6, 2'b0},
                        {stage0_7, 3'b0},
                        (1'b0, stage1_carry2});
defparam stage2_2.WIDTH = 13;
carry_save_add stage2_2(stage2_carry2, stage2_sum2,
                        {3'b0, stage1_sum1},
                        {2'b0, stage1_carry1, 1'b0},
                        {stage1_sum2, 3'b0});

defparam stage3.WIDTH = 15;
carry_save_add stage3(stage3_carry, stage3_sum,
                        {2'b0, stage2_sum2},
                        {1'b0, stage2_carry2, 1'b0},
                        {stage2_sum1, 4'b0});

defparam stage4.WIDTH = 16;
carry_save_add stage4(stage4_carry, stage4_sum, {1'b0,
stage3_sum},
                        {stage3_carry, 1'b0},
                        {stage2_carry1, 5'b0});
```

```
assign C = stage4_sum + {stage4_carry, 1'b0);

endmodule

module carry_save_add(carry, sum, a, b, c) ;
parameter WIDTH = 10;
output [WIDTH-1:0] carry;
output [WIDTH-1:0] sum;
input [WIDTH-1:0] a;
input [WIDTH-1:0] b;
input [WIDTH-1:0] c;
integer i;
reg [WIDTH-1:0] carry;
reg [WIDTH-1:0] sum;
reg [1:0] result;

always@(a or b or c)
begin
    for(i = 0; i < WIDTH; i = i + 1)
    begin
        result = a[i] + b[i] + c[i];
        sum[i] = result[0];
        carry[i] = result[1];
    end
end
endmodule
```

Because the Wallace multiplier uses parameterized modules, it can easily be extended to a larger multiplier, as shown in Example 16-27.

Example 16-27 A 16-by-16 Multiplier

```
module walmul16 (C,A,B);
  output [31:0] C;
  input  [15:0] A;
  input  [15:0] B;

  wire   [15:0] stage0_0 = {16{B[0]}} & A;
  wire   [15:0] stage0_l = {16{B[1]>} & A;
  wire   [15:0] stage0_2 = (16{B[2]}} & A;
  wire   [15:0] stage0_3  = {16{B[3]}} & A;
  wire   [15:0] stage0_4 = {16{B[4]}} & A;
  wire   [15:0] stage0_5 = {16{B[5]}} & A;
  wire   [15:0] stage0_6 = {16{B[6]}} & A;
  wire   [15:0] stage0_7 = {16{B[7]}} & A;
  wire   [15:0] stage0_8 = {16{B[8]}} & A;
  wire   [15:0] stage0_9 = {16{B[9]}} & A;
  wire   [15:0] stage0_10 = {16{B[10]}} & A;
  wire   [15:0] stage0_11 = {16{B[11]}} & A;
  wire   [15:0] stage0_12 = {16{B[12]}} & A;
  wire   [15:0] stage0_13 = (16{B[13]}> & A;
```

```
  wire    [15:0]  stage0_14 = {16{B[14]}} & A;
  wire    [15:0]  stage0_15 = {16{B[15]}} & A;

  wire    [17:0]  stage1_sum0;
  wire    [17:0]  stage1_carry0;
  wire    [17:0]  stage1_sum1;
  wire    [17:0]  stage1_carry1;
  wire    [17:0]  stage1_sum2;
  wire    [17:0]  stage1_carry2;
  wire    [17:0]  stage1_sum3;
  wire    [17:0]  stage1_carry3;
  wire    [17:0]  stage1_sum4;
  wire    [17:0]  stage1_carry4;

  wire    [20:0]  stage2_sum0;
  wire    [20:0]  stage2_carry0;
  wire    [20:0]  stage2_sum1;
  wire    [20:0]  stage2_carry1;
  wire    [18:0]  stage2_sum2;
  wire    [18:0]  stage2_carry2;

  wire    [23:0]  stage3_sum0;
  wire    [23:0]  stage3_carry0;
  wire    [24:0]  stage3_sum1;
  wire    [24:0]  stage3_carry1;

  wire    [28:0]  stage4_sum0;
  wire    [28:0]  stage4_carry0;
  wire    [26:0]  stage4_sum1;
  wire    [26:0]  stage4_carry1;

  wire    [31:0]  stage5_sum;
  wire    [31:0]  stage5_carry;

  wire    [32:0]  stage6_sum;
  wire    [32:0]  stage6_carry;

carry_save_add  stage1_0(stage1_carry0, stage1_sum0,
                         {2'b0, stage0_0},
                         {1'b0, stage0_1, 1'b0},
                         (stage0_2, 2'b0});
carry_save_add  stage1_1(stage1_carry1, stage1_sum1,
                         {2'b0, stage0_3},
                         {1'b0, stage0_4, 1'b0},
                         {stage0_5, 2'b0});
carry_save_add  stage1_2(stage1_carry2, stage1_sum2,
                         {2'b0, stage0_6},
                         (1'b0, stage0_7, 1'b0),
                         {stage0_8, 2'b0});
carry_save_add  stage1_3(stage1_carry3, stage1_sum3,
                         {2'b0, stage0_9},
                         (1'b0, stage0_10, 1'b0},
                         {stage0_11, 2'b0});
carry_save_add  stage1_4(stage1_carry4, stage1_sum4,
```

```
                            {2'b0, stage0_12},
                            {1'b0, stage0_13, 1'b0},
                            {stage0_14, 2'b0});

defparam stage2_0.WIDTH = 21;
carry_save_add stage2_0(stage2_carry0, stage2_sum0,
                            {3'b0, stage1_sum1},
                            {2'b0, stage1_carry1, 1'b0},
                            {stage1_sum2, 3'b0});
defparam stage2_1.WIDTH = 21;
carry_save_add stage2_1(stage2_carry1, stage2_sum1,
                            {3'b0, stage1_carry2},
                            {1'b0, stage1_sum3, 2'b0},
                            {stage1_carry3, 3'b0});
defparam stage2_2.WIDTH = 19;
carry_save_add stage2_2(stage2_carry2, stage2_sum2,
                            {1'b0, stage1_sum4},
                            {stage1_carry4, 1'b0},
                            {stage0_15,3'b0});

defparam stage3_0.WIDTH = 24;
carry_save_add stage3_0(stage3_carry0, stage3_sum0,
                            {6'b0, stage1_sum0},
                            {5'b0, stage1_carry0, 1'b0},
                            {stage2_sum0, 3'b0});
defparam stage3_1.WIDTH = 25;
carry_save_add stage3_1(stage3_carry1, stage3_sum1,
                            {4'b0, stage2_carry0},
                            {1'b0, stage2_sum1, 3'b0},
                            {stage2_carry1, 4'b0}} ;

defparam stage4_0.WIDTH = 29;
carry_save_add  stage4_0(stage4_carry0,  stage4_sum0,
                            {5'b0, stage3_sum0},
                            {4'b0, stage3_carry0, 1'b0},
                            {stage3_sum1, 4'b0});
defparam stage4_1.WIDTH = 27;
carry_save_add stage4_1(stage4_carry1, stage4_sum1,
                            {2'b0, stage3_carry1},
                            {1'b0, stage2_sum2, 7'b0},
                            {stage2_carry2, 8'b0});

defparam stage5.WIDTH = 32;
carry_save_add stage5(stage5_carry, stage5_sum,
                            {3'b0, stage4_sum0},
                            {2'b0, stage4_carry0, 1'b0},
                            {stage4_sum1, 5'b0});

defparam stage6.WIDTH = 33;
carry_save_add stage6(stage6_carry,  stage6_sum,
                            {1'b0, stage5_sum},
                            {stage5_carry, 1'b0},
                            {stage4_carry1, 6'b0});
```

```
assign C = stage6_sum + {stage6_carry, 1'b0};
endmodule

module carry_save_add(carry, sum, a, b, c);
parameter WIDTH = 18;
output [WIDTH-1:0] carry;
output [WIDTH-1:0] sum;
input [WIDTH-1:0] a;
input [WIDTH-1:0] b;
input [WIDTH-1:0] c;
integer i;
reg [WIDTH-1:0] carry;
reg [WIDTH-1:0] sum;
reg [1:0] result;

always@(a or b or c)
begin
    for(i = 0; i < WIDTH; i = i + 1)
    begin
        result = a[i] + b[i] + c[i];
        sum[i] = result[0];
        carry[i] = result[1];
    end
end
endmodule
```

Example 16-26 and Example 16-27 are good examples of Wallace multipliers for positive numbers. But what about negative numbers? Example 16-28 shows the Wallace multiplier modified for negative numbers.

Example 16-28 A 16-by-16 Wallace Multiplier for Signed Numbers

```
module swalmul16 (c,a,b);

output [31:00] c;    // product (2's complement number)
input  [15:00] a;    // multiplier (2's complement #)
input  [15:00] b;    // multiplicand (2's complement #)

wire   [15:00] a;    // multiplier
wire   [15:00] b;    // multiplicand

wire   [15:00] ci;   // carry in bits required when recode is
                     // negative
wire   [16:00] r0;   // recoded multiplicands
wire   [18:02] r1;
wire   [20:04]  r2;
wire   [22:06] r3;
wire   [24:08] r4;
wire   [26:10] r5;
wire   [28:12] r6;
wire   [30:14] r7;
```

```
wire    [31:16] se;

// stage one results
wire    [20:00] s1_s1;
wire    [21:01] s1_c1;
wire    [26:04] s1_s2;
wire    [27:05] s1_c2;
wire    [31:10] s1_s3;
wire    [32:11] s1_c3;

// stage two results
wire    [26:00] s2_s1;
wire    [27:01] s2_c1;
wire    [31:05] s2_s2;
wire    [32:06] s2_c2;

// stage three results
wire    [31:00] s3_s1;
wire    [32:01] s3_c1;

// stage four results
wire    [31:00] sum;
wire    [32:01] carry;

// final product
wire    [31:0] c;

// look at multiplier and recode multiplicand
rcd2 rc0( a[01:00],b,r0,ci[01:00])
rcd3 rc1( a[03:01],b,r1,ci[03:02])
rcd3 rc2( a[05:03],b,r2,ci[05:04])
rcd3 rc3( a[07:05],b,r3,ci[07:06])
rcd3 rc4( a[09:07],b,r4,ci[09:08])
rcd3 rc5( a[11:09],b,r5,ci[11:10])
rcd3 rc6( a[13:11],b,r6,ci[13:12])
rcd3 rc7( a[15:13],b,r7,ci[15:14])

// create sign extention for 2's complement numbers
assign se = {~r7[30], 1'b1, ~r6[28], 1'b1, ~r5[26], 1'b1,
             ~r4[24], 1'b1, ~r3[22], 1'bl, -r2[20], 1'b1,
             ~r1[18], 1'b1, ~r0[16], 1'b0};

// three sections to create stage 1 results
defparam c1.WIDTH = 21;
carry_save_add c1(s1_c1,s1_s1,
                  {4'h1, // constant 1 is se carryin
                   se[16], ci[15:00]},
                  {2'h0, se[18:17], r0[16:00]},
                  (se[20:19], r1[18:02], 2'h0} ) ;
defparam c2.WIDTH = 23;
carry_save_add  c2(s1_c2,s1_s2,{4'h0,se[22:21],r2[20:04]},
                   {2'h0,se[24:23],r3[22:06],2'h0},
                      {se[26:25],r4[24:08],4'h0});
defparam c3.WIDTH = 22;
```

```
carry_save_add c3(s1_c3,s1_s3,{3'h0,se[28:27],r5[26:10]},
                  {1'h0,se[30:29],r6[28:12],2'h0},
                  {se[31], r7[30:14], 4'h0});

// two sections to create stage 2 results
defparam c4.WIDTH = 27;
carry_save_add c4(s2_c1,s2_s1,{6'h00,s1_s1[20:00]},
                                (5'h00,s1_c1[21:01],1'h0),
                                    {s1_s2[26:04],4'h0});
defparam c5.WIDTH = 27;
carry_save_add c5(s2_c2,s2_s2,{4'h0,s1_c2[27:05]},
                                {s1_s3[31:10],5'h00},
                                {s1_c3[31:11],6'h00));

// one section for stage 3 results
defparam c6.WIDTH = 32;
carry_save_add c6(s3_c1,s3_s1,{5'h00,s2_s1[26:00]},
                                {4'h0 ,s2_c1[27:01],1'h0 },
                                {s2_s2[31:05],5'h00});

// one section for stage 3 results
defparam c7.WIDTH = 32;
carry_save_add c7(carry,sum,s3_s1[31:00],
                            {s3_c1[31:01],1'h0 },
                            {s2_c2[31:06],6'h00});

// final stage should be carry lookahead adder
assign c = sum + {carry[31:01],1'b0};

endmodule

module rcd2(a,b,rec,ci);
input [2:1] a;
input [15:0] b;
output [16:0] rec;
output [1:0] ci;

// note that rcd2 is a case of rcd3 with lsb of a = 0


reg    [16:0] rec;
reg    [1:0] ci;
always @(a or b)
  case (a)
    0: begin rec = 17'h00000; ci = 0; end  // 00 ==>  0 * b
    1: begin rec = {b[15],b}; ci = 0; end  // 01 ==>  1 * b
    2: begin rec = {~b,1'b0}; ci = 2; end  // 10 ==> -2 * b
    3: begin rec = {~b[15],~b}; ci = 1; end  // 11 ==> -1 * b
    default: begin rec = 17'h00000; ci = 0; end
  endcase

endmodule
```

```
module rcd3(a,b,rec,ci);
input  [2:0]  a;
input  [15:0] b;
output [16:0] rec;
output [1:0]  ci;

reg     [16:0] rec;
reg     [1:0]  ci;
always @(a or b)
  case  (a)
    0:  begin rec = 17'h00000; ci = 0; end  // 000 ==>  0 * b
    1:  begin rec = {b[15],b}; ci = 0; end  // 001 ==>  1 * b
    2:  begin rec = {b[15],b}; ci = 0; end  // 010 ==>  1 * b
    3:  begin rec = {b,1'b0}; ci = 0; end   // 011 ==>  2 * b
    4:  begin rec = {~b,1'b0}; ci = 2; end  // 100 ==> -2 * b
    5:  begin rec = {~b[15],~b}; ci = 1; end // 101 ==> -1 * b
    6:  begin rec = {~b[15],~b); ci = 1; end // 110 ==> -1 * b
    7:  begin rec = 17'h00000; ci = 0; end  // 111 ==>  0 * b
     default: begin rec = 17'h00000; ci = 0; end
  endcase

endmodule

module carry_save_add(carry, sum, a, b, c);
parameter WIDTH = 21;
output [WIDTH-1:0] carry;
output [WIDTH-1:0] sum;
input [WIDTH-1:0] a;
input [WIDTH-1:0] b;
input [WIDTH-1:0] c;
integer i;
reg [WIDTH-1:0] carry;
reg [WIDTH-1:0] sum;
reg [1:0]  result;

always@(a or b or c)
begin
    for(i = 0; i < WIDTH; i = i + 1)
    begin
        result = a[i] + b[i] + c[i];
        sum[i] = result[0];
        carry[i] = result[1];
    end
end
endmodule
```

A Proven, Successful Approach to Modeling

The final tip in this chapter is the use of progressive refinement, as demonstrated with the alarm examples. First, a behavioral model of the desired system answers this question: "What do you think you are designing?" Next, you should develop a

set of tests to exercise the operating functions. This step answers the question: "Does it work the way you expect it to?" With these two questions answered, you can be confident about what you are designing, and you have a set of tests to prove that the design works correctly.

Once you know exactly what you are building and have a set of tests, you can start to work on more detailed models. Each of the more detailed models can be tested with the original tests to insure correct functionality. Finally, a model of the ultimate implementation can be verified with the original tests.

Many people are hesitant to start with a behavioral model because they think it may be a waste of time. They may think: "If I am not drawing schematics or writing synthesizable code, I am not working toward my final implementation. So why do it?" The time spent developing and testing a behavioral model pays off as a huge time savings in the design and debugging of the final model. It was much easier to debug the behavioral model of the alarm than to debug the partial implementation. Developing the test bench for the behavioral model was easier because it was easier to distinguish whether the error was in the test bench or in the model.

When you are working on a larger system with more than one functional block, the payoff of starting with behavior models is even more pronounced. Each block can be modeled rapidly at the behavioral level. The blocks can be tested together as a system. The interactions between functional blocks in a system are often the source of unforeseen behaviors. These unforeseen interactions can be observed and debugged. By debugging the system early in the design cycle, you can avoid redesign later in the design cycle. As you complete more detailed descriptions of each of the functional blocks, you can still test the entire system by using behavioral descriptions for some of the blocks and more detailed descriptions for others.

Using behavioral design and testing early in the design cycle will shorten your design cycle by building the right system the first time. It is much easier to build something correctly if you *know* you are building the correct thing.

17 MODELING STYLE TRADE-OFFS

This chapter might be subtitled, *Zen and the Art of Verilog Modeling.* This chapter marks a departure from the previous chapters (full of syntax and examples) into why you would want to make certain modeling decisions.

Two obvious questions in the construction of a model are, "Will it simulate quickly?" and, "Will it synthesize into what I want to build?" Although simulation performance and synthesis are two considerations for modeling (and the choices you make as you write a model), they are not the only considerations in choosing a modeling style.

FORCES THAT INFLUENCE MODELING STYLE

So far in this book, the only forces influencing your models have been "write it fast" or "use this new construct." As new constructs were introduced in this book, you were encouraged to use each of them in a model, so the "use this construct" force on modeling style is an artifact of the learning process. The most common force on modeling style is "write it fast." The old adage "haste makes waste" still applies today. A hastily written model may appear to have the correct functionality, but it

may be terribly inefficient for simulation, or it may not be synthesizable. As mentioned in Chapter 1, there are three basic reasons for using a hardware description language: Simulation, documentation, and synthesis. Therefore, these three are also forces that act on modeling style.
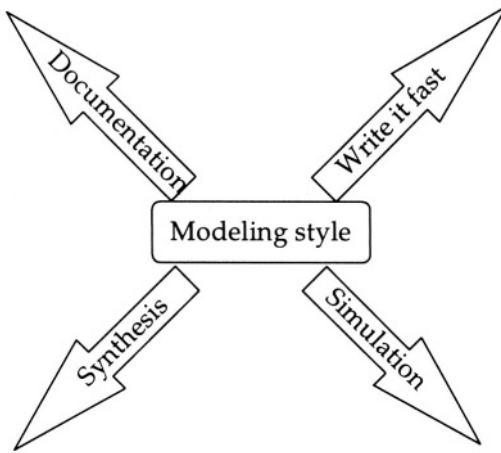
Figure 17-1 Forces That Act on Modeling Style

As shown in Figure 17-1, your modeling style may be affected by different forces. These forces may be pulling your modeling style in different directions. It would be naive to think there is only one force on modeling style, and that there is only one "correct" model for a given circuit. Unfortunately, many people have tried this "one correct model" approach and have not realized that they were costing themselves time and productivity.

Evolution of a Model

Every model starts with the write-it-fast approach, but where does the model go from there? You can hope that the model will be revised, move in the direction of documentation, and become a good reference for the design. Is a model that represents good documentation the final step in the evolution of model? No: You need to use the correct model for each job.

Consider for a moment a flip-flop: Which input changes more often, the data or the clock? Of course, the answer is dependent on the circuit that contains the flip-flop. In most cases, the clock input changes more often than the data. Sometimes the clock may change a thousand or more times before the data changes. You can apply this knowledge to a flip-flop model and see the impact on simulation performance.

Example 17-1 Normal D Flip-Flop

```
module ndff(q, clock, data);
input clock, data;
output q;
reg q;
always @(posedge clock)
  q <= #(`REG_DELAY) data;
endmodule
```

Example 17-1 shows the simple model for a normal D flip-flop we have been using. What happens if the clock changes one thousand times more often than the data? The expression $q = data$ would be executed one thousand times more than is needed. You can modify the module so that it is only executed when the data actually changes.

Example 17-2 Modified D-Flip-Flop

```
module mdff(q, clock, data);
input clock, data;
output q;
reg q;
always @data @(posedge clock)
  q <= #(`REG_DELAY) data;
endmodule
```

Example 17-2 shows how the flip-flop can be modified to wait first for the data to change and then wait for the clock to change. The two flip-flop models behave identically, but the second one may potentially be a thousand times faster. Before you get carried away and change all of your flip-flops, be aware that this trick has to be applied properly. If your data change at about the same rate as your clock, there may be no advantage to this modified flip-flop. You should also be aware that most modern simulators automatically optimize your flip-flops, so this model may actually go slower. This example is only meant to show that by taking into consideration the operating conditions of your models, you may be able to improve their simulation performance.

Modeling Style and Synthesis

There are some people who will argue, "If a model is not synthesizable, why write it?" The rebuttal is simple: "Why synthesize a circuit that has the wrong function?" Simulation has to come first. Simulation answers the important question: "Are you designing the correct system?" To answer that question the predominant forces on modeling style are "write it fast" and "simulation performance." Once you have verified that you are designing the correct system, you can make any changes to

your models that are needed for synthesis. If you start with synthesis as your only consideration, you may never answer the question, "Are you designing the correct system?"

Is It Synthesizable?

The answer to this question depends on which version of which tool you are using, and how you modeled your design. Unfortunately, the rules are not exactly the same for each of the available synthesis tools. Every construct in Verilog is synthesizable. Not all combinations of all constructs are synthesizable. With each release of a synthesis tool, more combinations of constructs become synthesizable.

There are some basic rules you can follow to insure that something is synthesizable. Change the question from "Is it synthesizable?" to "Is it combinatorial or sequential?" If your model is neither combinatorial nor sequential, or you can't determine which one it is, most likely the synthesis tool won't be able to either, and your model is not synthesizable. If each part of your circuit is either combinatorial or sequential, it should be synthesizable.

If you have a continuous assignment, it is combinatorial logic, so it is synthesizable. The only exceptions to this might be feedback, or complex operators like multiplication and division. *Functions* used to model combinatorial logic are also synthesizable. An *always* block that is executed whenever any of the inputs changes also models combinatorial logic and is synthesizable.

Sequential logic is also synthesizable, but not all sequential logic synthesizes. If you have only one edge of one clock, it is synthesizable. The one edge refers to the active clock edge. Asynchronous set and reset are synthesizable so it is legal to have three edges in the sensitivity list of the *always* statement..

These are most of the basic rules for determining whether a model is synthesizable. There are more detailed rules (such as how to model *reset)* that may vary from tool to tool. Figure 17-2 shows a simple flow chart to determine if a model is likely to be synthesizable.
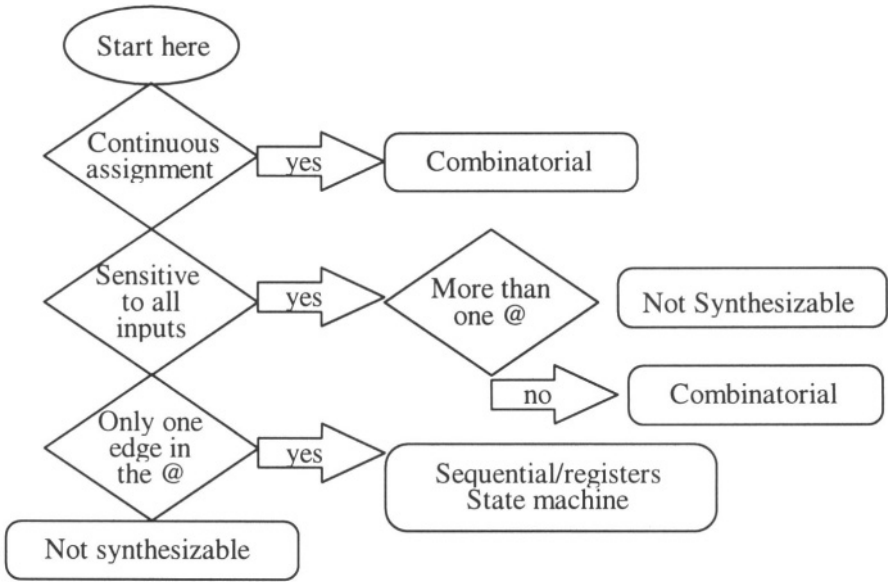
Figure 17-2 Synthesizablility flowchart

Appendix B lists all the modules in this book along with a checklist of their synthesizablility.

LEARNING FROM OTHER PEOPLE'S MISTAKES

With enough experience using Verilog, you will develop a skill for writing good models. This next set of examples is designed to help you avoid some common pitfalls. These examples highlight common modeling style errors made by novice Verilog users. Each example has its problems highlighted, with alternate improved models. By showing you these bad modules, I hope you can avoid these common problems.

Example 17-3 Bad Register

```
module reg16b(q, d, clk, clr_n);
input  [15:0] d;
input clk, clr_n;
output [15:0] q;
reg [15:0] q;

initial q = 0;

always
    begin
        if(clr_n != 0)
            @(posedge clk) #1 q = d;
        else
            #1 q = 0;
    end
endmodule
```

What are some of the problems with Example 17-3? Is the clear signal (*clr_n*) synchronous or asynchronous? What happens when *clr_n* is held low?

The most obvious problem is when *clr_n* is held low for a long time. This model executes at every time unit. This would be a huge waste of CPU time.

A more subtle problem is the case of how *clr_n* interacts with the clock (*clk*). This model creates a one-time-unit glitch on the output in some cases when clear is asserted. If clear is not asserted (*clr_n* = 1), the model waits for the clock, and the output (*q*) gets a new value one time unit after the clock. If the model is waiting for the clock, and then clear is asserted, the output does not change until the clock rises, whereupon the output will change to the then current value of the input (*d*). The *always* loop will recycle to the top, where it checks clear and then one unit later the output will go to 0.

This model also samples its input *d* one time unit after the clock. This is typically called negative setup time, and is generally undesirable.

Example 17-3 certainly is not efficient for simulation. It is not good documentation for the design because the behavior of reset is quite confusing. It can be improved in many ways.

Example 17-4 Improved Register

```
module reg16i(q, d, clk, clr_n);
input [15:0] d;
input clk, clr_n;
output [15:0] q;
reg [15:0] q;

always @(posedge clk)
    if(clr_n)
        q <= #1 d;
    else
        q <= #1 0;
endmodule
```

What improvements does Example 17-4 show over Example 17-3? Many! First, it is now easy to see that *clr_n* is synchronous because this model is only evaluated after the clock rises. Because the model does not needlessly set the output to 0 every time unit when *clr_n* is held low, the model will use less CPU time.

You may have noticed that the *begin-end* block is removed. The *begin-end* block is not needed because there is only one statement (the *if*) in the *always* block. Omitting the *begin-end* block around a single statement is a matter of preference. It is definitely not needed, though some designers may feel it improves readability. However, the extra *begin-end* block may waste a small amount of CPU time because it is an extra statement to compile or execute.

The expression for the *if* was also changed. The discussion for this change is similar to that for the removal of the *begin-end* block. It is a matter of preference. The meaning is unchanged, but the code is simplified.

The most significant change in functionality is the subtle change from *#1 q = d;* to *q <= #1 d;* This changes what was probably a subtle error in Example 17-3. In both cases there is a clock-to-output delay of one. However, the first case had negative setup time and the second case had zero setup time. Negative setup is rarely modeled intentionally. What is negative setup time? Without completely revisiting the subject of setup time, you should have learned in any logic design course that negative setup time simply means a change in the input data after the clock is propagated to the output. Normal setup time is described as follows: First the input is stable; then the clock rises; and finally the state of the input at or before the clock is propagated to the output. How did Example 17-3 accomplish negative setup time? First, the model waited for the clock, next, the model waited one more time unit; and then one unit after the clock, it sampled the value of the input (*d*) and propagated it to the output (*q*).

The negative setup time was corrected in Example 17-4 by using the intra-assignment delay. In this case, the model waites for the clock, sampled the input immediately, and then waites one time unit to propagate it to the output. The assignment is also changed from a blocking to a non-blocking assignment to be consistent with the preferred style.

The *initial* statement is also removed. Initializing flip-flops to a default value is generally a bad practice since it may lead to post-synthesis simulation mismatches. This flip-flop has a clear input which should be used to get the circuit into a known state.

The model in Example 17-4 is clearly an improvement, but is this the best possible example? It expresses the functionality clearly, so it is good documentation; it will synthesize; and it is more efficient for simulation. The model could be further tweaked for increased simulation performance.

Example 17-5 Tweaked Register

```
module reg16t(q, d, clk, clr_n);
input [15:0] d;
input clk, clr_n;
output [15:0] q;
reg [15:0] q;

always @(d or clr_n) @(posedge clk)
  if(clr_n)
    q <= #1 d;
  else
    q <= #1 0;
endmodule
```

Example 17-5 may be more efficient for simulation as explained at Example 17-2.

Example 17-6 Bad Adder

```
module add32b(o, a, b) ;
input [31:0] a, b;
output [31:0] o;
reg [31:0] o;

always
    #1 o = a + b;

endmodule
```

Example 17-6 is just a small step above the worst classic error. The worst classic error would have been to omit the *#1*. If the delay were omitted, this would be a zero-delay *always* loop. Because simulation time would not advance with a zero-delay *always* loop, nothing would work. In this case the novice user noticed the zero-delay *always* loop and added the *#1* to make the simulation work. Of course, this is huge waste of CPU time to recalculate a 32-bit addition at every time unit, independently of the inputs changing.

Example 17-7 Improved Adder

```
module add32i(o, a, b) ;
input [31:0] a, b;
output [31:0] o;
reg [31:0] o;

always @(a or b)
    #1 o = a + b;

endmodule
```

Example 17-7 is more efficient for simulation because the addition is performed only if one of the inputs changes.

Example 17-8 Adder Reduced to a Continuous Assignment

```
module add32ca(o, a, b) ;
input [31:0] a, b;
output [31:0] o;

assign #1 o = a + b;

endmodule
```

Example 17-8 further reduces the adder. The *reg* declaration is removed and the *always* block is replaced by a continuous assignment. Why write more code than you need? The continuous assignment is a fast and efficient way to model many functions.

A mux can also be modeled inefficiently by someone new to Verilog.

Example 17-9 Bad Mux

```
module mux16b(o, a, b, s) ;
input [15:0] a, b;
input s;
output [15:0] o;
reg [15:0] o;

initial o = 0;

always
    begin
        if( s == 1 )
            #1 o = b;
        else
            #1 o = a;
    end

endmodule
```

Example 17-9 repeats many errors you have seen before. It is evaluated at every time unit needlessly, and it wastes CPU time. It has an extraneous *begin-end*. It has an *initial* block that serves no useful purpose.

Example 17-10 Improved Mux

```
module mux16i(o, a, b, s);
input [15:0] a, b;
input s;
output [15:0] o;

assign #1 o = s ? b : a;

endmodule
```

Example 17-10 is only evaluated when the inputs change.

It is starting to look like using the continuous assignment is the most efficient way to model. However, modeling style is not as simple as that. There are cases in which the continuous assignment would be very inefficient to use. The continuous assignment is evaluated whenever any of the inputs change: If you have a complex expression with many inputs, a lot of unnecessary calculation may take place.

The final bad model is a barrel shifter.

Example 17-11 Bad Barrel Shifter

```
module bs32b(out, in, s);
input  [31:0]  in;
input [3:0] s;
output  [31:0]  out;

reg[31:0] out;
reg internal_clock;

initial out = 0;

initial
    fork
        internal_clock = 0;
        forever #10 internal_clock = ~internal_clock;
    join

always @(posedge internal_clock)
    begin
        out = in << s;
    end

endmodule
```

Example 17-11 repeats some style errors you have seen before, such as the extra *begin-end* block. You have seen the merits of removing extra *initial* blocks inserted for arbitrary initialization. This model evaluates the shift at every clock, independently of the inputs changing, and you have seen that error before as well.

The biggest inefficiency in Example 17-11 is the clock. In large systems, the clock is evaluated more often than anything else. Efficient generation and distribution of the clock can make a big difference in model performance. It does not make sense that each module in a system reproduces the clock; the clock should be generated externally and passed into the module. If you had a thousand or more instances of a module that generated the clock internally, you would spend most of your time in simulation generating a clock. The statement *internal_clock = ~internal_clock* makes this clock generator even more inefficient and time consuming because the computer must calculate the complement.

Finally, because Example 17-11 has an internal clock generator, it is not synthesizable.

Example 17-12 Improved Barrel Shifter

```
module bs32i(out, in, s, clock);
output [31:0] out;
reg [31:0] out;
input [31:0] in;
input [3:0] s;
input clock;

always @(posedge clock)
    out <= #(`REG_DELAY) in << s;

endmodule
```

Example 17-12 cleans up the barrel shifter in a number of ways. The clock is now passed in rather than regenerated. The port declarations are in the same order as the ports list. If you look carefully at the past several examples, you will notice that the port declarations are not always in the same order as the port list. There is no requirement in Verilog that the port declaration order match the port list order, but matching the orders improves readability.

WHEN TO USE UDPS

User-defined primitives (UDPs) are fast and efficient in Verilog. To model a flip-flop in Verilog, you might need to use four or more gates. But the same flip-flop might be modeled with a single user-defined primitive. An AND-OR-INVERT gate might require the use of three Verilog built-in primitives (two *ands* and a *nor*) but could be modeled with a single UDP. In general, Verilog simulators can evaluate a UDP as fast as they can a built-in primitive.

When designing a library for Verilog, as a general rule you should use a UDP if you can replace three or more gates. The only time you might not want to use a UDP is if there are many inputs. UDPs allow up to ten inputs. However, in practice, anything more than six inputs is difficult to write.

In a typical large circuit there are many instances of flip-flops. If the flip-flops are modeled efficiently with UDPs, the simulation performance will be noticeably improved.

The UDPs most commonly used are flip-flops, muxes, and AND-OR gates. In each of these common usages, the UDP replaces three or more primitive instances.

BLOCKING AND NON-BLOCKING ASSIGNMENTS

Chapter 4 introduced both the blocking and non-blocking procedural assignments, however no rules were presented explaining the best usage of each. When to use a blocking or non blocking assignment is a matter of style or personal preference. However this simple rule can avert modeling problems: For combinatorial logic use blocking assignments with no delays; for sequential logic use nonblocking assignments and unit delay.

Example 17-13 Blocking vs Non Blocking Assignments

```
module queuectl( . . . ); // this is an incomplete example
 . . .                    // details omitted for clarity
// Sequential logic
always @(posedge clock) begin
  if(myWrite) begin
    write <= #1 1;
    writeData <= #1 busData;
  end
  else
  begin
    write <= #1 0;
  end
end

always @(readPtr or writePtr)
  if(readPtr == writePtr)
    qEmpty = 1;
  else
    qEmpty = 0;

endmodule
```

Most of the examples in this book use blocking assignments for simplicity, Example 17-13 shows the preferred style of using zero delay blocking assignments for combinatorial logic and unit delay non-blocking assignments for sequential logic.

The advantage of this style for combinatorial logic is that the combinatorial logic is zero delay independent of the number of stages of logic. Since the actual depth and delay of the combinatorial logic is be determined by the synthesis tool, the delay in the source Verilog is irrelevant. An interesting side effect of this strategy is that a logic error resulting in a combinatorial feedback loop will be easy to detect as a zero delay oscillation in simulation.

The advantage of using the unit delay blocking assignment for sequential logic is two fold. It reduces the possibility of race conditions in simulation, and it makes it

easier to understand the results of simulation. Since synthesis ignores the delays, there is no harm in including them in the simulation.

If a simulated register has zero clock-to-q delay, it is possible that the data from this register can get to the next register before the clock. Although it is possible that the final implementation of a circuit will result in enough delay in the clock signal to mimic this condition of the data arriving at the next stage before the clock, rarely is data faster than the clock. Using a clock to q delay of one will eliminate these races in the verilog code and more closely model the final circuit.

Debugging a circuit is also simplified when the registers have clock-to-q delay. If you look at a result wave form from a simulation that had zero delay, it is difficult to know if a change that occurs with a clock was seen at that clock or caused by that clock. With a unit delay, it is quite easy to see that a signal changed as a result of the clock, and the value of that signal is seen on the edge of the clock.

18 TEST BENCHES AND TEST MANAGEMENT

INTRODUCTION TO TESTING

Once you have written your model, you are not even half done. Developing tests for your designs can take much more time and more code than the original models. Typical designs require three to ten times more effort for creating the tests than for developing the original designs.

You might well ask: "If it takes more time and more code to test a model, why is testing being introduced so late in the book?" The answer is that Verilog makes writing your tests easier by using the same techniques for writing tests as you use to write your models. In this chapter, you will use all the constructs that you have already learned to test models.

While you develop the test suite and behavior model for the alarm model in Chapter 16 or for any other complex system, you will undoubtedly find mismatches between the assumptions of behavior in the model and the assumptions of behavior in the test suite. The differences are resolved by fixing either the model or the test suite, until a single correct set of behaviors is found. The process of enhancing the model

and enhancing the test bench can be one of the most time-consuming parts of the design cycle.

Developing a good set of tests can make the difference between a design that behaves properly the first time, and a design that does not perform correctly under some conditions. While you develop your test suites, you can add *$monitor, $display,* and custom checking tasks to check for correct responses.

The code to apply tests often becomes repetitious. As with any Verilog module, if the same several lines of code are repeated many times, it becomes obvious that a *task* would be a more effective modeling technique. You may find that you use *tasks* more often in creating test benches than in the course of designing circuits.

Model Size versus Test Volume

The main purpose of modeling with Verilog is to determine if you are designing and building the correct circuit or system. When you model a design, you are trying to describe how your circuit should act. When you write tests, you are trying to simulate the environment in which the circuit will function, and to exercise all of the functionality of your circuit. Your tests should try to exercise every possible scenario that the circuit may encounter.

As mentioned earlier, writing the tests can take much more code and time than the model being tested. Consider the case of the 8-bit adder and its test bench described in chapter 3. All the code for the 8-bit adders requires only 33 lines of Verilog. The test bench, which applies only six tests and does not fully test the design, requires 48 lines. (The *test_adder* test bench is not the most efficient way to create a test bench; it was written to be easy to understand while you were first learning how to model in Verilog.)

The test bench for the *alu* model in Chapter 10, which is compact and uses an external data file, may need even more code than some implementations of the *alu.* When you consider the additional code that was written to generate that external data file (*alu_test.vec),* the volume of test code is much greater than the volume of code in the *alu* model.

For a simple, loadable counter example, it might take an expert about 15 minutes to model the counter, 30 minutes to create a test bench, and 15 minutes to debug the design. Most of the errors found during debugging will be in the test bench.

Some approaches to testing a new design involve modeling an entire system around it. For example, to test a model for a disk controller, the disk and a computer interacting with the disk controller might be modeled. In a case like this, there may

be a hundred times more code written to test the circuit than to model the circuit itself.

TYPES OF TESTS

Functional Testing

The most common types of tests written for a model are called *functional tests* because they test the expected functionality. These tests apply the expected inputs and look for correct responses. During the development of your model, your first tests will exercise what you expect to happen. Functional tests do not necessarily test every possible manufacturing fault, nor do they necessarily test every possible set of inputs.

Regression Testing

During the progress of a design, you should develop a set of tests with known responses. Each time you make a change to the design, you should rerun the tests and compare the results from the new model to the results from the old model. This rerunning of tests and matching the results to a known, good set of results is called regression testing. The purpose of regression testing is to verify that no errors are introduced into the circuit as the design evolves.

Sign-Off

When your design is ready for fabrication, you will need a set of tests that will be run against the design when it is fabricated. The purpose of these tests is different from that of the functional tests. The purpose of the sign-off tests is to detect any manufacturing faults. There are special simulators (called fault simulators) that can determine if your tests exercise and detect all possible manufacturing faults. It may be possible to exercise your circuit to catch all possible manufacturing faults with many less vectors than you used for functional testing.

Another concern when writing sign-off tests is the simulator(s) that your vendor will support for sign-off. Most ASIC vendors support Verilog-XL as a sign-off simulator. The simulators that vendors support are important because the vendors have verified that these simulators (with their libraries) produce the same results as the circuits they fabricate. You want the circuits which a vendor fabricates for you to behave the same as your simulations.

System Test versus Unit Tests

A *unit test* is a test that tests an individual module, or a small set of modules, that form a functional unit in your design. A *system test* is a test that tests a larger group of modules that forms a complete system. A large design project will have a combination of both system and unit tests.

In a large design, many of the problems are found in the interconnections between the parts of the system. The best way to find these is by using system tests. In system tests of a microcomputer system, software may be written and then run on the simulated system.

Unit tests should be written for every functional unit in a large design. It is much easier to debug a problem using a unit test. However, it might be easier to find a problem using system tests because system tests tend to exercise a circuit more fully, and run the circuit through many more cycles. Therefore, systems tests tend to catch more errors. When a problem is found with a system test, you can write a unit test to reproduce and debug the problem.

Typically, a large system design is partitioned into functional units. As you model each unit in Verilog, you should develop unit tests. As units become functional, they can be combined into subsystems, and you can develop subsystem tests. Next, all the modules can be combined into a system, often by adding further modules to build a complete system (including the operating environment). Systems tests are developed and run against the simulated system. The more tests you can run in simulation, the better the chance that the final design will be built without errors.

After all the, functional or behavioral models have been verified at the unit and system levels, each module can be refined either manually or through synthesis. Each of the refined modules can then be regression tested using the unit and system tests to ensure that no errors have been introduced during design evolution.

Both unit and system tests are used in creating a comprehensive test suite for your design. The techniques for developing unit and system tests are similar.

CREATING TEST PLANS

Before creating a Verilog model for a circuit a specification or plan should be created. Before creating a test module you should create a test plan. A test plan should look at all the functions of the model and list tests to be written.

A test plan for the adder could be:

- Test smallest possible sum
- Test carry in
- Test sums with and without carry in
- Test a sum that overflows

A test plan for the ALU exercise could simply be:

- Test all possible functions

Given a description for a loadable up - down counter with reset, similar to the examples in Chapter 10. A test plan for the counter could be:

- Test Reset
- Test Load
- Test Count up
- Test Count down

Test plans do not need to be long or detail each test. The test plan should at least outline the general tests you plan to write. Without a plan you will likely forget to test something. Please see Chapter 22 regarding code coverage for a correlation between test plans and code coverage.

THE BASIC TEST CYCLE
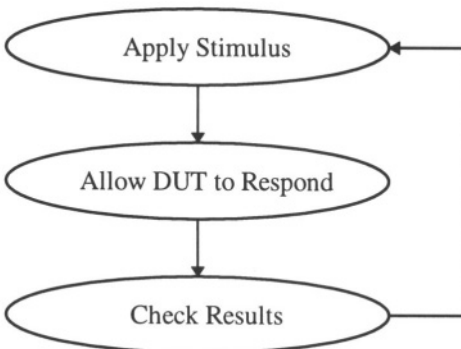
Figure 18-1 The Basic Test Cycle

Figure 18-1 shows the basic test cycle. Any test needs to follow this simple three step process. First apply stimulus. Second allow the device under test time to

respond. Third, observe the outputs and check the results. The cycle can then repeat for the next test.

Hardware Setup and Hold, and Response Time

Testbenches need to model the timing of the expected circuit. The test bench needs to provide inputs with adequate setup time and hold time. The test module must allow the model enough time to respond. The pre-synthesis model may be able to respond in near zero time, but the final implementation may need significant time to respond.

The Test Cycle for Combinatorial Models

The adder test module, first shown in the structural modeling chapter, is repeated in Example 18-2 below. This model uses a simple cycle for applying the stimulus, allowing the circuit to respond, and checking the results. Inputs are applied immediately at time zero. The test bench allows 100 time units for the adder to respond. The 100 time units would allow enough time for even a slow gate level adder to respond. The results are checked (by the if statement), and the cycle repeats. The inputs to a combinatorial model can be changed immediately after the results are checked. The applying of the stimulus and results checking are in the same *begin-end* block, so there is no race between when the results are checked and the new input is applied. If the inputs were applied from one *initial* or *always* block, and the results were being checked in a different *initial* or *always* block, the timing should allow the results to be checked before the stimulus is changed.

The Test Cycle for Sequential Models
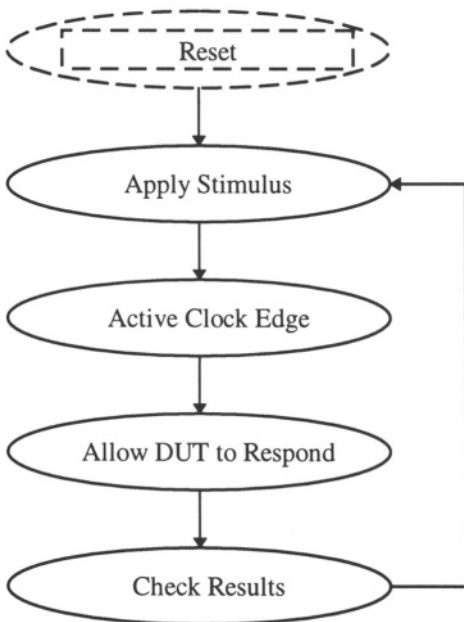
Figure 18-2 Test Cycle for Sequential Models

Figure 18-2 shows the test cycle for sequential models. Depending on the test and the model, the reset cycle may be optional. The reset itself may be one of the tests. In complicated models the reset may be a long sequence for example, initializing several devices on a bus.

The basic test cycle for the sequential device involves applying stimulus, waiting for the active edge of the clock, then checking the results, since a sequential device is likely to have setup and hold requirements. The test cycle timing should take all factors into account.
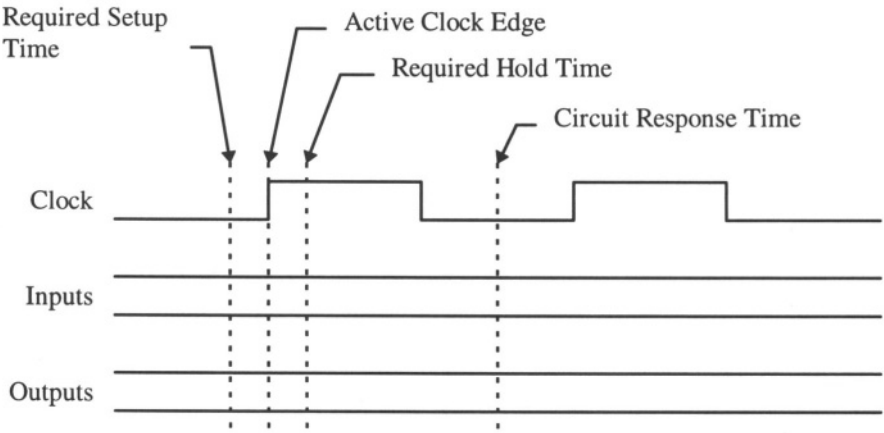
Figure 18-3 Sequential Test Cycle Timing

Figure 18-3 Shows a possible test cycle timing. It is quite possible that the setup, hold and response times are near zero for a pre-synthesis model, however after synthesis the model may require significant setup, hold, and response times. Considering all of these times may appear complicated, but it can be greatly simplified. The test cycle for the combinatorial circuit combined the time for checking the result and applying the next test. The same technique can be applied to the sequential test cycle. A single time can be located where the results can be checked and the next stimulus applied. Figure 18-4 shows an example of the simplified timing.
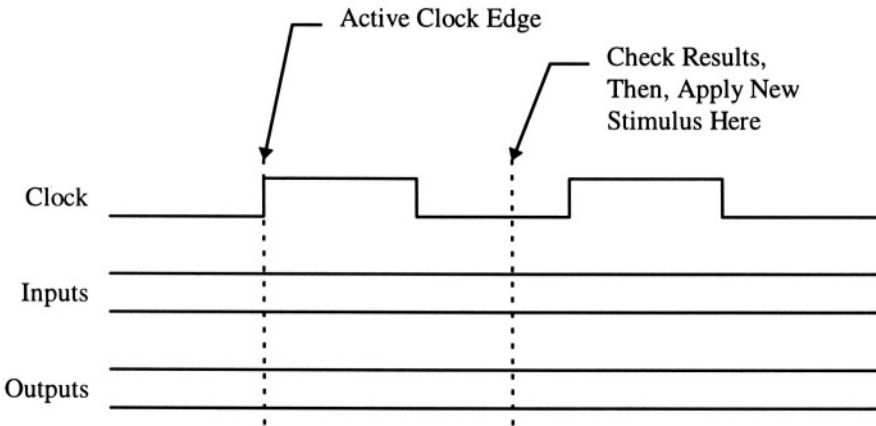
Figure 18-4 Simplified Sequential Test Cycle

As an example if the clock has a cycle time of 10, the circuit has setup and hold requirements of 1, and a response time of 7, Then a time of 8 units after the clock could be used for applying the stimulus and checking the results. Example 18-1 shows a skeleton for the test cycle just described.

Example 18-1 Basic Sequential Cycle Test Bench

```
reg clock, in1, in2;
wire out;

mymod dut(clock, in1, in2, out);

initial begin     // clock generator
  #5 clock = 0;
  #5 clock = 0;
end

inital begin  // stimulus and response checking
  @(posedge clock)
  #8 in1 = ... set up stimulus - first test
  @(posedge clock)
  #8 if ( ... check resuts - first test
  in1 = ... set up stimulus - second test
  @(posedge clock)
  #8 if( ... check resuts - second test
  in1 = ... set up stimulus - third test
  @(posedge clock)
  #8 if( ... check resuts - third test
```

In Example 18-1 the test sequence itself looks at the generated clock. This has several advantages. If the clock period is changed, the stimulus stays synchronized with the clock. The test does not need to calculate and anticipate when the active edges of the clock will occur. There was no need to calculate any complicated timing for the test bench. One improvement can be made here is to make the *#8* a text macro or parameter so timing can be maintained more easily if the clock period or circuit response time changes.

Similar to the combinatorial test cycle example; If the inputs were applied from one *initial* or *always* block, and the results were being checked in a different *initial* or *always* block, the timing should allow the results to be checked before the stimulus is changed. Please see Chapter 22 for an additional example.

SELF-CHECKING TEST BENCHES

The test bench for the 8-bit adder is a self-checking test bench: The test bench automatically checks the results from the adder and reports any errors. The test

bench for the *alu* model is also self-checking. There are many ways to build a self-checking test bench. The basic idea is to automatically check the response from the circuit and detect any differences from the expected results. Self-checking test benches are the easiest to use as regression tests because, after you run a self-checking test bench, it is easy to tell if the tests pass or fail.

How should you generate the correct answers for a self-checking test bench? In the test bench for the 8-bit adder, the correct answers were hard-coded into the test bench. This choice of modeling style was based on your presumed level of Verilog experience at the time that this test bench was introduced. That test bench could be improved in three ways. First, Verilog could be made to do the math and calculate the expected response. Second, the repeated code for each test could be simplified and put into a *task*. A third alternative for self-checking is to compare the response from two different circuits, typically a behavioral model and a gate-level model.

Example 18-2 Adder Test Module Repeated

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

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

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

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

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

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

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

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

  end
endmodule
```

Example 18-2 repeats the adder test module from Chapter 3. This was the first test bench introduced. It is an example of a simple, self-checking test bench, but what about some improvements? The test bench can be improved to calculate the correct responses, or the repeated code could be moved into a *task*. These and other possible modifications will be shown and discussed.

Example 18-3 Using Verilog To Calculate Responses

```
module test_adder_vc;
reg [7:0] a,b;
reg carry ;
wire [7:0] sum;

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

initial begin
    a = 0; b = 0; carry = 0;

    # 100 if (sum ! == a + b + carry)  begin
        $display("sum is wrong");
        $finish;
       end

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

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

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

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

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

  end
endmodule
```

Example 18-3 is a minor improvement on Example 18-2. Now Verilog calculates the correct responses, so the chance of a human-introduced error is minimized. However, this test bench still does not check the result on *carry_out,* and has a section of code that is repeated several times.

Example 18-4 Simplifying the Test Bench with a *task*

```
module test_adder_t;
reg [7:0] a,b;
reg carry ;
wire [7:0] sum;

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

initial begin
  test(0, 0, 0);
  test (1, 0, 0) ;
  test (0, 0, 1) ;
  test (5, 6, 1) ;
  test(200, 55, 1);
  test (18, 200, 1) ;
  $finish ;
end

task test;
  input [7:0] ax, bx;
  input cx;
  begin
```

```
    a = ax;  b = bx;  carry = cx;
    #100 if( {carry_out, sum}  !== ax + bx + cx) begin
      $display("result is wrong");
      $finish;
      end
    end
endtask

endmodule
```

Example 18-4 simplifies the test bench by using a *task* to replace the repeated
section of code. Because the checking code has been condensed into a *task*, it is also
easy to check *carry_out.* Notice the additional code in the *task* to calculate and
check *carry_out.*

Example 18-5 Using a Second Module To Check the Results

```
module test_adder_c;
reg [7:0] a,b;
reg carry ;
wire [7:0] sum, sum_check;

adder8 dut(carry_out, sum, a, b, carry);
nadder check_adder (carry_check, sum_check, a, b, carry);
defparam check_adder.size = 8;

initial begin
  test(0, 0, 0);
  test (1, 0, 0) ;
  test (0, 0, 1) ;
  test (5, 6, 1);
  test(200, 55, 1);
  test(18, 200, 1);
  $finish ;
end

task test;
  input [7:0] ax, bx;
  input cx;
  begin
    a = ax; b = bx;  carry = cx;
    #100 if( sum !== sum_check) begin
      $display("sum is wrong");
      $finish;
      end
    if( carry_out !== carry_check) begin
      $display("carry is wrong");
      $finish;
      end
```

```
     end
endtask

endmodule
```

In Example 18-5, an instance of a behavioral adder is used to check the results from the gate-level adder. This approach—running two models and checking the results—is useful to determine if you have built a model according to the specification in the behavioral model. The second adder instance *nadder* is an instance of the parameterized behavioral adder from Chapter 14. Each of the adders is connected to separate output wires and the *task test* compares the outputs. Another approach shown in Example 18-6 to comparing the results from modules in parallel is to connect the outputs to the same output wires and use the reduction exclusive OR to look for any *x*. If the two circuits are putting out the same result, the result will be both correct and known. If the circuits disagree on any output, the result will be an *x*.

Example 18-6 Generating x's for Miscompare

```
module test_bench;
reg clk;
parameter half_period = 5;
reg[7:0]  stimulus1,stimulus2; // input stimuli to models
wire[7:0] response1,response2; // results from models
wire error_flag = ^{response1,response2};

// instantiate models to be compared
behav_model dut1(response1,response2,stimulus1,stimulus2);
rtl_model dut2(response1,response2,stimulus1,stimulus2};

always begin
  clk = 0;
  #half_period clk = 1;

  #half_period if(error_flag===1'bx) begin
    $display("error at time ",$time);
    $displayb(response1,,response2);  //x(s)  mark bad bit(s)
    $finish;
   end
end
endmodule
```

RESPONSE-DRIVEN STIMULUS

Because Verilog uses the same language for modeling and stimulus, your stimulus can respond to your circuit. Stimulus can be simply written to apply different inputs

at different times. But what if the circuit is not ready for the input? You may need to rewrite your stimulus to get the correct timing. For example, assume you are designing a bus protocol controller that waits for a request, responds with a grant, and then expects data. You could write the stimulus to simply be time based, and guess how long it will take from the request to the grant, and guess when to apply the data. Instead, you could write a test that applies the request, then waits for the grant and then applies the data. Using the response-driven method, if the design parameters change and the time between the request and the grant changes, the test does not need to be rewritten.

Perhaps the simplest case for response-driven stimulus is a printer. The rate at which a printer can accept data is not constant. The rate depends upon the size of the printer's buffer, and whether the printer can keep printing as fast as the data are transferred to it.

Example 18-7 Printer Abstraction

```verilog
module printer(data, strobe, ack);
input [7:0] data;
input strobe;
output ack; reg ack;

parameter buffer_size =4; // size of the internal buffer
parameter print_time = 25;  // time to print a character
parameter buffer_write_time = 10;
time done; // the next time the buffer will be empty
initial begin ack = 0;  done = 0;  end
always @(posedge strobe)  begin
  if( done > $time + buffer_size * print_time) #print_time;
                                              // buffer full
  if(done > $time) done = done + print_time;
                             // calculate next empty time
     else done = $time + print_time;
  # buffer_write_time ack = 1;
  $write("%c", data); // print it
  # 1 ack = 0;
end
endmodule
```

Example 18-7 dynamically calculates the next time a printer can accept more data, based on the incoming data rate, the size of the buffer, and the print speed. This abstraction represents the model of a printer being able to accept data. If this printer were being designed by your design group, they could also modify the size of the buffer, the speed of the buffer, and the speed of the print mechanism.

If your job is to write a test bench to send the message *"this is a test message"*, how will you time the sending of the characters to the printer? One approach would be to send them slowly. Perhaps a delay of 100 between them would be conservative, but

that may not accomplish the goal of sending them as fast as the printer can accept them. Another approach might be to guess the timing in this fashion: send the first four characters (the size of the buffer) with a delay of 11 (the buffer write time plus the acknowledge time). With what timing would the fifth character be sent?

Example 18-8 Printer Test Bench with Guessed Timing

```
module print_test_1;
reg [7:0]  data;
reg strobe;

printer dut(data, strobe, ack) ;
 initial begin
      strobe = 0; data = "t";  #1 strobe = 1;
  #10  strobe = 0; data ="h";  #1 strobe = 1;
  #10  strobe = 0; data ="i";  #1 strobe = 1;
  #10  strobe = 0; data ="s";  #1 strobe = 1;
  #25  strobe = 0; data = " ";  #1 strobe = 1;
  #25  strobe = 0; data ="i";  #1 strobe = 1;
  #25  strobe = 0; data ="s";  #1 strobe = 1;
  #25  strobe = 0; data = " ";  #1 strobe = 1;
  #25  strobe = 0; data ="a";  #1 strobe = 1;
  #25  strobe = 0; data = " ";  #1 strobe = 1;
  #25  strobe = 0; data ="t";  #1 strobe = 1;
  #25  strobe = 0; data ="e";  #1 strobe = 1;
  #25  strobe = 0; data ="s";  #1 strobe = 1;
  #25  strobe = 0; data ="t";  #1 strobe = 1;
  #25  strobe = 0; data = " ";  #1 strobe = 1;
  #25  strobe = 0; data ="m";  #1 strobe = 1;
  #25  strobe = 0; data ="e";  #1 strobe = 1;
  #25  strobe = 0; data ="s";  #1 strobe = 1;
  #25  strobe = 0; data ="s";  #1 strobe = 1;
  #25  strobe = 0; data ="a";  #1 strobe = 1;
  #25  strobe = 0; data ="g";  #1 strobe = 1;
  #25  strobe = 0; data ="e";  #1 strobe = 1;
  #25  strobe = 0; data ="\n"; #1 strobe = 1;
  #25  $finish;
end

endmodule
```

In the test module in Example 18-8, the test bench must be modified to send the data at the correct rate each time the design team makes a change in the printer design. This means you will have to spend time modifying the timing in the test bench, rather than designing a better printer or test bench.

With the guessed timing in Example 18-8, the test bench and printer modules appear to work, the test message is printed properly, and the finish time is 553. Is this the optimal time? The test bench shown in Example 18-9 also prints the

complete message correctly, but finishes at simulation time 478, so the guessed timing was incorrect.

Example 18-9 Response-Driven Printer Test Bench

```
module print_test_2;
reg [1:8] data;
reg strobe;
parameter message_length = 23;
parameter [1 : 8*message_length] message
                                = "this is a test message\n";
integer i,j;

printer dut(data, strobe, ack);
initial begin
     strobe = 0;
  for (i=0; i<message_length; i=i+1) begin
    for(j=1; j<=8; j=j+1) data[j] = message[j+i*8];
    #1 strobe = 1;
    @(posedge ack) strobe = 0 ; // wait for the response
  end // for
  $finish;
end

endmodule
```

In Example 18-9, all the timing is removed from the test bench. Instead, the test bench gets its timing from the device under test. Test benches driven by the circuit's response can be used to simplify the timing, as shown here, or the test bench can apply a different set of inputs depending on the output from a circuit.

TEST BENCHES FOR *INOUTS*

Test benches for *inouts* can be a daunting problem. The key to writing a test bench for an *inout* is to recognize that there are two parts of the test bench for each *inout* port: There is a wire connected to the port and a register for driving the input. The state of the port is observed on the wire. Values to be driven into the *inout* port are placed in the register. The register is connected to the port with a continuous assignment. Figure 18-5 graphically depicts the register, the wire, and the continuous assignment required when providing a test bench for an *inout.* When you need to observe the value being driven out from the module under test, the register must either not be driving or else be disconnected. The simplest way to make a register not drive is to place a '*bz* in it. The continuous assignment that connects the register to the port could also be used to disconnect the register from the port by using the conditional operator (? :).
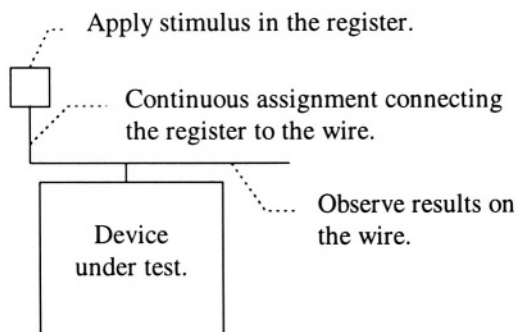
Figure 18-5 Test Bench for an *inout*

The most important thing to remember about applying stimulus to an *inout* is that you need both a register and a wire. Connect the wire to the port and the register to the wire with a continuous assignment.

Example 18-10 Test Bench for a RAM

```
module test_inout;
reg [7:0] address;          // drive address into the ram
reg [15:0] data_reg;        // drive data into the ram
wire [15:0] data;           // actual wire connected to the ram
reg read, write;            // control signals, active high

assign data = data_reg; // connect the register to the wire

ram dut(data, address, read, write); // device under test

initial begin
  data_reg = 16'bz; read = 0; write = 0 ;  // set all off
  address = 0 ;                             // test location 0;
  #100 if(data !== 16'bz )
    $display( "Data port did not turn off.");
  data_reg = 16'hfeed; write = 1; // drive data in
  #100 write = 0 ; data_reg = 16'bz;
  #100 if(data !== 16'bz )          // make sure it turns off
    $display( "Data port did not turn off.");
  read = 1;                         // read the data back out
  #100 if(data !== 16'hfeed )
    $display( "Data is wrong.");
  $finish;
end
endmodule
```

Example 18-10 is a simple test bench for the RAM model shown in Chapter 7. Only one location in the RAM is exercised. The wire *data* is connected to the *inout* port

on the RAM. The register *data_reg* is used to drive in an input value. *16'bz* is placed in *data_reg* when the test bench is not driving the *inout* port.

LOADING FILES INTO VERILOG MEMORIES

A Verilog memory might be used as a RAM or ROM to hold a program, or might simply be a holding place for test vectors to be applied. As a refresher, Example 18-11 shows a simple memory declaration. The width of the words in the memory is 8 bits, with bit 7 being the most significant bit. There are 256 words in the memory, with the first word being 0. By default, the memory is initialized to all *x*'s.

Example 18-11 Memory Declaration

```
reg [7:0] ROM  [0 :255] ;
```

There are two system tasks for reading a file into the memory, *$readmemh* (hexadecimal format) and $*readmemb* (binary format). Each of the tasks takes between two and four arguments. The last two arguments are optional and are usually omitted. The first argument is the name of the file to read in, and the second is the name of the memory to be loaded. You can optionally specify the starting and ending addresses as the third and fourth arguments.

The file is read into the memory in this order: The first line of the file is read into the first word of the memory. This may seem trivial, but the first word of the memory is the leftmost index, not the index numbered 0. Thus, reading the same file into the memories in Example 18-11 and Example 18-12 would yield different results. In Example 18-12, the first line of the file would read into location 255 in the memory so declared.

Example 18-12 Reversed Memory Declaration

```
reg [7:0] ROMB [255:0];
```

The file to be read in can be created with any text editor, or created from Verilog using a *$fdisplay* or *$fstrobe* command. The format of the file is simply numbers in hexadecimal format for *$readmemh,* or binary for *$readmemb,* separated by white space. The files may contain comments and the numbers may be uppercase or lowercase. The digits may be separated by underscores for clarity.

Example 18-13 Memory File *adder8.vec*

```
// aaaaaaaa bbbbbbbb carry in
   00000000_00000000_0
   00000001_00000000_0
   00000000_00000000_1
   00000101_00000110_1
   11001000_00110111_1
   00010010_11001000_1
```

The contents of Example 18-13 could be placed into a file *adder8.vec*. Example 18-14 shows the adder test bench modified to read the patterns in from the file. The test bench applies the patterns using a continuous assignment, uses a *for* loop to increment the index into the memory, and checks the results using a *task*. The *for* loop is hard-coded to loop through the first six locations (0 to 5), so this needs to match the file.

Example 18-14 Adder Test Bench Reading from a File

```
module test_adder_f;
wire [7:0] a, b, sum;
wire carry;
reg [8+8+1:1] stim[0:10];
integer index;
adder8 dut(carry_out, sum, a,b,carry);

assign {a,b,carry} = stim[index]; // apply stimulus from
                                   // the memory
initial begin
  $readmemb("adder8.vec", stim);
  for(index=0; index<6; index=index+1)
    test;
  $finish ;
end

task test;
  begin
    #100 if( {carry_out, sum} !== a + b + carry) begin
      $display("result is wrong");
      $finish;
      end
    end
endtask

endmodule
```

Files can also be in hexadecimal format and can include addresses. Unfortunately, the Verilog format does not match one of the industry standards you may know. In Verilog, the address is simply a hexadecimal number preceded by the @ symbol.

The data shown in Example 18-15 could be read into the PROM shown in Example 18-16. The last two lines in the file are read into the last 32 locations in the PROM. The comments at the end of the lines are for clarity only. The locations *a0* through *df* are not specified, and remain unknown or unchanged after loading this file.

Example 18-15 PROM Data File *prom.dat*

```
/* prom.dat data file */
54 68 69 73 20 77 61    73 20 77 72 69 74 74 65 // 00 - 0f
6e 20 62 79 20 4a 61    6d 65 73 20 4c 65 65 20 // 10 - 1f
61 73 20 61 6e 20 65    78 61 6d 70 6c 65 20 66 // 20 - 2f
69 6c 65 0a 31 32 33    34 35 36 20 69 66 20 79 // 30 - 3f
6f 75 20 61 72 65 20    72 65 61 64 69 6e 67 20 // 40 - 4f
74 68 69 73 0a 37 38    39 31 30 20 59 6f 75 20 // 50 - 5f
68 61 76 65 20 74 6f    6f 20 6d 75 63 68 20 74 // 60 - 6f
69 6d 65 20 6f 6e 20    79 6f 75 72 20 68 61 6e // 70 - 7f
64 73 21 0a 2a 2a 43    51 20 64 65 20 4e 31 44 // 80 - 8f
44 4b 2a 2a 00 00 00    00 00 00 00 00 00 00 00 // 90 - 9f
@e0
4e 6f 74 68 69 6e 67    20 6d 6f 72 65 20 74 6f // e0 - ef
20 73 61 79 00 00 00    00 00 00 00 00 00 00 00 // f0 - ff
```

Example 18-16 Simple PROM

```
module sprom(address, data);
input [7:0] address;
output [7:0] data;
reg [7:0] rom[0:255];

assign #45 data = rom[address]; // entire functionality
initial $readmemh("prom.dat", rom);
endmodule
```

The *$readmemh* or *$readmemb* task invocations do not need to be in the module that declares a memory array. The name of the memory being loaded may be a hierarchical name; for example, you could load all of your memories from the test bench. You do not need to load the memories at time 0 from an *initial* statement; they can be loaded at any time during the simulation. If the file you are trying to load cannot be read, Verilog issues a warning, and simulation continues without the file. Make sure that the files exist or you may be debugging a simulation with incorrect results because you overlooked the warning.

If you want to check that the correct number of locations is read, you can use the start and end address optional arguments to the *$readmemh* or *$readmemb* system tasks. If Verilog does not read the correct number of values from the file, a warning message will be printed. The start and end addresses are also useful to reverse the

direction of loading if you declared the range of words backwards or wish to load a range backwards.

TEST BENCHES WITH NO TEST VECTORS

The simplest test bench contains an instantiation of a module to be tested; declarations of registers for the inputs; and wires for the outputs. Example 18-17 shows a test bench with no vectors. Why would you want to do this? This allows the test vectors to be applied, and results to be checked, interactively.

Example 18-17 Test Bench with No Vectors

```
module test_adder_nv;
reg [7:0] a,b;
reg carry ;
wire [7:0] sum;

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

endmodule
```

Using the test bench in Example 18-17, you could enter Verilog interactively and manually specify values for *a, b,* and *carry,* and then observe the results. This type of test bench gets you into simulation as rapidly as possible and lets you experiment with your models. Chapter 21 (on debugging) describes how to use Verilog interactively.

USING A SCRIPT TO RUN TEST CASES

You can set values of registers, load files into memories, or trigger tasks you coded into your modules specifically for debugging—all interactively. If you designed a microprocessor system, the main stimulus is most likely the program loaded into memory, and you may have coded a task to reset the processor. A sequence of tests might be to load a program, reset the processor, let it run, load the next program, and so on. This sequence of loading programs may be repeated for several test programs. You can write a script of interactive commands to run a set of test cases. Interactive commands are read into Verilog using the *-i* command line argument, or interactively using *$input.* An interactive command file can even trigger the loading of another interactive command file.

The advantage of using a command script is you don't have to code all of your tests into a test bench in advance. As your tests develop or as you debug interactively,

you can begin to build a set of scripts for your test cases, rather than modifying test benches or test vector files.

MODELING BIST

Built-in self-test (BIST) is a technique for applying stimulus and collecting responses within a circuit. BIST requires the modeling of a linear feedback shift register (LFSR) to generate input patterns and a multiple-input shift register (MISR) to collect the responses. The LFSR uses a polynomial to generate a known sequence of pseudo-random inputs. The MISR uses another polynomial to generate a unique signature for the tests. The art of using the optimum polynomials is not discussed here, and is the subject of other texts. When the circuit is in test mode, the inputs are switched from their normal function to be connected to the LFSR, and the outputs are connected to the MISR. The LFSR is initialized to a known value and a fixed number of clocks is applied. At the end of the clocking sequence, the value in the MISR is compared to the correct signature and the pass/fail status is determined. The entire logic added for BIST includes muxes for the inputs; the LFSR and MISR; and a simple state machine to control the test sequence. The added BIST circuitry indicates pass or fail status either through a single pin or in some status register.
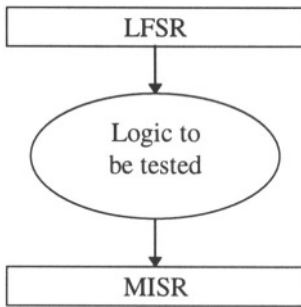
Figure 18-6 Logic Surrounded by BIST

Figure 18-6 depicts logic surrounded by BIST. As you can see, there are two main elements, the LFSR and MISR. The LFSR needs to be designed to apply a unique set of test vectors that stimulate the circuit in a meaningful way in the smallest number of clock cycles. The LFSR is driven by a polynomial depicting which bits are fed back into the shift register. Finding the correct polynomial is matter of research, trial and error, or simulation.

Example 18-18 LFSR

```
module test_lfsr;
reg [31:0] lfsr;
reg bit;
reg [32:0]  count;

initial begin
  lfsr = 1;
  count = 0;
  forever begin
    bit =lfsr[30]^lfsr[6]^-lfsr[4]^lfsr[1]^1fsr[0];
    1fsr = {bit,lfsr[31:l]};
    count = count + 1;

    if(lfsr == 0) begin
      $display("lfsr died at zero after %d cycles",  count);
      $finish;
    end
    if(lfsr == 1) begin
      $display("returned to 1 after %d cycles", count);
      $finish;
    end

    if (count[32]) begin
      $display("I must have repeated somewhere...");
      $finish;
    end
  end // forever
end  // initial
endmodule
```

Example 18-18 illustrates a module to test the polynomial for a LFSR. In this case, the LFSR uses as feedback the exclusive-OR of bits 30, 6, 4, 1, and 0. The test will report if the LFSR dies (goes to 0, where it will get stuck), gets stuck in a loop, or returns to its initial value. It is possible that the LFSR will eventually find a pattern that continually repeats. However, in this case the LFSR will return to its initial value after cycling through every other non-0 value. This module will take a long time to run because it runs through more than a million combinations.

Once you have determined the size and polynomial for the LFSR, you need to design the MISR. The size of the MISR determines how likely it will be to get a false positive result. Example 18-19 shows how to test the ALU with a simple LFSR and MISR. The BIST circuitry is abstracted in the model. A more complete model would have a pin to start the test; a clock or test-clock input; a state machine to run the tests; and an extra output pin or two indicating when the test is done and the status of the test.

Example 18-19 Testing the ALU with a LFSR and MISR

```
module  alu_lfsr;
reg [35:0]  lfsr;  // 16 + 16 + 4 -1
reg bit;
reg [31:0]  misr;
wire [15:0] a,b, aluout;
wire [3:0] f;
wire zero,parity,carry;
parameter final_signature = 32'ha592_2a45;
parameter test_cycles = 1023;

initial begin
  lfsr = 1;
  misr = 0;
  repeat (test_cycles) begin
    bit = lfsr[30]^lfsr[6]^lfsr[4]^lfsr[1]^lfsr[0] ;
    lfsr = {bit,lfsr[35:1]};  // applies stimulus
    #100  // wait for result, modify misr
    misr = {misr[0],misr[31:1]} ^ {aluout,zero,parity,carry};
  end
  if(misr !== final_signature)begin
    $display("Signature mismatch: expected %h, got %h",
        final_signature,misr);
  end else begin
    $display("tested OK");
  end
end

assign {f,a,b} = lfsr ;  // apply stimulus from lfsr

alu  dut(a,b,f,aluout,zero,parity,carry);
endmodule
```

THE SURROUND AND CAPTURE METHOD

The surround and capture method is used for generating unit tests out of system tests, or for testing a submodule that is normally only tested in a larger test case. As the name suggests, the module to be tested is surrounded, and its inputs and outputs are captured to a file. The captured values can then be played back in a new test bench against different versions of the module and checked for equivalence. Figure 18-7 shows a system in which a submodule's inputs and outputs are being captured.
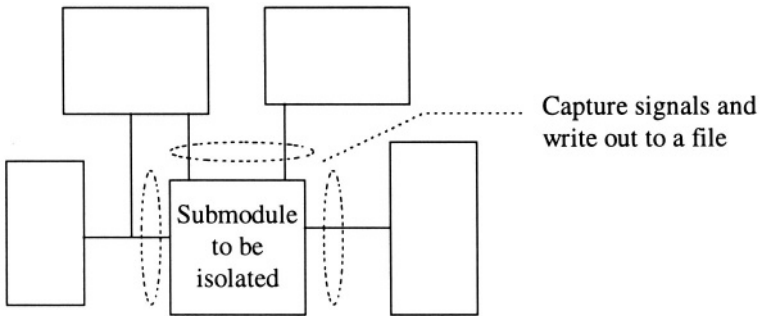
Figure 18-7 Surround and Capture Method

The system tasks *$fdisplay* or *$fstrobe* are used to write the values to a file. In which module should you put the tasks to capture the inputs and outputs? The easiest place to capture the inputs and outputs of a submodule is in the submodule itself. In the submodule, you can easily see all the inputs and outputs, and you also know when the inputs and outputs are valid. The second easiest place to capture the inputs and outputs is from the module that instantiated the submodule. Once you have decided where to do the capture, you need to know when to capture the values. If there is one time in your circuit when the inputs are stable and the outputs have reached their output for the current input, that is the best time to capture. Typically the time when both the input and outputs are stable is just before the inputs are going to change again. For example, if your system has a clock with a cycle time of 100, and each rising clock causes the next set of changes, the best time to capture the inputs and outputs might be to wait for the clock to rise, then wait 90 time units more, then use *$fdisplay* to send the values to a file.

Example 18-20 ALU Modified Capture of Inputs and Outputs

```
module alu (a,b,f,aluout,zero,parity,carry) ;
     input [3:0]  f;
     input [15:0] a;
     input [15:0] b;
     output [15:0] aluout;
     output zero;
     output parity;
     output carry;

/* Code inserted into alu to write vectors. */
integer fo;
initial begin
    fo = $fopen("alu_test.vec");
    forever @(f) #10 if(1'bx === ^a)
         $fdisplayh(fo, {16'bl,b,f,aluout,zero,parity,carry});
       else
         $fdisplayh(fo,  {a,b,f,aluout,zero,parity,carry});
 end
/* end write vectors */
 // functionality of ALU omitted for clarity
endmodule
```

Example 18-20 shows how an ALU in a RISC processor was modified to capture the ALU's inputs and outputs. The result of this capture was the file *alu_test.vec* that you used in chapter 6. In this case, the inputs and outputs are determined to be stable 10 time units after the *f* input changes. Notice that the code also checks for some cases when the *a* input is unknown; in such a case, the capture code merely writes out known values. Replaying a vector file containing unknowns may not always work, so it may be necessary to ensure that only known values are written out.

Once the vectors have been captured to a file, you need to create a new test bench that will play the vectors back into the module. The module that plays the vectors back should apply the inputs, allow time for the outputs to settle, and then check the outputs. The test bench for the ALU used in chapter 6 is an example of a module that plays back captured vectors.

Example 18-21 ALU Test Bench Repeated

```
module test_alu;
reg [16+16+4+16+3:1] stim_res[1:100]; /*stimulus, response */
wire [15:0] aluout;  /* declare wires for results */
wire zero,parity,carry;
integer pattern ;

reg [15:0] a,b ; /* declare registers for stimulus */
```

```verilog
reg [3:0]  f;

reg [15:0] aluout_comp; /* declare registers to compare to */
reg zero_comp, parity_comp, carry_comp;

/* instantiate device under test */

alu dut(a,b,f,aluout,zero,parity,carry);

initial begin
    pattern = 1 ;
    $readmemh("alu_test.vec", stim_res);
    forever run_test;
    end

task run_test;
begin
  {a,b,f,aluout_comp,zero_comp,parity_comp,carry_comp} =
stim_res[pattern];
  if (carry_comp === 1'bx) $finish;
  #100

  if (aluout !== aluout_comp) begin
    $display(
"compare error on aluout, pattern %d was %b should be %b.",
        pattern, aluout,aluout_comp);
    $finish;
  end

  if (zero !== zero_comp) begin
    $display(
"compare error on zero, pattern %d was %b should be %b.",
        pattern, zero,zero_comp);
    $finish;
  end

  if (parity !== parity_comp) begin
    $display(
"compare error on parity, pattern %d was %b should be %b.",
        pattern, parity,parity_comp);
    $finish;
  end

  if (carry !== carry_comp) begin
    $display(
"compare error on carry, pattern %d was %b should be %b.",
        pattern, carry,carry_comp);
    $finish;
  end

  pattern = pattern + 1;
end
endtask
endmodule
```

Once you have captured the vectors, and have created the new test module, it is important to test the vectors and new test module against the module with which you captured the vectors. You may have captured the vectors at the wrong times or applied them with the wrong timing, so double-check them against the original submodule before using them to test other versions of the submodule.

The resulting test module and test vectors form a unit test for the submodule, and these serve as a quick verification of correct implementation. This test bench is satisfactory as a check-off (or regression) test. Unfortunately, because the tests embodied in the test vectors may be difficult to understand, it may be difficult for you to isolate the cause of any discrepancies.

This chapter merely introduces basic aspects of testing. To further automate testing in Verilog, and to improve your testing technique, you should be aware that there are many additional built-in Verilog commands and external programs you can use for testing.

For example, you may have noticed that *$readmemh* was used to read the file of test vectors. This is the simplest way to read files of stimulus into Verilog. If you have a large number of signals or a large number of test vectors, you will have to allocate a large amount of memory in Verilog to hold the test vectors. This might not be practical for a large circuit. Verilog has some system tasks that are more efficient in handling the reading and writing of files for test vectors. These routines are *$incpattern_write* (to write out a file of test vectors); *$incpattern_read* (to read in the file and apply the stimulus); and *$compare* or *$strobe_compare* (for comparing the response from the circuit to the expected values).

This Page Intentionally Left Blank

19 MODEL ORGANIZATION

Throughout this book modeling concepts have been presented without particular attention to organization of the models.

FILE ORGANIZATION

You may have noticed that the examples on CDROM with this book are in files where the file name matches the module name. It is standard practice to have only one module in a file and to give the file the same name as the module, with .v appended.

Verilog simulators allow you to supply a list of files to simulate. As you build more complex circuits, the file list is a way to organize your design. For example the 8 bit adder design should be in 5 files as shown in Example 19-1. For many tools you can create a text file with the list of files. This list is often suffixed with *.vc* (verilog commands) or *.f* (file). Many simulators accept this list of files by using a *-f* option, as in Example 19-2. The Silos simulator included with this book includes a project manager where you specify the files in your project.

Example 19-1 File List of 8 bit Adder *adder.vc or adder.f*

```
adder1.v
adder2.v
adder4.v
adder8.v
test_adder.v
```

Example 19-2 Using the file list

```
verilog -f adder.vc
```

Simply listing the design files is not the only organizational trick you can use. Verilog includes an `*include* compile directive. The `*include* in verilog is similar to the #*include* in 'C'. It allows another file to be part of the current file. It is typically used for common declarations such as REG_DELAY used in many of the examples in this book. Generally, included files have the extension *.vh* (verilog header) or *.v* (verilog). Example 19-3 shows a simple counter that includes two files. The *timing.vh* file in Example 19-4 is included outside the module since the `timescale directive must be outside a module. The *system.vh* file in Example 19-5 defines common information for the system. It could be included inside or outside the module.

Example 19-3 Counter Using `include

```
`include "timing.vh"
module counter1(count, reset, clk) ;
`include "system.vh"
output [`BUSWIDTH-1:0] count;
input reset, clk;
reg    [`BUSWIDTH-1:0]    count;
always @(posedge clk or posedge reset)
  if(reset)
    count <= #(`REG_DELAY) `BUSWIDTH 'b0;
  else
    count <= #(`REG_DELAY) count + `BUSWIDTH 'b1;
endmodule
```

Example 19-4 Timing.vh

```
`timescale Ins/Ins
`define REG_DELAY 1
// `define REG_DELAY 0
// `define REG_DELAY $random %3
```

Example 19-5 System.vh

```
`define BUSWIDTH 8
```

The `include` can include a full file path name. Standard practice suggests that `include` should only have the file name and not the path. This allows the files to be more portable and moved around your system or to another system. The verilog simulator allows you to specify a set of directories to search for the included files. For many simulators the command line option is *+incdir*.

DECLARATION ORGANIZATION

The port declarations of all the modules in this book have been simple and undocumented. A better practice is to declare 1 port per line and comment the use of the port. Example 19-6 shows an improvement of the port declarations. The ports are now one per line and commented. The output port and its *reg* re-declaration are combined into a single line. This improves the readability of this module.

Example 19-6 Counter with commented ports

```
`include "timing.vh"
module  counter2(
  count,    // output, width is `BUSWIDTH
  reset,    // active high asynchronous reset
  clk       // rising edge clock
  );
`include "system.vh"
output [`BUSWIDTH-1:0] count;  reg [`BUSWIDTH-1:0] count;
input reset;
input clk;

always @(posedge clk or posedge reset)
  if (reset)
    count <= #(`REG_DELAY) `BUSWIDTH 'b0;
  else
    count <= #(`REG_DELAY) count + `BUSWIDTH 'b1;
endmodule
```

ANSI Style ports

You may have noticed that the output *count* is repeated three times. Once in the port list, a second time when the direction is declared and a third time for the *reg* declaration. The input puts are declared twice , and could have been declared three times if we had re-declared the inputs as wires (which they are by default).

The IEEE 1364-2001 standard now defines ANSI style port declarations. If you are familiar with ANSI C, you will see the benefit of using the ANSI style declaration to eliminate some of this redundancy. Check your simulator and synthesis tools before moving to the new style. Example 19-7 shows the counter with ANSI style ports.

Example 19-7 Counter with commented ports

```
`include  "timing.vh"
`include  "system.vh"
module  counter3(
output reg [`BUSWIDTH-1: 0] count, // output
input wire   reset,     // active high asynchronous reset
input wire   clk        // rising edge clock
  );
always @(posedge clk or posedge reset)
  if(reset)
    count <= #(`REG_DELAY) `BUSWIDTH 'b0;
  else
    count <= #(`REG_DELAY) count + `BUSWIDTH 'b1;
endmodule
```

TESTCASE ORGANIZATION

When running simulations for a large system there will be many test cases. This does not always imply that there are many testbenches. There many be many more test cases than testbenches. A simple example of many test cases with a single test bench is system with an embedded processor. The test bench may be exactly the same, but a different program may be run on the processor. A system regression script may run the simulator again and again with either a different file name passed to the simulator, or with a new program file copied or linked to the file the simulated memory always reads.

Often, it is a good idea for each test case to have its own directory. This way the test can have unique input and output files, and the results remain in the test directory. The user or system test script would then change into the directory where the desired test is located and then run.

Including Test Cases

If each test case has a unique directory we can make use of two powerful techniques to simplify creating a set of tests for a complex system. Imagine a complex system with many devices on a bus, and one bus master is responsible for initializing the bus, then conducting the tests. The system reset and initialization sequence is

common, but each test is conducted by a unique set of write and read commands issued on the bus. Each unique test sequence will be in a file called *current_test.v* and in its own directory. The test bench has the reset sequence and then includes the *current_test.v* file.

Example 19-8 System Test Bench

```
module system_testbench;
  ...
event start_tests;
reg passed;

initial begin
  reset = 1;
  passed = 1;
  repeat (6) @(posedge clk) ; // need 6 clocks of reset
  wait (ready) // system responds with ready after internal
reset finishes
  write(`BOARD1, `BOARD1_CONFIG); // initialize system boards
  write(`BOARD2, `BOARD2_CONFIG);
  write(`BOARD3, `BOARD3_CONFIG);
  read_until(`BOARD4, `BOARD4_STATUS, `READY) ;
  -> start tests;
end

task done;
begin
  if(passed)
    $display("Passed");
  else
    $display("ERROR:  Test Failure.");
  repeat (3) @(posedge clk); // 3 clocks for bufferes to
flush
  $finish;
end
endtask

`include "current_test.v"
endmodule
```

There are many interesting things in this test bench. The event *start_tests* is used to signal the test sequence(s) to start once the system is initialized. The system initialization sequence is abstracted via text macros. A global variable *passed* is declared and initialized to true. If a test fails *passed* is set to zero. There is a task *done* which is called from the current test upon success or failure. The task also runs some additional time to allow the system to complete what was doing. Often in complex systems, a test may pass or fail, but the next few cycles are interesting.

The *write, read,* and *read_until,* tasks are assumed to be declared in the test bench, and cycle through the bus protocol for this system. Their details are not important.

Example 19-9 Current_test.v

```
integer i, j;
reg  [`BUSWIDTH-1  :0] data1, data2;
initial begin
  @ start_tests
  for(i=0; i< `MAX_BOARD1; i=i+l)
    write(`BOARD1+i, i); // write address = data pattern;
  for(i=0; i< `MAX_BOARD1; i=i+l)
    begin
      read(`BOARD1+i,data1);
      if(data1 !== i)
        begin
          passed = 0;
          done ;
        end
    end
  end

initial begin
  @ start_tests
  for(j=0; j< `MAX_BOARD2; j=j+l)
    write(`BOARD2+j, j); // write address = data pattern;
  for(j=0; j< `MAX_BOARD2; j=j+l)
    begin
      read(`BOARD2+j, data2);
      if(data2 !== j)
        begin
          passed = 0;
          done ;
        end
    end
  done ;
  end
```

This *current_test.v* does some interesting things. Since it is included in the module but not in a *begin-end* block, it can declare some additional data it needs, *i, j, data1,* and *data2.* This test also has two initial blocks that wait for the *start_tests* event before they begin testing. In this example a data equals address test in run by first writing data to two boards in parallel, then reading them back in parallel. The test competes by calling *done* on the first failure or when all is complete. Note, it is assumed that the *write* and *read* tasks (not defined) can read and write two boards at the same time. It is also assumed that the second board finishes after the first.

Remember the exact details of this test are not important, the important concept to learn is that the test can be broken between the test bench and the test sequence, and we can have many test sequences run by having a unique *current_test.v* in many test case directories.

Conditionally Running Rests

Often breaking tests up based on a design configuration or desired test mode simplifies testing. For example it might be desirable to run a simple test rather than a full test. Conditional compilation may be used to modify the test sequence.

Example 19-10 Conditional Test

```
initial begin
  @ start_tests
`ifdef SIMPLETEST
    write('BOARD2+2, 32'h1234_5678);
    read(`BOARD2+2, data2);
    if(data2 !== 32'h1234_5678) passed = 0;
`else
  for(j=0;  j<  `MAX_BOARD2;  j=j+l)
    write(`BOARD2+j, j); // write address = data pattern;
  for(j=0;  j<  `MAX_BOARD2;  j=j+l)
    begin
      read(`BOARD2+j, data2);
      if(data2 !== j)
        begin
          passed = 0;
          done;
        end
`endif
  done;
  end
```

The `ifdef, `else and `endif can be used to conditionally compile different sections of code. In Example 19-10, if *SIMPLETEST* is defined a single write-read is done, then it proceeds to *done* after the `endif, otherwise it runs the whole test.

MODEL REUSE

Similar models often can be constructed using a single module, where parameters may not work well. Example 19-11 shows an adder that can have two or three inputs.

Example 19-11 Adder with two or three inputs

```
module add23(
  a,
  b,
`ifdef ADD3
  c,
```

```
`endif
  sum
 );
output [7:0] sum;
input [7:0] a, b;
`ifdef ADD3
input [7:0] c;
`endif

assign sum =
`ifdef ADD3
             c +
`endif
             a + b ;
endmodule
```

Example 19-11 is a simple example of how conditional compilation can make models configurable. More common examples may be slight differences in a design depending on a FPGA or ASIC implementation.

SUMMARY OF MODEL ORGANIZATION COMPILE DIRECTIVES

The `ifdef can be used for conditional compilation of models. The definitions checked may be set with the `*define* construct or with a command line option +*define*+.

Compile Directive	Description
` include	Includes another file, avoid using path names.
`ifdef	Starts conditional compilation.
`else	Alternate conditional compilation.
`endif	Ends conditional compilation.
`ifndef	Starts conditional compilation. New in 2001 standard.
`elsif	Alternate conditional compilation. New in 2001 standard.

Pre-defined Text Macros

Most tools define unique text macros to allow conditional compilation of tool specific code. For example Verilog-XL predefines verilog, and Silos predefines silos.

20 COMMON ERRORS

Before moving on to Chapter 21, which covers debugging, it may save some time to look at some of the common modeling errors and how to correct them.

MISMATCHED PORTS

One of the first things to double-check in a simulation is the port connections. Mismatched ports are extremely common. They are most common in designs developed by more than one person. Port mismatches are differences between a module's declaration and its connections when instantiated.

There are three types of mismatched port problems:

- incorrect number of ports
- incorrect size of ports
- incorrect order of ports

Verilog warns you if there are too many or too few port connections. Verilog also warns you if the port sizes between the declaration and instantiation are mismatched. There is no single clear warning if the order of ports is mismatched.

Mismatched port order is the most common of the port connection errors. Again, there may be no warning at all if the incorrect hookup results in a legal circuit. Mismatched port order may also result in any number of Verilog error or warning messages. If you switch the order of two signals that are of different sizes, you might get a port size mismatch warning message. If you connect an output port of an instaniated module to a reg in the current module, you may get an error message about an illegal declaration.

The best way to avoid mismatched ports is to adopt some simple rules for port order. For simple modules with one output (such as a mux), use the same rules as those that apply to the Verilog built-in primitives: State the output first, then the inputs followed by the control input. For an adder, you might code the design by starting with the most significant output and proceed to the least significant input. For example, *carry_out, sum, a, b,* and *carry_in.* For more complex logic, make up rules of your own: Are clocks described first or last? Do inputs come first, or do outputs? Should ports be in alphabetical order?

Even having a good set of rules for port order does not always prevent mismatched port order. Whenever you create an instance of a module, compare the instance against its declaration.

MISSING OR INCORRECT DECLARATIONS

Missing declarations can generate various warning and error messages. A missing reg declaration might generate a message about an illegal assignment or missing declaration. A missing width declaration can generate a warning about mismatched port sizes, illegal use of bit selects or part selects, and other messages.

Missing Regs

In behavioral modeling, most of the data objects you use in the model are regs. In Verilog, you must declare a reg before you use it. There are two places where you might forget a reg declaration: Your outputs may need to be declared as *regs*; internal storage and temporary variables may need to be declared as *regs*.

Look at every procedural assignment. The object on the left-hand side of the assignment must be declared as a *reg* (or possibly as an *integer, real,* or *time*). The error messages for a missing reg declaration points to the procedural assignment. If

you have an error message about the left-hand side of a procedural assignment, chances are you are missing a reg declaration.

Missing Widths

Missing width declarations can be more difficult to find than missing *reg* declarations. It is possible that no error or warning message will be issued for a missing width declaration. It is easy to forget to declare the width of a *port, net, reg,* or *function.* Carefully check each of your modules for the width of each of these.

Each port of a different width needs to be declared separately. If the width of a port is not declared, it defaults to a width of 1 bit. For example, you might discover the error of having a 1-bit-wide port when you wanted a wider bus. This error could appear as a port size mismatch in an upper module instead of the submodule, where the error actually occurred. Error messages about illegal bit or part selects may occur if the module with the missing port width uses bits of the bus separately. It is possible that no error message will be generated, but the circuit may give incorrect results due to only passing around 1 bit of information when several were expected.

All buses (multibit wires or vectors) used within a module must be declared. You might have omitted a net and width declaration, with the result that you see not an error message, but only incorrect behavior. Verilog generates implicit nets for every undeclared wire name that is used in a module or primitive instance. If an implicit net is generated due to a missing declaration, a warning about port size mismatches may be issued. If you declare a net but omit its width, there may be no error message—merely incorrect results. This is especially true if that net is used only in continuous assignments. If you are expecting to use any multibit nets, carefully check their width declarations.

By default, *regs* are only 1 bit wide. Because *regs* are primarily used in procedural assignments, there may be no error or warning messages generated from a missing or incorrect width. The only symptom of a missing or incorrect range (width) may be incorrect behavior. A *reg* only holds as large a value as the declared width allows. A common type of error would be to check an 8-bit *reg* for the value 256 or greater than 255. Carefully look at all assignments and comparisons of *regs* and make sure that the *reg* is declared large enough to hold the largest required value.

By default a *function* returns only 1 bit. If you want to return a multibit value from a *function,* the width of the *function* must be declared. The error of omitting the width of a *function* does not generate any error or warning message. A *function* with a width of 1 bit can only return the value 1 or 0, so this error manifests itself only in incorrect behavior.

Reversed Ranges

When you declare the range of a *port, net, reg,* or *function,* you are declaring width, the index of the most significant bit (MSB), and the index of the least significant bit LSB). The left index is always the index of the MSB and the right index is the index of the LSB. If the index of the MSB is greater than the index of the LSB, the range is descending. If the index of the MSB is less than the index of the LSB, the range is ascending. Both ascending and descending ranges are legal in Verilog.

You can use both ascending and descending ranges within the same module. Once the range is declared for an object, it must always be used with that same range. You cannot use an ascending range part select on an object declared with a descending range. Using ascending and descending ranges on the same object causes an error message.

A more subtle error (one that can be more difficult to debug) occurs when you connect something that is declared to be ascending to something that is declared to be descending. The connection could occur in a module instance, procedural assignment, or continuous assignment. Although connecting objects with reversed ranges is not an error, such a practice can be confusing. MSBs are always connected to MSBs, so the resulting bit indexes are reversed. The best way to avoid the confusion created by reversed ranges is to adopt a standard of using only ascending or descending ranges for your project.

In sum, incorrect declarations can cause misleading error messages or merely incorrect behavior. This section did not list every possible problem, but tried to highlight some of the more common errors of this type. You have to check that the declaration of each object you use matches its use and your expectations. For example, a novice might be unsure when to declare a *net* versus a *reg.* If you are unsure what should be a *net* and what should be a *reg,* reread the sections of Chapter 9 that compare procedural and continuous assignments. Omitting a range from a declaration results in a width of only 1 bit. The most common range-related error occurs when values are too large for the declared range.

IMPROPER USE OF PROCEDURAL CONTINUOUS ASSIGNMENTS

Modeling *reset* is the most common usage for the procedural continuous assignment. Often novice users inadvertently create procedural continuous assignments while trying to resolve syntax errors. Remember that if you have a *reg* declaration and the *assign* keyword, you have a procedural continuous assignment. If you are not trying to model *reset,* the procedural continuous assignment is generally the wrong construct to use.

If your goal is to model sequential logic, you should generally remove the *assign* keyword and change your code to use procedural assignments.

If you are trying to model combinatorial logic, there are two possible methods of removing a procedural continuous assignment: Remove the *reg* declaration and *always* block, and change the code to be a continuous assignment; or remove the *assign* keyword and change the code to a procedural assignment.

If you have the *assign* keyword inside an *always* block, you have created a procedural continuous assignment and it is most likely the wrong thing to do.

MISSING *INITIAL* OR *ALWAYS* BLOCKS

The compiler catches this type of error, but the error message may simply indicate a syntax error. The key questions to ask are these: When do you want the code to run? and When did you tell Verilog to run it? You will recall that *initial* and *always* are the only two starting places for behavioral code in Verilog.

There are many constructs in Verilog that can go directly into a module. However, some statements (such as *begin-end, if,* and *case*) cannot go directly into a module. They all need a starting place. A common error is to omit an *always* statement and some event specification. For example, if you have a *case* statement in a module that is not started by an *initial* or *always,* this is an error, and Verilog prints an error message.

Look at each block of code in your modules and make sure you know when you want them to run. Ensure that your Verilog description matches the point at which you expect your code to run.

ZERO-DELAY *ALWAYS* LOOPS

A zero-delay *always* loop prevents other code within your simulation from progressing beyond time 0. If you have an *always* loop with no delay or event operators in it, perhaps your model would be better off using a continuous assignment because your model would be simpler, and more correct.

Most commonly, every *always* statement is immediately followed by the *wait for event* (@) operator. If you have any *always* loops without the *wait for event* operator, you may unintentionally introduce a zero-delay *always* loop. If you do not have an @ but have the *wait( )* (level-sensitive delay operator), you may still have a zero-delay loop when the condition for the *wait( )* is true.

Examine each of your *always* loops and answer these two questions: When does this *always* loop run? How long does this *always* loop take to run?

INITIAL INSTEAD OF *ALWAYS*

An *initial* block runs only once. If you use an *initial* block to model the function of your circuit, the block runs only once. Most models are supposed to run more than once. For example, whenever the inputs change. If you start the behavior of your model with only an *initial* block, the model will run only once and at time 0. Of course there are exceptions, because there are looping constructs you could put inside the *initial* block.

You should reserve the *initial* block for stimulus and initialization. Look at each *initial* block and ask yourself: When should this code run? If the code should run once, starting at time 0, then it is a valid *initial* block. If the code should run when the inputs change, you should model with an *always* block.

MISSING INITIALIZATION

Every wire and reg in Verilog starts out unknown. Any circuit that depends on a known value to get started needs some form of initialization. Make sure that any counter or state machine has a *reset* or some way to get into a known state. The simplest case of a missing initialization is a clock generator.

Example 20-1 Missing Initialization

```
module cgen_ni(clock);
output clock; reg clock;
always #50 clock = ~clock;
endmodule
```

The clock generator in Example 20-1 seems to generate a clock with a period of 100. What is the initial value of *clock?* The *clock* output starts out unknown. The complement of an unknown is an unknown, so this clock never outputs anything other than an unknown.

State machines modeled with a *case* statement can use a *default* clause to get the state register into a known state. Using a *default* also reduces the chances that the final hardware can get stuck in an unknown state.

There must be some way for each of the regs in every model to get to a known value. Try to avoid adding an *initial* statement to set each reg. If each reg is

arbitrarily initialized, the model may work in simulation, but the hardware may not. There is no way to know the power-up state of every flip-flop. The preferred method to ensure a known state is to use *load* and *reset* signals driven from the stimulus to model the way the final hardware is reset to a known state.

OVERLY COMPLEX CODE

If a section of code looks confusing, it has a good chance of being wrong. Extra code tends to creep into modules during the initial debugging process. To suppress compiler errors, a novice may add additional code, additional declarations, or may create procedural continuous assignments. A simple bidirectional buffer that could be modeled with only two continuous assignments could grow to have two temporary regs, two *always* blocks, *if* statements, and procedural continuous assignments, for a total of ten or more lines of code.

If a section of code looks too complex for the function you intend to be modeled, it is likely that the model started off with the wrong construct. Start with the basics: Is the function being modeled combinatorial or sequential? What constructs are best for that type of logic?

UNINTENDED STORAGE

Regs and *always* blocks are a powerful and versatile way to model almost anything in Verilog. To avoid storing a value unintentionally, you must know when values in regs are updated. Occasionally a reg retains an old value when the value should change. Remember that a reg will hold a value forever until that value is changed. If your intended model is combinatorial, make sure that the code is evaluated whenever any inputs change. For combinatorial logic, each *if* should have an *else,* and all branches of *case* statements must be specified or a *default* clause is needed.

TIMING ERRORS

Behavioral modeling can create some timing situations that are incorrect (or even impossible) in hardware. You must carefully model each section of code, bearing in mind when it should run, when it should sample values, and when it updates those values.

The style guideline of using blocking for combinatorial logic and unit delay non blocking for sequential logic presented in Chapters 9 and 16 may help reduce timing errors from race conditions.

Negative Setup Time

Negative setup time is easily modeled in Verilog. Negative setup is simply sampling an input after the clock. Example 20-2 shows a simple register with negative setup time.

Example 20-2 Negative Setup Time

```
module reg_ns(q, clock, d) ;
output q; reg q;
input d, clock;
always @(posedge clock) #1 q <= d;
endmodule
```

At first glance, the negative setup merely looks like a *clock-to-q* delay. If you look carefully at the code, you will notice that *d* is actually sampled one time unit after the clock. This is a negative setup. The *clock-to-q* delay can be maintained without the negative setup by using the intra-assignment delay as shown in Example 20-3.

Example 20-3 Corrected Register

```
module reg_ok(q, clock, d);
output q; reg q;
input d, clock;
always @(posedge clock) q <= #1 d;
endmodule
```

Carefully examine when inputs are sampled relative to the clock. Make sure the timing of the sampling is representative of a real circuit.

Zero-Delay Races

In Verilog it is possible to perform many operations in zero time. Even though operations take place in zero time, they still have some order in which they must occur. A net or reg can change values many times during the same time unit. If a model is sampling a net or reg during the same time unit when it changes, does the model see the final value, the previous value, or an intermediate value?

In gate-level modeling, regs and combinatorial circuits typically have delays, so zero-delay race conditions are eliminated. In behavioral modeling, regs and combinatorial circuits can have zero delay. Thus it is possible that a value may be sampled before it has a chance to change, or a value may be sampled after it changed when in fact the previous value was desired. It is also possible that a circuit

may behave properly with zero-delay races. It is best to avoid zero-delay races because they can cause unpredictable results. The easiest way to avoid zero-delay races it to model regs and/or combinatorial logic with unit delays.

Make sure you know when values are sampled and generated. Use a delay between sampling and generating new values. Remember to allow for delay between the time that new values are generated and when they are sampled to ensure sampling the new values.

TOOL SPECIFIC PRAGMAS

Tool specific pragmas should be avoided.   Improper use of directives such as *full_case* or *parallel_case* may create logic that simulates differently from the resulting synthesized logic.

This Page Intentionally Left Blank

21 DEBUGGING A DESIGN

It is obviously much easier and less expensive to find and debug an error in your Verilog code than in a completed chip. In Verilog you can watch the exact sequence of events, look at values buried within the circuit, and even see what is driving a multiply driven signal. It takes technique, strategy, and experience to find errors quickly and correct them. This chapter explains a few basic techniques and provides strategies for when to apply those techniques.

OVERVIEW OF FUNCTIONAL DEBUGGING

During functional debugging, you check the assumptions of behavior in the model against the assumptions of behavior in the test bench, and against the assumptions you make while observing and interpreting results. Debugging can be as simple as manually applying stimulus and looking at the resulting printouts or waveforms. Debugging can get as complicated as trying to determine if an error is in a program running on a simulated computer, in the simulation models, or in the test bench.

One thing to remember about debugging is this: The more information you extract from the simulation, the slower the simulation will go. Printing information to the

screen, saving values to a file, or sending signals to a waveform all slow down simulation. If you print out every value in your circuit at every simulation time, the simulator will spend all its time writing out values, and you will spend all your time analyzing them. A more efficient approach is to start out by examining a few critical signals and then narrowing the scope of your debugging. Working on a small, focused portion of the circuit at a time has a minimal effect on the simulator and allows you to understand all the values you are examining.

Where Are the Errors?

During functional debugging, you are as likely to find errors in the test bench as you are in the design. Because there can be more code in the test bench than in the model, you may actually find more errors in the test bench! Another common place to find errors is in the interconnection between modules. If the original specification was subject to interpretation, the assumptions made in one module may not match the assumptions in a connected module.

During the debugging, remember that errors can be anywhere—so look carefully at everything before you rush to correct anything.

UNIVERSAL TECHNIQUES

There are some universal debugging techniques that work no matter what you are debugging. These techniques are looking at values, and seeing what is active. These are universal techniques since they work with Verilog, software, and even oscilloscopes.

Printing Out Messages

The easiest way to find out what is going on inside your models is to print out messages. There are two basic types of messages to print out: messages that indicate when code is running, and messages that report on values that are received or generated.

"I am here."

The simplest type of message to print out is a message that indicates where a block of code is running. You can add a *$display* statement inside an *always* block to indicate when certain tests are about to start, or to indicate the completion of a test.

Adding a *$display* statement to an *always* block can tell you when that block was triggered. Even a *function* can have *$display* statements in it. Knowing what is running gives you some insight into the functionality of the design. Knowing whether or not a critical test or event takes place can greatly narrow down the search for bugs.

Values

Adding statements to print out the values received and generated by a *function* or *always* block allows you to check visually for proper behavior of a block of code. You can then determine if an error is caused by values passed in, or by the values generated.

This method of printing out values can rapidly isolate errors. For example, if you are debugging the ALU exercise with the provided test bench (Chapter 10), the test bench prints out only which pattern number failed. You have no idea which of the ALU's sixteen operations is incorrect. Simply printing the function code input to the ALU isolates a problem within one-sixteenth of the circuit.

In exercise 3 (Chapter 5), values from several levels of hierarchy were printed from the test bench. You can print values from the test bench or add statements into the modules themselves to print out critical values.

If you are printing values out from many areas within the circuit, it may become difficult to determine the origin of a message and the values to which it refers. Make sure that the messages you create clearly indicate where the values are coming from and when.

Be careful when you print out values. If you mistakenly print out values when they are changing, and use a *$display* statement, it is possible that you will see some old and new values being printed. This can clearly make it difficult to interpret results. Thus, it is imperative that the values are stable when the *$display* task runs.

Look carefully at the results from exercise 3 (Chapter 5). The inputs change every 100 time units. The results should be printed every 50 time units. Thus, at time 150, the printout should be correct, and easy to interpret. But at time 100, it is possible that your printout shows the new inputs with the old outputs. An alternative to *$display* is *$strobe,* which waits until everything has finished changing at the current time before *$strobe* prints out a message.

The most common task used to print values is *$monitor,* which prints out whenever any of the signals it is monitoring changes. The output from *$monitor* can be difficult to interpret if you are looking for the timing between changes. There may be ten lines printed one time unit apart, and then no printouts for a thousand time

units, depending on the activity of the signals being monitored. Even though you may be printing out the simulation time along with the signals, you may not easily see the timing relationships. In conclusion, use *$monitor* to see a sequence of changes and not timing; use *$display* or *$strobe* for periodic printouts, or for printouts triggered by interesting sequences of events.

The Log File

All output that goes to the screen also goes into a log file. By default, the log file is named *verilog.log.* You can specify a different name for the log file with the *-l* command-line option. There are two additional commands, *$nolog* and *$log,* that that you can use within Verilog to turn logging off, turn it back on, and open a new log file for following results.

A summary of log file commands and options is shown in Table 21-1.

Table 21-1 Log File Options

Command	Where used	Description
-l filename	Invocation command line	Set the name of the log file
$nolog	Interactive command	Turn off logging
$log	Interactive command	Turn on logging into *verilog.log*
$log("filename")	Interactive command	Start logging into *filename*

USING WAVEFORMS

Waveforms are a great tool for visualizing both the values and the timing of circuits. Every Verilog simulator has a waveform tool. All these tools allow you to select signals to be written to the waveform display. Some allow interactive, marching waveform displays; others are only postprocessors that let you look at waveforms after simulation is complete. All waveform displays allow you to measure the distance between edges of signals. Consult the documentation for the tool set you are using for exact instructions on using a waveform tool.

This technique is limited by how much data you can analyze with a graphical waveform. All the data from a small circuit that runs for a short time can be

displayed and analyzed in a waveform. There is usually too much data from a large simulation for effective graphical viewing and analysis.

Waveforms are the easiest way to verify the timing of critical signals. It is often much easier to analyze timing information with a graphical waveform than with a textual printout.

You can run a simulation and save a set of waveforms and then compare those waveforms to the waveforms from another simulation run. Graphical analysis of the differences between regression tests can be an easy way to see if a difference in test results is critical or acceptable.

Example 21-1 Initial Block to Create VCD Wave File

```
initial begin
  $dumpfile("myfile.vcd"); // file name for wave data
  $dumpvars(0,dut);  // dump all starting at dut hierarchy
end
```

The most common format for wave data is VCD format. VCD stands for *Value Change Dump.* VCD is not a particularly efficient format, but it is recognized by most wave display tools. Example 21-1 shows an initial block you can add to your code to generate a full wave database for you design; be sure to use a proper file name and select the hierarchical name for what to dump.

Example 21-2 Initial Block to Create SHM Wave File

```
initial begin
  $shm_open("waves.shm");
  $shm_probe(dut, "AS");
end
```

An alternate format for wave files is SHM, SHM stand for *Simulation History Manager.* SHM format is used by Cadence products. Example 21-2 shows an initial block that will generate a wave file in SHM format.

Many other formats for wave files exist. Consult the documentation for your simulator and wave viewer for complete details. Most simulators and wave viewers support the industry standard VCD format and commands.

A common problem when looking at wave files is trying to figure out what data is seen at a clock edge if the data changes at the same time as the clock. If you follow the suggestion for unit delay sequential blocks, it will be clear what data is seen at the clock, and what changes are as a result of the clock.

INTERACTIVE DEBUGGING

Interactive debugging is the ultimate way of finding problems in a simulation. You can look at any value; see what is driving a wire; change values on wires and regs; set breakpoints; add signals to a waveform display; and so on. The entire simulation becomes an open book for you to look at.

This book has attempted to be simulator-independent. This section provides details on interactive debugging using commands native to Verilog-XL™. Some clone simulators may include some or all of these commands and features by the same name or by using other commands. Companies that primarily use simulators other than Verilog-XL often have some copies of Verilog-XL for debugging use.

Going Interactive

The main command for going interactive is *$stop.* The *$stop* command is a breakpoint that stops simulation and places the simulator into interactive mode. There are other ways to put the simulator into interactive mode. It is often desirable to start the simulator in interactive mode. You can start the simulator in interactive mode by using the *-s* command-line option. The *-s* stands for "stop at time 0." It is equivalent to having *initial $stop;* in one of your Verilog modules.

You can also put Verilog into interactive mode by interrupting it. You can start Verilog in the normal fashion, and they hit the interrupt key. The interrupt key can be a button in a graphical user interface, or a keyboard key. The interrupt key is most commonly *Control-c.* For the purposes of this section, *Control-c* is used as the interrupt key. If your system uses a different key, use that one instead.

Another way of entering the interactive mode is hitting a breakpoint. Verilog-XL includes a number of specialized breakpoint commands for breaking on edges of signals, on particular values, or at a particular time. All of these breakpoint commands are similar to encountering a *$stop* statement. If you are using the graphical user interface to Verilog-XL, the menu commands for breakpoints use these special commands. Breakpoints are usually set after Verilog is in interactive mode. To enter the interactive mode, start Verilog with the *-s* command-line option. Breakpoints are typically added by typing interactive commands or by using a graphical user interface.

Now we will turn to examples of using Verilog interactively. Enter the small module in Example 21-3 and run it in Verilog. Verilog always needs at least one module to work with, so this is the smallest example you can use to demonstrate interactive Verilog.

Example 21-3 Interactive Verilog Module

```
module interact;
initial $stop;
endmodule
```

The Prompts

Once Verilog has completed compiling the module you will see an interactive prompt on your screen. Verilog numbers the prompts for history and reuse. The first prompt is C1>. If you have run Example 21-3, your screen should look like Example 21-3 Results 1.

Example 21-3 Results 1

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

L2 "interact.v": $stop at simulation time 0
Type ? for help
C1 >
```

Verilog does not have a special language for interactive debugging. You use the same commands you have already learned (in behavioral modeling) for interactive debugging. There are a few special keystrokes, and a few commands, that are most useful for interactive debugging that will be introduced in this chapter.

Each prompt is like an *initial* statement. What can you do at an *initial* statement? The first simulation in exercise 1 (Chapter 3) was to print "Hello Verilog," so try to do that interactively. Type *$display("Hello Verilog!");* at the C1> prompt. Did it print out "Hello Verilog!"? The most common error in interactive debugging is to forget the semicolon. In Verilog, each statement must end with a semicolon, and interactive statements are no exception to this rule. If you forgot the semicolon, your screen will look like Example 21-3 Results 2. You will see a prompt with no number; this is a continuation prompt, and Verilog is asking you to complete the statement.

Example 21-3 Results 2

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

L2  "interact.v": $stop at simulation time 0
Type ? for help
C1 > $display("Hello Verilog interactive")
>
```

If you forgot the semicolon, add it now. If you included the semicolon, you already have the printed message on your screen.

How do you end an interactive simulation? The same way you end a regular simulation: with *$finish.* Type *$finish;* as an interactive command and see what happens. Your simulation should appear as in Example 21-3 Results 3.

Example 21-3 Results 3

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

L2  "interact.v": $stop at simulation time 0
Type ? for help
C1 > $display("Hello Verilog interactive")
> ;
Hello Verilog  interactive
C2 > $finish;
C2: $finish at simulation time 0
4 simulation events
End of VERILOG-XL
```

Now that you have entered interactive mode, typed a few commands, and exited interactive mode, some of the output needs careful inspection and explanation. Just before the first prompt, the message *L2 "interact.v": $stop at simulation time 0* was printed. This message explains why the simulator is in interactive mode. The message tells us that line 2 (*L2*) of the file *interact.v* caused a *$stop.* The message also reports the current simulation time, which is time 0.

Likewise, after you typed the *$finish;* the simulator *reports C2: $finish at simulation time 0.* The *C2:* indicates that interactive command 2 is what caused the *$finish.* These little indicators of what is causing a breakpoint or finish become more important when you have many possible breakpoints.

Special Keys in Interactive Mode

You have already learned that it is important to remember the semicolons in interactive mode, just as in Verilog source files. There are two other equally important keys. The period key has a special meaning in interactive Verilog; it means "continue." The comma key means "single step", or executed as a single statement. Enter the module in Example 21-4 and start Verilog with the *-s* option, *verilog -s sstep. v.*

Example 21-4 Single-Stepping

```
module sstep;
initial begin
  $display("this is a first message");
  $display("this is a second message");
  $stop;
  $display("this is a third message");
  $display("this is a fourth message");
end
endmodule
```

If you remembered the *-s* option, your screen should look like Example 21-4 Results 1.

Example 21-4 Results 1

```
verilog -s sstep.v
Compiling source file "sstep.v"
Highest level modules:
sstep

Type ? for help
C1 >
```

Notice that there is no line or command indicating when or why Verilog went into interactive mode. Add a few interactive breakpoints: type *#10 $stop;* and then *#100 $stop;*. Your screen should now look like Example 21-4 Results 2.

Example 21-4 Results 2

```
verilog -s sstep.v
Compiling source file "sstep.v"
Highest level modules:
sstep

Type ? for help
C1 > #10 $stop;
C2 > #100 $stop;
C3 >
```

There are now three *$stops* in your simulation. The first is in the module *sstep,* and the other two are from interactive commands. The *$stop* in *sstep.v* is set for time 0 after the first two *$display* statements. To tell Verilog to continue running the simulation until the next *$stop,* a special key (period) is used. Type a period, then press *Return,* and you should have the results shown in Example 21-4 Results 3.

Example 21-4 Results 3

```
verilog -s sstep.v
Compiling source file "sstep.v"
Highest level modules:
sstep

Type ? for help
C1 > #10 $stop;
C2 > #100 $stop;
C3 > .
this is a first message
this is a second message
L5 "sstep.v": $stop at simulation time 0
C3 >
```

Notice that Verilog tells us this *$stop* occurred from executing line 5 of the *sstep.v* file. Now try single stepping. What is the next command to run? The comma key is used to single step in Verilog. Type a comma and press *Return.* The screen should now look like Example 21-4 Results 4.

Example 21-4 Results 4

```
verilog -s sstep.v
Compiling source file "sstep.v"
Highest level modules:
sstep

Type ? for help
C1 > #10 $stop;
C2 > #100 $stop;
C3 > .
this is a first message
this is a second message
L5 "sstep.v": $stop at simulation time 0
C3 > ,
L6 "sstep.v": $display("this is a third message");
this is a third message
C3 >
```

Try two more single steps in a row: enter comma, comma, and then press *Return.* Notice that the fourth message executes. The *end* on line 8 of *sstep.v* is reached. Simulation time advances to 10 and the #10 of the #*10 $stop;* (entered as command 1) has elapsed and the balance of command 1 is ready to continue. Example 21-4 results 5 shows the current state of the simulation.

Example 21-4 Results 5

```
verilog -s sstep.v
Compiling source file "sstep.v"
Highest level modules:
sstep

Type ? for help
C1 > #10 $stop;
C2 > #100 $stop;
C3 > .
this is a first message
this is a second message
L5 "sstep.v": $stop at simulation time 0
C3 > ,
L6 "sstep.v": $display("this is a third message");
this is a third message
C3 > , ,
L7 "sstep.v": $display("this is a first message");
this is a first message
L8 "sstep.v": end
SIMULATION TIME IS 10
C1: #10 >>> CONTINUE
C3 >
```

Because it is now time 10, what happens if you enter *#20 $stop;* and then continue? At what time will the simulation stop? It will stop at time 30. Each interactive prompt is like an *initial* statement, except the commands do not have to start at time 0, they start at the current time. Because the current time is 10, 10 + 20 = 30, and time 30 will be the new breakpoint. Try it: You should get the results shown in Example 21-4 Results 6.

Example 21-4 Results 6

```
verilog -s sstep.v
Compiling source file "sstep.v"
Highest level modules:
sstep

Type ? for help
C1 > #10 $stop;
C2 > #100 $stop;
C3 > .
this is a first message
this is a second message
L5 "sstep.v": $stop at simulation time 0
C3 > ,
L6 "sstep.v": $display("this is a third message");
this is a third message
C3 > , ,
L7 "sstep.v": $display("this is a first message");
this is a first message
L8 "sstep.v": end
SIMULATION TIME IS 10
C1: #10 >>> CONTINUE
C3 > #20 $stop;
C4 > .
C1: $stop at simulation time 10
C4 > .
C3: $stop at simulation time 30
C4 >
```

The first period continued the simulation until command 1 finished executing its *$stop.* The second period continued simulation until the new *$stop* (just entered as command 3) executes.

Another way to start an interactive debugging session is to use *Control-c* to interrupt the running simulation. Example 21-5 is a simple module that contains an *always* loop. If you run this module in Verilog, the simulation will never terminate on its own.

Example 21-5 *always* Loop Module

```
module loop;
 integer count;
always
  #1 count = count + 1;
endmodule
```

Simulate the module in Example 21-5. The simulation starts, and then nothing more is printed out. Type *Control-c.* Verilog responds by entering interactive mode. Now you can see the value of *count.* Now type *$display(count);* and note that the value of *count* is unknown because it was never initialized. Because the *always* loop take one time unit, the time shows how many times the simulation went through the loop before you hit *Control-c.* The results of this simulation are shown in Example 21-5 Results 1.

Example 21-5 Results 1

```
verilog loop.v
Compiling source file "loop.v"
Highest level modules:
loop

VERILOG interrupt at time 13272
Type ? for help
C1 > $display(count);
          x
C2 >
```

Because Verilog is in interactive mode, you can even set the value of *count,* and continue the simulation. Type *count = 0;* to set count to a known value, single step it a few times, continue the simulation, interrupt it again, and then finish it. Results are shown in Example 21-5 Results 2.

Example 21-5 Results 2

```
C2 > count = 0;
C3 > ,,,,,
SIMULATION TIME IS 13272
L4 "zdloop.v": #1 >>> CONTINUE
L4 "zdloop.v": count = count + 1; >>> count = 32'h1, 1;
L3 "zdloop.v": always
L4  "zdloop.v": #1
SIMULATION TIME IS 13273
L4 "zdloop.v": #1 >>> CONTINUE
L4 "zdloop.v": count = count + 1; >>> count = 32'h2, 2;
L3 "zdloop.v": always
C3 > .
VERILOG interrupt at time 24705
C3 > $display(count);
      11433
C4 > $finish;
C4: $finish at simulation time 24705
End of VERILOG-XL
```

The special keystrokes used in interactive simulation are shown in Table 21-2.

Table 21-2 Special Keys for Interactive Simulation

Keystroke	Description
.	Continue simulation
,	Single step
Control-c	Interrupt simulation

Command History

When you type *$history;* Verilog shows you a list of all the commands you have
typed so far in the current simulation. Example 21-4 Results 7 shows the output for
the history command. The commands marked with asterisks are active commands.
Command 2 is active because it has not finish running. Command 4 is the currently
running history command. Commands 1 and 3 have finished running and are no
longer active.

Example 21-4 Results 7

```
C4 > $history;

Command history:
C1  #10
        $stop;
C2* #100
        $stop;
C3  #20
        $stop;
C4* $history;

C5 >
```

Now add one more command to explore reactivating and deactivating commands. Type *forever #5 $display("This message repeats");* Type *1 <Return>* which reactivates command 1 (the *#70 $stop;*), which should now stop at time 30. Type *3 <Return>* which reactivates command 3 (the *#20 $stop;*), which should now stop at time 40. Type *4 <Return>* to reactivate the history command to see what has happened. Your screen should now look like Example 21-4 Results 8.

Example 21-4 Results 8

```
C5 > forever #5 $display("This message repeats");
C6 > 1
C6 > 3
C6 > 4

Command history:
C1* #10
        $stop;
C2* #100
        $stop;
C3* #20
        $stop;
C4* $history;
C5* forever
        #5
            $display("This message repeats");

C6 >
```

Now continue the simulation twice more and note that the command 1 and command 3 breakpoints have taken their new times. The screen should now look like Example 21-4 Results 9.

Example 21-4 Results 9

```
C6 > .
This message repeats
C1: $stop at simulation time 40
C6 > .
This message repeats
This message repeats
C3: $stop at simulation time 50
C6 >
```

Now type five commas and then press *Return* and see what happens. You can see
the next five statements execute. Rerun the history command: Command 4 will
show that commands 2 and 5 are still active. Your results should look like Example
21-4 Results 10.

Example 21-4 Results 10

```
C6 > ,,,,,
SIMULATION TIME IS 50
C5:   #5 >>> CONTINUE
C5:   $display("This message repeats");
This message repeats
C5:   forever
C5:   #5
SIMULATION TIME IS 55
C5:   #5 >>> CONTINUE
C5:   $display("This message repeats");
This message repeats
C5:   forever
C6 > 4

Command history:
C1   #10
        $stop;
C2*  #100
        $stop;
C3   #20
        $stop;
C4*  $history;
C5*  forever
        #5
            $display("This message repeats");

C6 >
```

Entering the number for a command reactivates the command: How do you
deactivate a command? If you wanted to continue running the simulation until the
breakpoint at time 100 caused by command 2, the message from command 5 would
repeat many more times. Deactivate command 5 by typing *-5*. Look at the history

again and continue the simulation. You should now see the results shown in Example 21-4 Results 11.

Example 21-4 Results 11

```
C6 > -5
C6 > 4

Command history:
C1   #10
        $stop;
C2*  #100
        $stop;
C3   #20
        $stop;
C4*  $history;
C5   forever
        #5
            $display("This message repeats");

C6 > .
C2: $stop at simulation time 100
C6 >
```

Another period or comma and the simulation will end the simulation because there is nothing left to execute.

The Key File

Verilog creates a record of your keystrokes in a file called *verilog.key*. Look at the *verilog.key* file shown in Example 21-4 Results 12.

Example 21-4 Results 12 The *verilog.key* File

```
#10 $stop;
#100 $stop;
.
,
,,
#20 $stop;
.
.
$history;
forever #5 $display("This message repeats");
1
3
4
.
.
,,,,,
4
-5
4
.
,
```

The keystroke file contains everything you typed during the Verilog interactive
session. You can use the keystroke file to replay the simulation. Copy or rename the
keystroke file to a new name, such as *copy.key*. Copy or rename the log file to a new
name, for example, *copy.log.* This will prevent the old log file and keystroke file
from being overwritten by a subsequent simulation. Now rerun the previous
simulation with the command *verilog -s sstep.v -i copy.key.* Note how the entire
simulation instantly repeats, as shown in Example 21-4 Results 13.

Example 21-4 Results 13 Replayed Simulation

```
verilog -s sstep.v -I copy.key
Highest level modules:
sstep

Type ? for help
C1 > #10 $stop;
C2 > #100 $stop;
C3 > .
this is a first message
this is a second message
L5 "sstep.v": $stop at simulation time 0
C3 > ,
L6 "sstep.v": $display("this is a third message");
this is a third message
C3 > ,,
```

```
L7 "sstep.v": $display("this is a fourth message");
this is a fourth message
L8 "sstep.v": end
SIMULATION TIME IS 10
C1: #10 >>> CONTINUE
C3 > #20 $stop;
C4 > .
C1: $stop at simulation time 10
C4 > .
C3: $stop at simulation time 30
C4 > $history;

Command history:
C1  #10
        $stop;
C2* #100
        $stop;
C3  #20
        $stop;
C4* $history;

C5 > forever #5 $display("This message repeats");
C6 > 1
C6 > 3
C6 > 4

Command history:
C1* #10
        $stop;
C2* #100
        $stop;
C3* #20
        $stop;
C4* $history;
C5* forever
        #5
            $display("This message repeats");

C6 > .
This message repeats
C1: $stop at simulation time 40
C6 > .
This message repeats
This message repeats
C3: $stop at simulation time 50
C6 > ,,,,,
SIMULATION TIME IS 50
C5: #5 >>> CONTINUE
C5: $display("This message repeats");
This message repeats
C5: forever
C5: #5
SIMULATION TIME IS 55
C5: #5 >>> CONTINUE
```

```
C5: $display("This message repeats");
This message repeats
C5: forever
C6 > 4

Command history:
C1  #10
        $stop;
C2* #100
        $stop;
C3  #20
        $stop;
C4* $history;
C5* forever
        #5
            $display("This message repeats");

C6 > -5
C6 > 4

Command history:
C1  #10
        $stop;
C2* #100
        $stop;
C3  #20
        $stop;
C4* $history;
C5* forever
        #5
            $display("This message repeats");

C6 > .
C2: $stop at simulation time 100
C6 > ,
48 simulation events
End of VERILOG-XL
```

Compare *copy.key* and *verilog.key*: There are no differences. Compare *verilog.log* and *copy.log.* The only differences are the clock times reported by the simulator.

There are two ways of replaying a keystroke file. You can use the *-i* command line option to read in a keystroke file. The other way to read in a keystroke file is by using the *$input()* command. Enter the commands in Example 21-6 into a file called *my.key.*

Example 21-6 *my.key* Command File

```
$display("This message is from my.key");
$display("This message is also from my.key");
#10 $stop;
.
```

Rerun *sstep.v* with just the *-s* option: *verilog -s sstep.v*. At the first prompt, type *$input("my.key");* Notice that the commands from the file *my.key* are read in and acted upon as if you had typed them. Example 21-6 Results 1 shows the resulting output.

Example 21-6 Results 1

```
verilog -s sstep.v
Compiling source file "sstep.v"
Highest level modules:
sstep

Type ? for help
C1 > $input("my.key");
C2 > $display("This message is from my.key");
This message is from my.key
C3 > $display("This message is also from my.key");
This message is also from my.key
C4 > #10 $stop;
C5 > .
this is a first message
this is a second message
L5 "sstep.v": $stop at simulation time 0
C5 >
```

If you want, you can reactivate command 1; you could use *$input* to read in *copy.key*; or you can finish the simulation.

You can create command files to help automate common sequences of interactive commands. You can create the command files by editing keystroke files or you can create them from scratch.

There are a few remaining details to complete the discussion on keystroke and command files. The *-k* command line option allows you to specify an alternate file for writing the keystrokes. The command *$nokey* turns off the capture of keystrokes. The command *$key* turns keystroke writing back on, or can change the name of the file capturing the keystrokes.

A brief demonstration of all the keystroke file options can be shown by running Example 21-3 with the command *verilog interact.v -k interact.key* and by typing the additional commands as shown in Example 21-3 Results 4.

Example 21-3 Results 4 On the Screen

```
verilog interact.v -k interact.key
Highest level modules:
interact

L2 "interact.v": $stop at simulation time 0
Type ? for help
C1 > $display("hello");
hello
C2 > $nokey;
C3 > $display("You can't see me");
You can't see me
C4 > $key;
C5 > $display("this is in the key file");
this is in the key file
C6 > $key("another.key");
C7 > $display("this is in the other key file");
this is in the other key file
C8 > .
9 simulation events
End of VERILOG-XL
```

Example 21-3 Results 5 Contents of *interact. key*

```
$display("hello");
$nokey;
```

Example 21-3 Results 6 Contents of *verilog.key*

```
$display("this is in the key file");
$key("another.key");
```

Example 21-3 Results 7 Contents of *another.key*

```
$display("this is in the other key file");
.
```

A summary of keystroke and command script commands and options is shown in Table 21-3.

Table 21-3 Keystroke-Related Commands

Command	Where Used	Description
-k *filename*	Invocation command line	Set the name of the keystroke file
$nokey	Interactive command	Turn off logging of keystrokes
$key	Interactive command	Turn on keystroke logging into *verilog.key* file
$key("*filename*")	Interactive command	Turn on keystroke logging into *filename*
-i *filename*	Invocation command line	Read in interactive commands from *filename*
$input("*filename*")	Interactive command	Read in interactive commands from *filename*

Traversing and Observing

Now that you know the basics of running a simulation in interactive mode, you are ready to run a simulation and traverse through the design and observe values. When a simulation is interactive, you can navigate through the design and observe the values anywhere in the circuit. The commands and examples shown in this section are specific to Verilog-XL. However, other simulators may support these or similar commands.

Table 21-4 lists the primary commands for traversing and observing a design. Each module in Verilog has a unique hierarchical name. The location you are observing the circuit from is called the current scope. The first command, *$showscopes,* asks Verilog what scopes, or hierarchical locations, are available from the current scope. The second command, *$scope,* changes the current scope.

Table 21-4 Commands for Traversing and Observing

Command	Explanation
$showscopes	Show what scopes are available
$scope	Change scope
$list	Show source code and current values
$display	Observe values
:	Show current scope

A hierarchical 8-bit adder will be used as an example because it is a simple circuit, but contains enough hierarchy to make traversing interesting. Example 21-7 runs the adder until there are some known values, and demonstrates traversing.

Example 21-7 Hierarchical 8-Bit Adder

```
/*This file contains the hierarchical adders
  adder8 - 8-bit adder made from two 4-bit adders
  adder4 - 4-bit adder made from two 2-bit adders
  adder2 - 2-bit adder made from two 1-bit full adders
  adder  - 1-bit full adder made from Verilog primitives
*/
module adder8(carry_out, sum, a,b,carry);
output carry_out;
output [7:0] sum;
input  [7:0] a, b ;
input carry;

adder4 hi4(carry_out, sum[7:4], a[7:4], b[7:4], int_carry);
adder4 lo4(int_carry, sum[3:0], a[3:0], b[3:0], carry);

endmodule

module adder4(carry_out, sum, a,b,carry);
output carry_out;
output [3:0] sum;
input  [3:0] a, b ;
input carry;

adder2 hi2(carry_out, sum[3:2], a[3:2], b[3:2], int_carry);
adder2 lo2(int_carry, sum[1:0], a[1:0], b[1:0], carry);

endmodule

module adder2(carry_out, sum, a,b,carry);
output carry_out;
output [1:0] sum;
input  [1:0] a, b ;
input carry;

adder  hi (carry_out, sum[1], a[1], b[l], int_carry);
adder  lo (int_carry, sum[0], a[0], b[0], carry);

endmodule

module adder(cout,sum,x,y,cin);

output sum,cout;
input x,y,cin;
// using xor, and, nor gates to implement a full
//                adder (addbit)
```

```
xor    x1   (half_sum,x,y),
       x2   (sum,half_sum,cin);

and    a1   (ov,half_sum,cin),
       a2   (half_carry,x,y);

or     o1   (cout,ov,half_carry);

endmodule
```

Example 21-7 shows the circuit that will be used for demonstrating hierarchical traversal. Example 21-7 Results 1 shows the initial scope and a method to traverse into the least significant 2-bit adder.

Example 21-7 Results 1

```
verilog adder8.v test_add.v -s
Compiling source file "adder8.v"
Compiling source file "test_add.v"
Highest level modules:
test_adder

Type ? for help
C1 > # 550 $stop;
C2 > .
C1: $stop at simulation time 550
C2 > $showscopes;
Directory of scopes at current scope level:
    module (adder8), instance (dut)

Current scope is (test_adder)
Highest level modules:
test_adder

C3 > $scope(dut);
C4 > $showscopes;
Directory of scopes at current scope level:
    module (adder4), instance (hi4)
    module (adder4), instance (lo4)

Current scope is (test_adder.dut)
Highest level modules:
test_adder

C5 > $scope(lo4);
C6 > $showscopes;
Directory of scopes at current scope level:
    module (adder2), instance (hi2)
    module (adder2), instance (lo2)

Current scope is  (test_adder.dut.lo4)
```

```
Highest level modules:
test_adder
```

The primary reason for traversing the hierarchy is to observe the value of some signal buried in the hierarchy. Example 21-7 Results 2 shows the values of all the inputs and outputs of this module along with the source code, and the value of an internal signal.

Example 21-7 Results 2

```
C7 > $list;
  // adder8.v
  28  module adder4(carry_out, sum, a, b, carry);
  29      output
  29        carry_out; // = St0
  30      output  [3:0]
  30        sum; // = 4'hb, 11 (scalared)
  31      input  [3:0]
  31        a, // = 4'h2, 2 (scalared)
  31        b; // = 4'h8, 8 (scalared)
  32      input
  32        carry; // = St1
  34      adder2
  34        hi2(carry_out, sum[3:2], a[3:2], b[3:2],
int_carry);
  35      adder2
  35        lo2(int_carry, sum[1:0], a[1:0], b[1:0],
carry);
  37  endmodule
C8 > $display(int_carry);
0
C9 > $displayb(sum);
1011
```

The command *$list* decompiles the module at the current scope. This is the easiest way to see the values of the inputs and outputs. As Verilog-XL decompiles or lists the source code, it adds comments showing the current value of each of the ports in both hexadecimal and decimal. The numbers in the left-hand column are the line numbers from the original source file. Looking at the decomposed source and values of inputs and outputs may provide important clues when debugging a circuit. On large circuits, it may not be effective to use *$list* if the modules are large and *$list* would generate too much output to be readable.

Another effective command to observe values is *$display*. The *$display* command can be used to display the value of any signal in any radix. Since *$scope* was used to traverse into a particular hierarchy, the hierarchical names of the displayed signals

are relative to the current scope. Interactive debug is the most common place of using the alternate forms of *$display*.

If you forget where you have traversed to in the hierarchy, the colon (:) special interactive command shows where in the hierarchy you currently are. To traverse back to the top, set the scope to the top-level module. You can also traverse directly to any scope. Example 21-7 results 3 demonstrates these commands.

Example 21-7 Results 3

```
C10 > :
Command 9
Scope is (test_adder.dut.lo4)
C10 > $scope(test_adder);
C11 > :
Command 10
Scope is (test_adder)
C11 > $scope(dut.lo4);
C12 > :
Command 11
Scope is (test_adder.dut.lo4)
```

Back-Tracing Fan-In

Just knowing the value of a signal is not always sufficient to understand the behavior (or misbehavior) or a circuit. Sometimes it is necessary to see what is driving a wire (especially if a wire is driven by more than one source). The command *$showvars* shows all the drivers of a wire.

You can use the combination of the *$scope* command and the *$showvars* command to back-trace the source of a signal as far as needed to isolate the ultimate source of the values you are looking for. This is demonstrated in Example 21-7 Results 4.

Example 21-7 Results 4

```
C12 > $showvars(int_carry);
int_carry (test_adder.dut.lo4) wire = St0
    St0 <- (test_adder.dut.lo4.lo2.hi): or o1(cout, ov,
half_carry);
C13 > $scope(test_adder.dut.lo4.lo2.hi);
C14 > $showvars(ov, half_carry);
ov (test_adder.dut.lo4.lo2.hi) wire = St0
    St0 <- (test_adder.dut.lo4.lo2.hi): and a1(ov, half_sum,
cin);
half_carry (test_adder.dut.lo4.lo2.hi) wire = St0
    St0 <- (test_adder.dut.lo4.lo2.hi): and a2(half_carry, x,
y);
C15 > $showvars(x,y);
x (test_adder.dut.lo4.lo2 .hi) wire = St1
    St1 <- (test_adder.dut): port 3
y (test_adder.dut.lo4.lo2.hi) wire = St0
    St0 <- (test_adder.dut): port 4
C16 > $scope(test_adder.dut);
C17 > $list;
  // adder8.v
  17  module adder8(carry_out, sum, a, b, carry);
  18      output
  18         carry_out; // = St0
  19      output  [7:0]
  19         sum; // = 8'hdb, 219 (scalared)
  20      input  [7:0]
  20         a, // = 8'h12, 18 (scalared)
  20         b; // = 8'hc8, 200 (scalared)
  21      input
  21         carry; // = St1
  23      adder4
  23         hi4(carry_out, sum[7:4], a[7:4], b[7:4],
int_carry);
  24      adder4
  24         lo4(int_carry, sum[3:0], a[3:0], b[3:0],
carry);
  26  endmodule
C18 > $showvars(a,b) ;
```

Using *force* and *release*

You can use the Verilog-XL *force* command to set a net to an expression or to a fixed value. For example you could use *force* to change an AND gate into an OR gate. The *release* command simply undoes *force,* and the net goes back to its previous behavior.

Example 21-7 Results 5 briefly demonstrates *force* and *release.*

Example 21-7 Results 5

```
C20 > $showvars(ov);
ov (test_adder.dut.lo4.lo2.hi) wire = St0
    St0 <- (test_adder.dut.lo4.lo2.hi): and a1(ov, half_sum,
cin);
C21 > force ov = half_sum | cin;
C22 > $showvars(ov);
ov (test_adder.dut.lo4.lo2.hi) wire = St1
    Node is in forced state from: Command 21
    St1 <- (test_adder.dut.lo4.lo2.hi): and a1(ov, half_sum,
cin);
C23 > release ov;
C24 > $showvars(ov);
ov (test_adder.dut.lo4.lo2.hi) wire = St0
    St0 <- (test_adder.dut.lo4.lo2.hi): and a1(ov, half_sum,
cin);
```

Waveforms, Graphical User Interfaces, and Other Conveniences

Waveforms and graphical user interfaces make it easier for you to traverse hierarchy and observe values. Graphical tools can greatly reduce your debugging time by enhancing visualization of values, simplifying hierarchy traversal, and simplifying the selection of signals to display. The techniques demonstrated by Example 21-7 and its results can be repeated using the graphical tools provided with your simulator.

Each set of graphical tools is a little different, but all such tools provide the same basic functionality (demonstrated with Example 21-7) of traversing hierarchy and observing values.

CATCHING PROBLEMS LATER IN A SIMULATION

Catching an error in simulation can be difficult. If an error occurs early in a simulation, you might catch it with tracing from the start. If an error occurs after a large amount of simulation has occurred, it is often more difficult to find and isolate the problem. Looking through a large amount of trace output to find the problem might take a long time. You could run the simulation without tracing until near the time when you expect the error to occur, and then turn on tracing with *$settrace*. You can turn off tracing with the *$cleartrace* command. Tracing always tells you the details of what happened in a simulation.

The traverse-and-observe method is the most common debugging method. However, the trick is to know when to traverse and observe, and to find an easy way to get to the correct simulation time at which to start the traversing and observing.

You have seen how to add a *$stop* interactively or in your source code to try and stop when you think an error is going to occur. Often you may find that you want to start over again and look at the simulation at an earlier time. The command *$reset* resets simulation time to 0, so you can start again and set earlier breakpoints or try other techniques.

If you want to have a known point in the simulation to return to and resume simulation from other than time 0, you can create a save-restore file. At any time during simulation you can issue the *$save* command and Verilog-XL creates a snapshot of the current simulation. You can return to the saved state of the simulation two ways. The *$restore* command reads back in a saved simulation snapshot. The *-r* command line option lets you jump back into a previously saved snapshot. The combination of *$save* and *-r* allows you to save where you are in your debugging process, exit, and then resume your simulation later.

The *$save* and *$restore* commands help you return to a state of simulation that might otherwise be difficult to reach. If you are debugging a problem that only occurs after several hours of simulation, you can use *$save* to save the circuit a few hours into the simulation. As you are debugging, rather than starting from time 0 each time you are looking for the error, you can restore and save the time it took to get to that state. You can have as many save-restore files as you have disk space.

Table 21-5 lists the commands and options for tracing, saving, and restoring.

Table 21-5 The *trace*, *save*, and *restart* Commands

Command	Where Used	Description
-t	Invocation command line	Start tracing at time 0
$settrace	Interactive command	Turn on tracing
$cleartrace	Interactive command	Turn off tracing
$reset	Interactive command	Reset to time 0
-r *filename*	Invocation command line	Restart from saved file
$save("*filename*")	Interactive command	Save current simulation state
$restart("*filename*")	Interactive command	Load in previous simulation state

ISOLATING DIFFERENCES IN MODELS

How do you figure out what is wrong when a difference occurs between a behavioral model and a gate-level model? How do you determine what is wrong when one behavioral model works and another one, which is supposedly equivalent, does not?

In such cases, one of the first things to check is timing. Are both models for the circuit expecting inputs at the same time? Are the inputs being provided at the proper times? Different models often have different output timing. A behavioral model might have zero or unit delay, but a gate-level model may have some longer delay associated with the actual gates. Is the circuit taking longer than expected by the test bench or other connection modules to produce correct outputs, or are you just looking for the outputs too soon?

Another common gate-level problem involves unknowns. Use the techniques shown earlier in this chapter to back-trace the source of any unknowns. If the unknowns lead back to flip-flops or registers, perhaps the these are not properly reset.

Another source of unknowns is differences in how certain conditions are handled. The behavioral model for a mux might have a known output when the two inputs are known, but the select might be unknown, and the gate-level model might pass the unknown to the output.

Unknowns may cause other types of mismatches. The pre-synthesis model may have used *x* to indicate a don't care condition to synthesis. After synthesis these *x*'s will become *1* or *0*.

If timing is the only difference between two simulations, is the timing of both models acceptable? Can you modify the test bench to accept both timings? If unknowns constitute the difference, can the circuits and stimulus be modified to eliminate the unknowns? If the difference is functional, which model is correct? Once you know the incorrect behavior, debug it!

SUMMARY OF DEBUGGING

There is no magic secret to debugging a design. Debugging is a process of looking for common errors and then observing the operation of the circuit. With experience, you will develop techniques that are a combination of the commands presented here and other commands. This chapter did not list every possible interactive or debugging command, but it provides the basic, tried-and-true commands and techniques that form the basis of an effective debugging process. Table 21-6 summarizes the commands, special keystrokes, and command-line options related to debugging explained in this chapter.

Table 21-6 Debugging Commands, Keystrokes, and Command-Line Options

Command/Keystroke	Where Used	Description
-s	Invocation	Stop at time 0 before any execution
-l *filename*	Invocation	Set the name of the log file
$nolog	Interactive	Turn off logging
$log	Interactive	Turn on logging into *verilog.log* file
$log("*filename*")	Interactive	Start logging into filename
.	Interactive	Continue simulation
,	Interactive	Single step
control-c	Interactive	Interrupt simulation
-k *filename*	Invocation	Set the name of the keystroke file

Command/Keystroke	Where Used	Description
$nokey	Interactive	Turn off logging of keystrokes
$key	Interactive	Turn on keystroke logging into *verilog.key* file
$key("*filename*")	Interactive	Turn on keystroke logging into *filename*
-i *filename*	Invocation	Read in interactive commands from *filename*
$input("*filename*")	Interactive	Read in interactive commands from *filename*
$showscopes	Interactive	Show what scopes are available
$scope	Interactive	Change scope
$list	Interactive	Show source code and current values
$display	Interactive	Observe values
:	Interactive	Show current scope
$stop	Source code or interative	Enter interactive mode
-t	Invocation	Start tracing at time 0
$settrace	Interactive	Turn on tracing
$cleartrace	Interactive	Turn off tracing
$reset	Interactive	Reset to time 0
-r *filename*	Invocation	Restart from saved file
$save("*filename*")	Interactive	Save current simulation state
$restart("*filename*")	Interactive	Load in previous simulation state

This Page Intentionally Left Blank

22 CODE COVERAGE

Code coverage is a method by which you measure test benches and models. Code coverage tools can help you answer some difficult questions:
- Have I created enough test cases?
- Have I written any code that never gets run?
- Where are the bottlenecks in my simulation performance?

Code coverage tools answer these questions by monitoring the activity in the simulation and measuring the number of times each statement is executed. Code coverage can be used during the design phase and initial testing phase to find sections or lines of design that are not tested.

Code coverage, sometimes called code profiling, can also be used to increase simulation performance. Simulation performance can be tuned by locating the lines of code the simulator executes most frequently. Once the heavily executed lines have been identified, the designer can look to optimize the heavily executed code.

As designs become more complicated, it is important to measure that every branch of each *if* and *case* statement is executed as part of the simulations. If you find statements that are not executed, you need to write more tests that try to exercise the

previously non-executed statements. If you cannot write tests that exercise the non-executed parts of your code, you may have cases in which it is not possible to execute those statements. The non-executed statements may require an impossible set of conditions to be executed. In this case the code may be *dead* code and should be removed. In other cases, the code may simply be difficult to test.

Code coverage and fault simulation are both ways to measure how well a design is tested; Code coverage measures if each line of the code is executed, while fault simulation determines if each potential manufacturing fault can be observed. Code coverage is not a replacement for fault simulation and both may be needed as part of your design flow. 100% code coverage does not ensure 100% fault coverage.

Many simulators have code coverage capabilities. The Silos simulator that accompanies this book makes code coverage quite simple. Enable code coverage from the menu before running simulation. Then select *view line report.* If the source is open, annotations indicate which lines were executed and which lines were not.

CODE COVERAGE AND TEST PLANS

Code coverage can be used as a tool to gauge the effectiveness of a test plan. Example 22-1 is a repeat of the counter from Chapter 10. A test plan for this counter was identified in Chapter 18.

Example 22-1 Repeat of Counter Using *if*

```
module countc(clock, reset, load, up, load_data, count);
input clock,reset, load, up;
input [15:0] load_data;
output [15:0] count;
reg [15:0] count;

always @(posedge clock)
    if (reset)
        count <= 8'b0;
    else
        if (load)
            count <= #(`REG_DELAY) load_data;
        else
            if (up)
                count <= #(`REG_DELAY) count + 1;
            else
                count <= #(`REG_DELAY) count - 1;
endmodule
```

Below is the test plan from Chapter 18. To test the loadable up down counter, four basic tests were identified:

- Test Reset
- Test Load
- Test Count up
- Test Count down

If this test plan is complete, code coverage should prove it. The test bench in Example 22-2 does not follow the test plan; it tests only *reset* and *up*.

Example 22-2 Counter Test Bench #1

```
`timescale 1ns/1ns
`define REG_DELAY 1
`disable_codecoverage
module test_counter1;
reg up, load, clk, reset;
reg [15:0] load_data;
wire [15:0] count;
always begin// generate clock
  #5 clk =0;
  #5 clk =1;
end

// instantiate device under test
countc dut( .clock(clk), .reset(reset), .up(up), .load(load),
        .load_data(load_data), .count(count) );

initial begin // counter test sequence
  reset = 1;
  @(posedge clk) #3 if (count !== 16'h0000)
    begin
      $display("Counter reset error");
      $finish;
    end
  reset = 0; load = 0; up = 1;
  @(posedge clk) #3 if (count !== 16'h0001)
    begin
      $display("Counter count to 1 error");
      $finish;
    end
  @(posedge clk) #3 if (count !== 16'h0002)
    begin
      $display("Counter count to 2 error");
      $finish;
    end
  $finish;
end
endmodule
`enable_codecoverage
```

The test bench in Example 22-2, follows the basic test cycle for sequential models, described in Chapter 18. It applies stimulus, waits for the clock, then allows the circuit to respond, and checks results. The test bench also defines the timing for the module. Code coverage of the test bench is disabled. Since we expect no errors from the counter we expect many of the lines containing error checks in the test bench to be skipped.

Example 22-3 shows the test bench improved to follow the test plan.

Example 22-3 Counter Test Bench #2

```
`timescale 1ns/1ns
`define REG_DELAY 1
`disable_codecoverage
module test_counter2;
reg up, load, clk, reset;
reg [15:0] load_data;
wire [15:0] count;

always begin // generate clock
  #5 clk =0;
  #5 clk =1;
end

// instantiate device under test
countc dut( .clock(clk), .reset(reset), .up(up), .load(load),
        .load_data(load_data),  .count(count) );

initial begin // counter test sequence
  reset =1 ;  // Reset test plan #1
  @(posedge clk) #3 if (count !== 16'h0000)
    begin
      $display("Counter reset error");
      $finish;
    end
  load_data = 16'hl234;
  reset = 0; load = 1; // Load test plan #2
  @(posedge clk) #3 if (count !== 16'hl234)
    begin
      $display("Counter  load error");
      $finish;
    end
  load = 0; up = 1; //Up test plan #3
  @(posedge clk) #3 if (count !== 16'hl235)
    begin
      $display("Counter up error");
      $finish;
    end
  up = 0 ; // Down test plan #4
  @(posedge clk) #3 if (count !== 16'hl234)
    begin
```

```
        $display("Counter count down error");
        $finish;
      end
    $finish;
end
endmodule
`enable_codecoverage
```

CODE COVERAGE AND FIFOS

A FIFO (first in first out buffer) is a good example of code that is difficult to
exercise completely. Conditions such as *reading from an empty FIFO or writing to
a full FIFO* may never appear.

Testing a model with a FIFO is complicated: This model can be fully tested with a
unit level test bench. The unit level test bench can control the rate of the reads and
writes, therefore the conditions of *empty* and *reading from an empty FIFO* can be
tested. The condition of *full* and *writing to a full FIFO* can also be tested. Example
22-4 shows a simple FIFO model.

Example 22-4 FIFO Model

```
module fifo(reset, wr_data, wr_strobe,
            rd_data, rd_strobe, full, empty);
input reset;
input  [7:0] wr_data;
input wr_strobe;
output [7:0] rd_data;
input rd_strobe;
output full, empty;
reg [7:0] fifo [0:15]; // 16 deep fifo
reg [3:0] rd_ptr, wr_ptr;

always @(posedge wr_strobe or posedge reset)
  if( reset )
    wr_ptr <= #(`REG_DELAY) 4 'h0;
  else
    if(full)
      wr_ptr <= #(`REG_DELAY) wr_ptr; // hold
    else
      begin
        wr_ptr <= #(`REG_DELAY) wr_ptr  + 4 'h1;
        fifo [wr_ptr]   <= #(`REG_DELAY)  wr_data;
      end

always @(posedge rd_strobe or posedge reset)
  if( reset )
    rd_ptr <= #(`REG_DELAY) 4'h0;
```

```
  else
    if(empty)
      rd_ptr <= #(`REG_DELAY) rd_ptr; // hold
    else
      rd_ptr <= #(`REG_DELAY) rd_ptr  + 4'h1;

assign rd_data = fifo[rd_ptr];

assign empty = (rd_ptr == wr_ptr);
assign full = (rd_ptr == (wr_ptr + 4'h1));

endmodule
```

Is it possible that less than 100% coverage is a good thing? Yes! The unit level test bench for the FIFO Example 22-5 tests the FIFO 100%. In the system context, the FIFO should become *full* and *empty*. However it would be a functional error if the system simulation read from an empty FIFO or wrote to a full FIFO. The system level code coverage should show that *read from empty* and *write to full* code is not executed.

Code coverage is the easiest way to ensure the difficult conditions *FIFO full* and *FIFO empty* are tested in the system context. Keep in mind that the FIFO itself should not have 100% coverage in the system context, since the system should never read from an empty FIFO or write to a full FIFO.

Example 22-5 is a unit level test bench for the FIFO model. This test bench stimulates the FIFO for 100% line and expression coverage.

Example 22-5 Unit Testbench for FIFO Model

```
`timescale 1ns/1ns
`define REG_DELAY 1
`disable_codecoverage
module test_fifo;
reg reset;
reg [7:0] wr_data, last_data, expect_data;
reg wr_strobe;
wire [7:0] rd_data;
reg rd_strobe;
wire full, empty;

fifo dut (reset, wr_data, wr_strobe,
          rd_data, rd_strobe, full, empty);

initial
  begin
    last_data = 8'hff;
    reset = 1'b1;
```

```
    #10 reset = 1'b0;
    fork
      write_data;
      read_data;
    join
  end

task write_data;
begin
  wr_data = 0;
  while (!full)
    begin
      wr_data =  wr_data +1;
      #1 wr_strobe = 1 ;
      #1 wr_strobe = 0 ;
   end
  $display(
    "Attempting to write to full fifo, last data was %h",
     wr_data);
  last_data = wr_data;
  wr_data = 8'hee; // error data
  #1 wr_strobe = 1;
  #1 wr_strobe = 0 ;
  if(!full)
    $display(
      "Error, No longer full after write to full fifo");
  #1 wr_strobe = 1;
  #1 wr_strobe = 0 ;
  if(!full)
    $display(
      "Error, No longer full after write to full fifo");
end //task
endtask // write_data


task read_data;
begin
  expect_data = 0;
  while(expect_data <= last_data )
    begin
    wait(!empty)
      expect_data = expect_data + 1;
      if(rd_data !== expect_data)
        $display(
          "Error: Wrong read data read %h expected %h",
                    rd_data, expect_data);
      #1 rd_strobe = 1;
      #1 rd_strobe = 0;
      #17 ; // slow read rate
    end // while
  $display("Attempting to read beyond the end of the fifo");
```

```
  if(rd_data !== expect_data)
    $display("Error: Wrong read data read %h expected %h",
                        rd_data, expect_data);
  #1 rd_strobe = 1;
  #1 rd_strobe = 0;
  if(!empty)
    $display("Error, No longer empty after read from fifo");
  if(rd_data !== expect_data)
    $display("Error: Wrong read data read %h expected %h",
                        rd_data, expect_data);
  #1 rd_strobe = 1;
  #1 rd_strobe = 0;
  if(!empty)
    $display("Error, No longer empty after read from fifo");
end //task
endtask
endmodule
`enable_codecoverage
```

CODE COVERAGE AND STATE MACHINES

Complex state machines may have many paths and branches. Even with a good test plan, how can you determine if all the paths are tested? Code coverage! Code coverage can help find untested or untestable parts of a state machine.

As with the case of the FIFO less than 100% coverage may be acceptable for state machines. Next state logic implemented with a *case* statement should include a *default.* A properly designed state machine should never hit the *default* case.

CODE COVERAGE AND MODELING STYLE

Modeling style can have an effect on code coverage. An obsolete modeling style from the early days of Verilog and logic synthesis was composed of only continuous assignments and instances of registers. This style gives erroneously high coverage. Example 22-6 shows a loadable counter done in this obsolete style. Many statements on a single line hamper readability and code coverage. A single statement per line is the best way to have readable code and ensure accurate code coverage information. Example 22-7 shows the counter in an improved style.

Example 22-6 Old Style Counter

```
module count_old(elk, reset, load, ldata, count, match);
input clk, reset, load;
input [7:0] ldata;
```

```
output [7:0] count;
output match;
wire [7:0] next_count;
assign next_count=load ? ldata : count + 1;
assign match=(count==ldata);

reg8 count_reg(clk, reset, next_count, count);
endmodule

module reg8(clk, reset, d, q);
input clk, reset;
input [7:0] d;
output [7:0] q;
reg [7:0] q;

always @(posedge clk or posedge reset)
  if(reset)
    q <= #(`REG_DELAY) 8'h00;
  else
    q <= #(`REG_DELAY) d;
endmodule
```

The counters in Example 22-6 and Example 22-7 are functionally equivalent simple loadable counters. The counter has an added feature: The *match* output from the counter is a combinatorial circuit showing that the current count output matches the load input data.

Example 22-7 Improved Style Counter

```
module count_new(clk, reset, load, ldata, count, match);
input clk, reset, load;
input [7:0] ldata;
output [7:0] count;  reg[7:0] count;
output match;           reg      match;
always @(posedge clk or posedge reset)
  if(reset)
    count <= #(`REG_DELAY) 8'h00;
  else
    if(load)
      count <= #(`REG_DELAY) ldata;
    else
      count <= #(`REG_DELAY) count + 8'h01;
always @ count
  if(count==ldata)
    match = 1'b1;
  else
    match = 1'b0;
endmodule
```

The coding style of Example 22-7 with one statement per line improves readability and makes code coverage easy to observe. Using complex continuous assignments is discouraged for the same reasons. One statement per line is greatly preferred over the style used in Example 22-6. Example 22-6 used complex continuous assignments as for the *match* output and *next_count* signal.

Example 22-8 is a test bench that does an incomplete job of testing the counters. The match output from the counters will never be true. The load input is changed, but not true at a positive edge of the clock. This test bench does not check any of the outputs. It is only used as an example to demonstrate code coverage.

When you run this example, look initially at line coverage. You will see the first counter has 100% line coverage, but that the new style counter never tests *load* or *match*. If you look at an expression coverage report, you can see that the *(count==ldata)* expression is never true for either counter.

Example 22-8 Test bench for Counters

```
`define REG_DELAY 1
`disable_codecoverage
module test_coverage;
reg clk, reset, load;
reg [7:0] ldata;
wire [7:0] count1, count2;
wire match1, match2;
count_old dut1(clk, reset, load, ldata, count1, match1);
count_new dut2(clk, reset, load, ldata, count2, match2);

always begin
  #5 clk = 0;
  #5 clk = 1;
end

initial
  begin
    ldata = 8'h22; load = 1'b0; reset = 1'b1;
    @(posedge clk)
    ldata = 8'h22; load = 1'b0; reset = 1'b0;
    @(posedge clk)
    1data = 8'h22; load = 1'b1; reset = 1'b0;
    @(negedge clk)
    ldata = 8'h22; load = 1'b0; reset = 1'b0;
    repeat(3) @(posedge clk) ;
    $finish;
  end
endmodule
`enable_codecoverage
```

Appendix A GATE-LEVEL DETAILS

Chapters 2 and 3 briefly introduced the built-in primitives. This appendix will briefly describe each of the built-in primitives and the options when instantiating them. The delay and strength options for primitive instances will be explained.

PRIMITIVE DESCRIPTIONS

Logic Gates

AND

Figure A-1 AND Gate

The *and* primitive can have two or more inputs, and has one output. When all of the inputs are "1" then the output is "1".

Table A-1 Logic Table for *and* Primitive

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

NAND

Figure A-2 NAND Gate

The *nand* primitive can have two or more inputs, and has one output. When all of the inputs are "1" then the output is "0".

Table A-2 Logic Table for *nand* Primitive

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

OR

Figure A-3 OR Gate

The *or* primitive can have two or more inputs, and has one output. When any of the inputs is "1" then the output is "1".

Table A-3 Logic Table for *or* Primitive

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

NOR

Figure A-4 NOR Gate

The *nor* primitive can have two or more inputs, and has one output. When any of the inputs is "1" then the output is "0".

Table A-4 Logic Table for *nor* Primitive

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

XOR
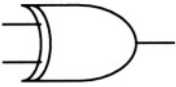
Figure A-5 XOR Gate

The *xor* primitive can have two or more inputs, and has one output. When an odd number of inputs is "1" then the output is "1", unless any input is "x". When any of the inputs is "x" then the output is "x".

Table A-5 Logic Table for *xor* Primitive

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

XNOR
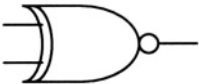
Figure A-6 XNOR Gate

The *xnor* primitive can have two or more inputs, and has one output. When an odd number of inputs is "1" then the output is "0", unless any input is "x". When any of the inputs is "x" then the output is "x".

Table A-6 Logic Table for *xnor* Primitive

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

Buffers

BUF
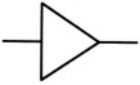
Figure A-7 BUF Gate

The *buf* primitive has one input and can have one or more outputs. The output(s) pass the same value as the input, except an input of "z" produces an output of "x".

Table A-7 Logic Table for *buf* Primitive

Input	Output
0	0
1	1
x	x
z	x

NOT
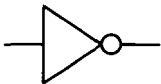
Figure A-8 NOT Gate

The *not* primitive has one input and can have one or more outputs. The output(s) pass the opposite value as the input, except an input of "x" or "z" produces an output of "x".

Table A-8 Logic Table for *not* Primitive

Input	Output
0	0
1	1
x	x
z	x
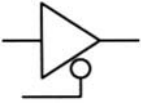
BUFIF0

Figure A-9 BUFIF0 Gate

The *bufif0* primitive has two inputs, (data and control) and one output. When the control input is "0" the output passes the same value as the data input, except that an input of "z" produces an output of "x". When the control input is "1" the output is "z". The port order of the primitive is output, input, control.

The remaining logic tables in the section show both strength and value. Verilog uses a notation of three characters, two for the strength, and one for the value. Table A-20 lists the two character codes used for the strength and the value is *0, 1, x, or z.*

Table A-9 Logic Table for *bufif0* Primitive

Data	Control 0	1	x	z
0	St0	HiZ	StL	StL
1	St1	HiZ	StH	StH
x	StX	HiZ	StX	StX
z	StX	HiZ	StX	StX

BUFIF1
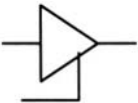
Figure A-10 BUFIF1 Gate

The *bufif1* primitive has two inputs, (data and control) and one output. When the control input is "1" the output passes the same value as the data input, except that an input of "z" produces an output of "x". When the control input is "0" the output is "z". The port order of the primitive is output, input, control.

Table A-10 Logic Table for *bufif1* Primitive

Data	0	1	x	z
0	HiZ	St0	StL	StL
1	HiZ	St1	StH	StH
x	HiZ	StX	StX	StX
z	HiZ	StX	StX	StX

NOTIF0
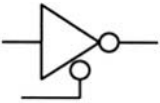
Figure A-11 NOTIF0 Gate

The *notif0* primitive has two inputs, (data and control) and one output. When the control input is "0" the output passes the opposite value as the data input, except that an input of "z" produces an output of "x". When the control input is "1" the output is "z". The port order of the primitive is output, input, control.

Table A-11 Logic Table for *notif0* Primitive

Data	0	1	x	z
0	St1	HiZ	StH	StH
1	St0	HiZ	StL	StL
x	StX	HiZ	StX	StX
z	StX	HiZ	StX	StX

NOTIF1
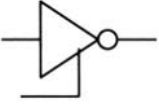
Figure A-12 NOTIF1 Gate

The *notif1* primitive has two inputs, data and control and one output. When the control input is "1" the output passes the opposite value as the data input, except that an input of "z" produces an output of "x". When the control input is "0" the output is "z". The port order of the primitive is output, input, control.

Table A-12 Logic Table for *notif1* Primitive

Data	0	1	x	z
0	HiZ	St1	StH	StH
1	HiZ	St0	StL	StL
x	HiZ	StX	StX	StX
z	HiZ	StX	StX	StX

PULLDOWN

Figure A-13 Pulldown

The *pulldown* primitive has only one terminal. It outputs a *0* of strength pull (*pu0*). When nothing else is driving a net stronger than the *pulldown,* the *pulldown* will drive the net to *0*.

PULLUP

Figure A-14 Pullup

The *pullup* primitive has only one terminal. It outputs a *1* of strength pull (*pul*). When nothing else is driving a net stronger than the *pullup*, the *pullup* will drive the net to *1*.

Switches

Verilog supports simulating transistors as switches. There are switch models of unidirectional and bi-directional switches. There are model of both ideal and resistive switches. Table A-21 details the signal strength reduction through the switches.

NMOS and RNMOS

Figure A-15 NMOS or RNMOS Transistor

The *nmos* and *rnmos* primitives are abstractions of unidirectional switches. When the gate input is 1, this input is passed to the output, otherwise the output is high impedance. The difference between the *nmos* and *rnmos* is the *nmos* passes the input to the output with the strength unchanged, but the *rnmos* passes the value to the output with a decreased strength. The port order of the primitive is output, input, gate.

The output strength from any of the transistor primitives is dependent on the strength of the input. The input strength for the following tables is *strong,* unless otherwise noted.

Table A-13 Logic Table for *nmos* Primitive

Data	0	1	x	z
0	HiZ	St0	StL	StL
1	HiZ	St1	StH	StH
x	HiZ	StX	StX	StX
z	HiZ	HiZ	HiZ	HiZ

Table A-14 Logic Table for *rnmos* Primitive

Data	0	1	x	z
0	HiZ	Pu0	PuL	PuL
1	HiZ	Pu1	PuH	PuH
x	HiZ	PuX	PuX	PuX
z	HiZ	HiZ	HiZ	HiZ

PMOS and RPMOS
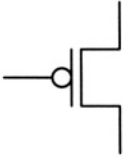
Figure A-16 PMOS or RPMOS Transistor

The *pmos* and *rpmos* primitives are abstractions of unidirectional switches. When the gate input is 0, this input is passed to the output, otherwise the output is high impedance. The difference between the *pmos* and *rpmos* is the *pmos* passes the input to the output with the strength unchanged, but the *rpmos* passes the value to the output with a decreased strength. The port order of the primitive is output, input, gate.

Table A-15 Logic Table for *pmos* Primitive

Data	0	1	x	z
0	St0	HiZ	StL	StL
1	St1	HiZ	StH	StH
x	StX	HiZ	StX	StX
z	HiZ	HiZ	HiZ	HiZ

Table A-16 Logic Table for *rpmos* Primitive

Data	0	1	x	z
0	Pu0	HiZ	PuL	PuL
1	Pu1	HiZ	PuH	PuH
x	PuX	HiZ	PuX	PuX
z	HiZ	HiZ	HiZ	HiZ

CMOS and RCMOS
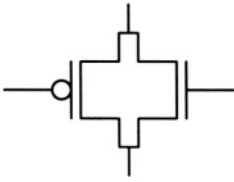
Figure A-17 CMOS or RCMOS transistor

The *cmos* and *rcmos* primitives are the equivalent or *nmos* and *pmos* (or *rnmos* and *rpmos)* primitives connected back to back. The primitive has four terminals. The port order is output, input, ngate, pgate.

Table A-17 Logic Table for *cmos* Primitive

ngate	pgate	0	1	x	z
0	0	St0	St1	StX	HiZ
0	1	HiZ	HiZ	HiZ	HiZ
0	x	StL	StH	StX	HiZ
0	z	StL	StH	StX	HiZ
1	0	St0	St1	StX	HiZ
1	1	St0	St1	StX	HiZ
1	x	St0	St1	StX	HiZ
1	z	St0	St1	StX	HiZ
x	0	St0	St1	StX	HiZ
x	1	StL	StH	StX	HiZ
x	x	StL	StH	StX	HiZ
x	z	StL	StH	StX	HiZ
z	0	St0	St1	StX	HiZ
z	1	StL	StH	StX	HiZ
z	x	StL	StH	StX	HiZ
z	z	StL	StH	StX	HiZ

Table A-18 Logic Table for *rcmos* Primitive

ngate	pgate	0	1	x	z
0	0	Pu0	Pu1	PuX	HiZ
0	1	HiZ	HiZ	HiZ	HiZ
0	x	PuL	PuH	PuX	HiZ
0	z	PuL	PuH	PuX	HiZ
1	0	Pu0	Pu1	PuX	HiZ
1	1	Pu0	Pu1	PuX	HiZ
1	x	Pu0	Pu1	PuX	HiZ
1	z	Pu0	Pu1	PuX	HiZ
x	0	Pu0	Pu1	PuX	HiZ
x	1	PuL	PuH	PuX	HiZ
x	x	PuL	PuH	PuX	HiZ
x	z	PuL	PuH	PuX	HiZ
z	0	Pu0	Pu1	PuX	HiZ
z	1	PuL	PuH	PuX	HiZ
z	x	PuL	PuH	PuX	HiZ
z	z	PuL	PuH	PuX	HiZ

TRAN and RTRAN

The *tran* and *rtran* primitives are bidirectional pass gates. They have only two terminals that are the bidirectional data pins. Delay specifications are not allowed on these primitives. The *rtran* primitive is resistive, and the strength is reduced as the value passes through.

TRANIF0 and RTRANIF0

The *tranif0* and *rtranif0* primitives are bidirectional pass gates. They have three terminals. The first two are the bidirectional data pins, the third is the control input. When the control input is *0,* data passes between the two bidirectional data pins. Delay specifications on these primitives are only turn on and turn off delays. The *rtranif0* primitive is resistive, and the strength is reduced as the value passes through.

TRAN1F1 and RTRANIF1

The *tranif1* and *rtranif1* primitives are bi-directional pass gates. They have three terminals. The first two are the bi-directional data pins, the third is the control input. When the control input is *1,* data passes between the two bi-directional data pins. Delay specifications on these primitives are only turn on and turn off delays. The *rtranif1* primitive is resistive, and the strength is reduced as the value passes through.

INSTANCE DETAILS

When you create an instance of one of the built-in primitives, the instance name is optional as you learned in chapter 3. There are other optional arguments you may specify when creating primitive instances: delays and strengths.

Delays

A primitive may have from zero to nine delay specifications. If a primitive instance has no delay specification the primitive is zero delay. A primitive may have separate delays for both rise (to *1*) and fall (to *0*) time. A primitive that can output high impedance (such as *bufif1, bufif0,* and the switches) has an optional turn off (to *z*) delay. The delay to *x* is always the least of the delays specified. If a single delay is specified, it is used for both rise and fall (and turn off if applicable).

Delays are declared with the # delay operator. The order of the delays is rise, fall, and turnoff.

Example A-1 Delays in Primitive Instances

```
module delays;
and a1 (y, a, b), a2 (w, d, e, f) ; // zero delay and gates
and #1 a3 (x, a, c, e), a4 (z, e, f); // unit delay and gates
and #(3,2) a5(c, w, z); // rise delay of 3, fall delay of 2
bufif1 #(3,2,4) b1(x, a,b); // turn off time is 4
bufif0 #(2:3:4,1:2:3,3:4:5) b2 (f, e, d) ; // min:typ:max
endmodule
```

Example A-1 shows primitive instances with between zero and nine delays specified. Notice that similar to *reg* and *wire* declarations where the range in the specification applies to each declaration, the delay(s) affect each of the instances in the declaration. The instances *a3* and *a4* are both unit delay, and that delay of 1 is used for both rise and fall time. Instance *a5* illustrates specifying separate rise and

fall delays. When multiple delays are specified they are enclosed in parentheses. Instance b1 shows adding the third delay specification of turn off time.

Each of the delay specifications can be expanded to be a minimum, typical, and maximum delay specification. When the min, typical, and max delays are specified they are separated by colons as shown with instance *b2*.

Delay Units

All the delays throughout the book were unitless. In behavioral modeling a delay of one could mean anything. In the phone example in chapter 1 a delay unit was interpreted as one minute. In gate-level modeling you might want to use delays with units such as 1 nanosecond or 1 picosecond.

Verilog allows both the specification of a delay unit and a delay precision on a per-module basis. The delay unit and precision are declared with the `timescale compiler directive.

The main drawback to using delay units is that if you use them on one module you must use them on all modules. When you declare a delay unit and precision, that declaration applies to all following modules. You can declare the unit and precision for just one module and have that declaration carry forward to all your other modules: Just make sure that the file with the delay unit and precision declaration is the first one compiled. If you compile a module before a `timescale` directive and a `timescale` directive appears later in the source, Verilog will issue an error message and abort compiling your files.

Table A-19 Delay and Precision Units

Code	Definition
fs	femtoseconds
ps	picoseconds
ns	nanoseconds
us	microseconds
ms	milliseconds
s	seconds

Table A-19 lists the units that can be used in a timescale declaration. The timescale declaration goes outside a module, just before the module. The delay unit and precision are a combination of 1, 10, or 100, and one of the unit codes in Table A-19.

Example A-2 Time Scales

```
`timescale 1ns / 100ps
module mod1;
   // #1.1 in this module = 1.1 ns
endmodule

`timescale 100ps / 1 ps
module mod2;
   // #2 in this module = 200 ps
endmodule
```

Example A-2 shows two modules with their timescales. With the addition of the time unit and precision, decimal delays are possible. Timescales and decimal delays are usable for both behavioral and gate-level modeling, but most delays in behavioral modeling are unitless. Behavioral modeling tends to be more abstract and delays less important than delays in gate-level models.

Printing Out Time and the Timescale

The format code *%t* will print out the value of *$time* or *$realtime* including the units.

Strengths

As mentioned in chapter 2, Verilog supports seven levels of strength. A behavioral statement such as a register or continuous assignment always drives with a strength of strong. By default all gate instances drive with strength strong, with the exception of the *pullup, pulldown,* and resistive transistors which are pull, or are decreased by one strength level.

A primitive instance may optionally include the drive strengths from the *drive0* and *drive1* columns of Table A-20 to indicate the drive strength of the gate.

Table A-20 strengths

value	%v	drive0	drive1	Description
0	Hi	highz0	highz1	High impedance
1	Sm	-	-	Small capacitor
2	Me	-	-	Medium capacitor
3	We	weak0	weak1	Weak drive
4	La	-	-	Large capacitor
5	Pu	pull0	pull1	Pull drive
6	St	strong0	strong1	Strong drive
7	Su	supply0	supply1	Supply drive

The drive strength of an instance is specified before the delay and the order of strength declarations may be either *strength1* or *strength0* first. Example A-3 shows declarations including strengths and delays.

Example A-3 Strength Declarations

```
module strength;
nand (strong0, highz1) oc1(z, a, b) oc2(w, d, e);
buf (weak1, weak0) #6 wimpo(out, in);
myudp (pull0, pull1) #(2:3:4,1:2:3) u1(q, c, d, r, s);
endmodule
```

Instances *oc1* and *oc2* in Example A-3 could be an example of a TTL open collector NAND gate such as the 7401. As you may know, open collector drivers have strong drive low, but no drive high, so they need a pullup resistor to achieve a value of 1. The example buffer shows combining the syntax for specifying strengths with the syntax of specifying a delay.

The final instance, *u1*, is an example of a user-defined primitive. The user-defined primitive *myudp* is assumed to be declared elsewhere. UDP instances also allow the specification of both strengths and delays.

Displaying Strengths with *%v*

The format code %v provides more information than %b. The output produced by %v is three characters, which describe in more detail the value and strength of a net. The first two characters describe the strength as described in Table A-20. The third character describes the value. Two new value codes are added to the *0, 1, X, Z* set. The two new codes are *H* and *L*. The value *H* represents either a *1* or a *Z*. The value *L* represents either a *0* or a *Z*.

Nets in Verilog may be one of 120 possible values. All of these values can be printed out with %v. Some of the values do not print out as nicely as the codes from Table A-20. These values print out the strength portion of the value with a two-digit numeric code. Values such as *65X* and *241* can be printed out by *%v*. If the third character in the value is *X,* the first number represents the zero strength and the second number represents the one strength. If the third character in the value is *0* or *1,* then the first two numbers represent a possible range of strength for the value.

Strength Reduction of Switch Primitives

Table A-21 shows the output strength for the switch primitives. The *nmos, pmos, cmos, tran, tranif0,* and *tranif1* primitives are considered to be ideal devices since the strength is not reduced through them, except that supply strength is reduced to strong. The *rnmos, rpmos, rcmos, rtran, rtranif0,* and *rtranif1* primitives are considered to be resistive devices since the strength is reduced through them.

Table A-21 Switch Strength Reduction

Input Strength	Ideal Device Output Strength	Resistive Device Output Strength
supply	strong	pull
strong	strong	pull
pull	pull	weak
weak	weak	medium
large	large	medium
medium	medium	small
small	small	small
high impedance	high impedance	high impedance

All this detail with strengths is important for switch-level modeling, but is not used in behavioral modeling.

INDEX

T

U