to index of chapters

CHAPTER 10
VERILOG HDL

In this chapter we look at the **Verilog** hardware description language. Gateway Design Automation developed Verilog as a simulation language. The use of the Verilog-XL simulator is discussed in more detail in Chapter 13. Cadence purchased Gateway in 1989 and, after some study, placed the Verilog language in the public domain. Open Verilog International (OVI) was created to develop the Verilog language as an IEEE standard. The definitive reference guide to the Verilog language is now the Verilog LRM, IEEE Std 1364-1995 [1995]. 1 This does not mean that all Verilog simulators and tools adhere strictly to the IEEE Standard--we must abide by the reference manual for the software we are using. Verilog is a fairly simple language to learn, especially if you are familiar with the C programming language. In this chapter we shall concentrate on the features of Verilog applied to high-level design entry and synthesis for ASICs.

---

1. Some of the material in this chapter is reprinted with permission from IEEE Std

Chapter start

Previous page

Next page

11.1 A Counter

The following Verilog code models a "black box" that contains a 50 MHz clock (period 20 ns), counts from 0 to 7, resets, and then begins counting at 0 again:

```
‘timescale 1ns/1ns // Set the units of time to be nanoseconds.
module counter;
  reg clock; // Declare a reg data type for the clock.
  integer count; // Declare an integer data type for the count.
initial // Initialize things; this executes once at t=0.
  begin
    clock = 0; count = 0; // Initialize signals.
    #340 $finish; // Finish after 340 time ticks.
  end
/* An always statement to generate the clock; only one statement follows the always
always  #10 clock = ~ clock; // Delay (10ns) is set to half the clock cycle.
/* An always statement to do the counting; this executes at the same time (concurren
always   begin    // Wait here until the clock goes from 1 to 0.
    @ (negedge clock);
    // Now handle the counting.
    if (count == 7)
      count = 0;
    else      count = count + 1;
    $display("time = ",$time," count = ", count);
  end
endmodule
```

Verilog **keywords** (reserved words that are part of the Verilog language) are shown in bold type in the code listings (but not in the text). References in this chapter such as [Verilog LRM 1.1] refer you to the IEEE Verilog LRM.

The following output is from the Cadence Verilog-XL simulator. This example includes the system input so you can see how the tool is run and when it is finished. Some of the banner information is omitted in the listing that follows to save space (we can use "quiet" mode using a `'-q'` flag, but then the version and other useful information is also suppressed):

```
> verilog counter.v
VERILOG-XL 2.2.1   Apr 17, 1996  11:48:18
```

```
    ... Banner information omitted here...
Compiling source file "counter.v"
Highest level modules:
counter
time =                   20 count =           1
time =                   40 count =           2
(... 12 lines omitted...)
time =                  300 count =           7
time =                  320 count =           0
L10 "counter.v": $finish at simulation time 340
223 simulation events
CPU time: 0.6 secs to compile + 0.2 secs to link + 0.0 secs in simulation
End of VERILOG-XL 2.2.1   Apr 17, 1996  11:48:20
>
```

Here is the output of the VeriWell simulator from the console window (future examples do not show all of the compiler output-- just the model output):

```
Veriwell -k VeriWell.key -l VeriWell.log -s :counter.v
... banner information omitted ....
Memory Available: 0
Entering Phase I...
Compiling source file : :counter.v
The size of this model is [1%, 1%] of the capacity of the free version
Entering Phase II...
Entering Phase III...
No errors in compilation
Top-level modules:
   counter
C1> .
time =                   20 count =           1
time =                   40 count =           2
(... 12 lines omitted...)
time =                  300 count =           7
time =                  320 count =           0
Exiting VeriWell for Macintosh at time 340
0 Errors, 0 Warnings, Memory Used: 29468
Compile time = 0.6, Load time = 0.7, Simulation time = 4.7
Normal exit
Thank you for using VeriWell for Macintosh
```

Chapter start

Previous page

Chapter start

Previous page

Next page

11.2 Basics of the Verilog Language

A Verilog **identifier** [Verilog LRM2.7], including the names of variables, may contain any sequence of letters, digits, a dollar sign '$' , and the underscore '_' symbol. The first character of an identifier must be a letter or underscore; it cannot be a dollar sign '$' , for example. We cannot use characters such as '-' (hyphen), brackets, or '#' (for active-low signals) in Verilog names (escaped identifiers are an exception). The following is a shorthand way of saying the same thing:

```
identifier ::= simple_identifier | escaped_identifier
simple_identifier ::= [a-zA-Z][a-zA-Z_$]
escaped_identifier ::=
  \ {Any_ASCII_character_except_white_space} white_space
white_space ::= space | tab | newline
```

If we think of '::=' as an equal sign, then the preceding "equation" defines the syntax of an identifier. Usually we use the Backus-Naur form (BNF) to write these equations. We also use the BNF to describe the syntax of VHDL. There is an explanation of the BNF in Appendix A. Verilog syntax definitions are given in Appendix B. In Verilog all names, including keywords and identifiers, are case-sensitive. Special commands for the simulator (a system task or a system function) begin with a dollar sign '$' [Verilog LRM 2.7]. Here are some examples of Verilog identifiers:

```
module identifiers;
/* Multiline comments in Verilog
  look like C comments and // is OK in here. */
// Single-line comment in Verilog.
reg legal_identifier,two__underscores;
reg _OK,OK_,OK_$,OK_123,CASE_SENSITIVE, case_sensitive;
reg \/clock ,\a*b ; // Add white_space after escaped identifier.
//reg $_BAD,123_BAD; // Bad names even if we declare them!
initial begin
legal_identifier = 0; // Embedded underscores are OK,
two__underscores = 0; // even two underscores in a row.
_OK = 0; // Identifiers can start with underscore
OK_ = 0; // and end with underscore.
OK$ = 0; // $ sign is OK, but beware foreign keyboards.
OK_123 =0; // Embedded digits are OK.
CASE_SENSITIVE = 0; // Verilog is case-sensitive (unlike VHDL).
case_sensitive = 1;
\/clock = 0; // An escaped identifier with \ breaks rules,
\a*b = 0; // but be careful to watch the spaces!
$display("Variable CASE_SENSITIVE= %d",CASE_SENSITIVE);
$display("Variable case_sensitive= %d",case_sensitive);
$display("Variable \/clock = %d",\/clock );
$display("Variable \\a*b = %d",\a*b );
end endmodule
```

The following is the output from this model (future examples in this chapter list the simulator output directly after the Verilog code).

```
Variable CASE_SENSITIVE= 0
Variable case_sensitive= 1
Variable /clock = 0
Variable \a*b = 0
```

11.2.1 Verilog Logic Values

Verilog has a predefined **logic-value system or value set** [Verilog LRM 3.1] that uses four logic values: `'0'`, `'1'`, `'x'`, and `'z'` (lowercase `'x'` and lowercase `'z'`). The value `'x'` represents an uninitialized or an unknown logic value--an unknown value is either `'1'`, `'0'`, `'z'`, or a value that is in a state of change. The logic value `'z'` represents a high-impedance value, which is usually treated as an `'x'` value. Verilog uses a more complicated internal logic-value system in order to resolve conflicts between different drivers on the same node. This hidden logic-value system is useful for switch-level simulation, but for most ASIC simulation and synthesis purposes we do not need to worry about the internal logic-value system.

11.2.2 Verilog Data Types

There are several **data types** in Verilog--all except one need to be declared before we can use them. The two main data types are **nets** and **registers** [Verilog LRM 3.2]. Nets are further divided into several net types. The most common and important net types are: **wire** and **tri** (which are identical); **supply1** and **supply0** (which are equivalent to the positive and negative power supplies respectively). The `wire` data type (which we shall refer to as just `wire` from now on) is analogous to a wire in an ASIC. A `wire` cannot store or hold a value. A `wire` must be continuously driven by an assignment statement (see Section 11.5). The default initial value for a `wire` is `'z'` [Verilog LRM3.6]. There are also **integer**, **time**, **event**, and **real** data types.

```
module declarations_1;
wire pwr_good, pwr_on, pwr_stable; // Explicitly declare wires.
integer i; // 32-bit, signed (2's complement).
time t; // 64-bit, unsigned, behaves like a 64-bit reg.
event e; // Declare an event data type.
real r; // Real data type of implementation defined size.
// An assign statement continuously drives a wire:
assign pwr_stable = 1'b1; assign pwr_on = 1; // 1 or 1'b1
assign pwr_good = pwr_on & pwr_stable;
initial begin
i = 123.456; // There must be a digit on either side
r = 123456e-3; // of the decimal point if it is present.
t = 123456e-3; // Time is rounded to 1 second by default.
$display("i=%0g",i," t=%6.2f",t," r=%f",r);
#2 $display("TIME=%0d",$time," ON=",pwr_on,
  " STABLE=",pwr_stable," GOOD=",pwr_good);
$finish; end
endmodule
i=123 t=123.00 r=123.456000
TIME=2 ON=1 STABLE=1 GOOD=1
```

A register data type is declared using the keyword `reg` and is comparable to a variable in a programming language. On the LHS of an assignment a register data type (which we shall refer to as just `reg` from now on) is updated immediately and holds its value until changed again. The default initial value for a `reg` is `'x'`. We can transfer information directly from a `wire` to a `reg` as shown in the following code:

```
module declarations_2;
reg Q, Clk; wire D;
```

```
// Drive the wire (D):
assign D = 1;
// At a +ve clock edge assign the value of wire D to the reg Q:
always @(posedge Clk) Q = D;
initial Clk = 0; always #10 Clk = ~ Clk;
initial begin #50; $finish; end
always begin
$display("T=%2g", $time," D=",D," Clk=",Clk," Q=",Q); #10; end
endmodule
T= 0 D=z Clk=0 Q=x
T=10 D=1 Clk=1 Q=x
T=20 D=1 Clk=0 Q=1
T=30 D=1 Clk=1 Q=1
T=40 D=1 Clk=0 Q=1
```

We shall discuss assignment statements in Section 11.5. For now, it is important to recognize that a `reg` is not always equivalent to a hardware register, flip-flop, or latch. For example, the following code describes purely combinational logic:

```
module declarations_3;
reg a,b,c,d,e;
initial begin
  #10; a = 0;b = 0;c = 0;d = 0; #10; a = 0;b = 1;c = 1;d = 0;
  #10; a = 0;b = 0;c = 1;d = 1; #10; $stop;
end
always begin
  @(a or b or c or d) e = (a|b)&(c|d);
  $display("T=%0g",$time," e=",e);
end
endmodule
T=10 e=0
T=20 e=1
T=30 e=0
```

A single-bit `wire` or `reg` is a **scalar** (the default). We may also declare a `wire` or `reg` as a **vector** with a **range** of bits [Verilog LRM 3.3]. In some situations we may use implicit declaration for a scalar `wire` ; it is the only data type we do not always need to declare. We must use explicit declaration for a vector `wire` or any `reg` . We may access (or **expand**) the range of bits in a vector one at a time, using a **bit-select**, or as a contiguous subgroup of bits (a continuous sequence of numbers--like a straight in poker) using a **part-select** [Verilog LRM 4.2]. The following code shows some examples:

```
module declarations_4;
wire Data; // A scalar net of type wire.
wire [31:0] ABus, DBus; // Two 32-bit-wide vector wires:
// DBus[31] = leftmost = most-significant bit  = msb
// DBus[0] = rightmost = least-significant bit = lsb
// Notice the size declaration precedes the names.
// wire [31:0] TheBus, [15:0] BigBus; // This is illegal.
reg [3:0] vector; // A 4-bit vector register.
reg [4:7] nibble; // msb index < lsb index is OK.
integer i;
initial begin
i = 1;
```

```
vector = 'b1010; // Vector without an index.
nibble = vector; // This is OK too.
#1; $display("T=%0g",$time," vector=", vector," nibble=", nibble);
#2; $display("T=%0g",$time," Bus=%b",DBus[15:0]);
end
assign DBus [1] = 1; // This is a bit-select.
assign DBus [3:0] = 'b1111; // This is a part-select.
// assign DBus [0:3] = 'b1111; // Illegal : wrong direction.
endmodule
T=1 vector=10 nibble=10
T=3 Bus=zzzzzzzzzzzz1111
```

There are no multidimensional arrays in Verilog, but we may declare a **memory** data type as an **array** of registers [Verilog LRM 3.8]:

```
module declarations_5;
reg [31:0] VideoRam [7:0]; // An 8-word by 32-bit wide memory.
initial begin
VideoRam[1] = 'bxz; // We must specify an index for a memory.
VideoRam[2] = 1;
VideoRam[7] = VideoRam[VideoRam[2]]; // Need 2 clock cycles for this.
VideoRam[8] = 1; // Careful! the compiler won't complain about this!
// Verify what we entered:
$display("VideoRam[0] is %b",VideoRam[0]);
$display("VideoRam[1] is %b",VideoRam[1]);
$display("VideoRam[2] is %b",VideoRam[2]);
$display("VideoRam[7] is %b",VideoRam[7]);
end
endmodule
VideoRam[0] is xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
VideoRam[1] is xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxz
VideoRam[2] is 00000000000000000000000000000001
VideoRam[7] is xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxz
```

We may also declare an **integer array** or **time array** in the same way as an array of `reg` , but there are no real arrays [Verilog LRM 3.9]:

```
module declarations_6;
integer Number [1:100]; // Notice that size follows name
time Time_Log [1:1000]; // - as in an array of reg.
// real Illegal [1:10]; // Illegal. There are no real arrays.
endmodule
```

11.2.3 Other Wire Types

There are the following other Verilog wire types (rarely used in ASIC design) [Verilog LRM 3.7]:

- `wand` , `wor` , `triand` , and `trior` model wired logic. Wiring, or dotting, the outputs of two gates generates a logic function (in emitter-coupled logic, ECL, or in an EPROM, for example). This is one area in which the logic values `'z'` and `'x'` are treated differently.
- `tri0` and `tri1` model resistive connections to VSS or VDD.

- `trireg` is like a `wire` but associates some capacitance with the net, so it can model charge storage.

There are also other keywords that may appear in declarations:

- `scalared` and `vectored` are properties of vectors [Verilog LRM 3.3].
- `small` , `medium` , and `large` model the charge strength of `trireg` connections [Verilog LRM 7].

11.2.4 Numbers

Constant numbers are integer or real constants [Verilog LRM 2.5]. **Integer constants** are written as

```
width'radix value
```

where `width` and `radix` are optional. The **radix** (or base) indicates the type of number: **decimal** (`d` or `D`), **hex** (`h` or `H`), **octal** (`o` or `O`), or **binary** (`b` or `B`). A number may be **sized** or **unsized**. The length of an unsized number is implementation dependent. We can use `'1'` and `'0'` as numbers since they cannot be identifiers, but we must write `1'bx` and `1'bz` for `'x'` and `'z'` . A number may be declared as a **parameter** [Verilog LRM 3.10]. A parameter assignment belongs inside a module declaration and has **local scope** [Verilog LRM3.11]. **Real constants** are written using decimal (100.0) or scientific notation (1e2) and follow IEEE Std 754-1985 for double-precision floating-point numbers. Reals are rounded to the nearest integer, ties (numbers that end in .5) round away from zero [Verilog LRM 3.9], but not all implementations follow this rule (the output from the following code is from VeriWell, which rounds ties toward zero for negative integers).

```
module constants;
parameter H12_UNSIZED = 'h 12; // Unsized hex 12 = decimal 18.
parameter H12_SIZED = 6'h 12; // Sized hex 12 = decimal 18.
// Note: a space between base and value is OK.
// Note: '' (single apostrophes) are not the same as the ' character.
parameter D42 = 8'B0010_1010; // bin 101010 = dec 42
// OK to use underscores to increase readability.
parameter D123 = 123; // Unsized decimal (the default).
parameter D63 = 8'o 77; // Sized octal, decimal 63.
// parameter ILLEGAL = 1'o9; // No 9's in octal numbers!
// A = 'hx and B = 'ox assume a 32 bit width.
parameter A = 'h x, B = 'o x, C = 8'b x, D = 'h z, E = 16'h ????;
// Note the use of ? instead of z, 16'h ???? is the same as 16'h zzzz.
// Also note the automatic extension to a width of 16 bits.
reg [3:0] B0011,Bxxx1,Bzzz1; real R1,R2,R3; integer I1,I3,I_3;
parameter BXZ = 8'b1x0x1z0z;
initial begin
B0011 = 4'b11; Bxxx1 = 4'bx1; Bzzz1 = 4'bz1; // Left padded.
R1 = 0.1e1; R2 = 2.0; R3 = 30E-01; // Real numbers.
I1 = 1.1; I3 = 2.5; I_3 = -2.5; // IEEE rounds away from 0.
end initial begin #1;
$display
("H12_UNSIZED, H12_SIZED (hex) = %h, %h",H12_UNSIZED, H12_SIZED);
$display("D42 (bin) = %b",D42," (dec) = %d",D42);
$display("D123 (hex) = %h",D123," (dec) = %d",D123);
$display("D63 (oct) = %o",D63);
$display("A (hex) = %h",A," B (hex) = %h",B);
```

```
$display("C (hex) = %h",C," D (hex) = %h",D," E (hex) = %h",E);
$display("BXZ (bin) = %b",BXZ," (hex) = %h",BXZ);
$display("B0011, Bxxx1, Bzzz1 (bin) = %b, %b, %b",B0011,Bxxx1,Bzzz1);
$display("R1, R2, R3 (e, f, g) = %e, %f, %g", R1, R2, R3);
$display("I1, I3, I_3 (d) = %d, %d, %d", I1, I3, I_3);
end
endmodule
H12_UNSIZED, H12_SIZED (hex) = 00000012, 12
D42 (bin) = 00101010 (dec) =  42
D123 (hex) = 0000007b (dec) =         123
D63 (oct) = 077
A (hex) = xxxxxxxx B (hex) = xxxxxxxx
C (hex) = xx D (hex) = zzzzzzzz E (hex) = zzzz
BXZ (bin) = 1x0x1z0z (hex) = XZ
B0011, Bxxx1, Bzzz1 (bin) = 0011, xxx1, zzz1
R1, R2, R3 (e, f, g) = 1.000000e+00, 2.000000, 3
I1, I3, I_3 (d) =           1,           3,          -2
```

11.2.5 Negative Numbers

Integer numbers are **signed** (two's complement) or **unsigned**. The following example illustrates the handling of negative constants [Verilog LRM 3.2 , 4.1]:

```
module negative_numbers;
parameter PA = -12, PB = -'d12, PC = -32'd12, PD = -4'd12;
integer IA , IB , IC , ID ; reg [31:0] RA , RB , RC , RD ;
initial begin #1;
IA = -12; IB = -'d12; IC = -32'd12; ID = -4'd12;
RA = -12; RB = -'d12; RC = -32'd12; RD = -4'd12; #1;
$display("     parameter    integer   reg[31:0]");
$display ("-12     =",PA,IA,,,RA);
$displayh("          ",,,,PA,,,,IA,,,,,RA);
$display ("-'d12   =",,PB,IB,,,RB);
$displayh("          ",,,,PB,,,,IB,,,,,RB);
$display ("-32'd12 =",,PC,IC,,,RC);
$displayh("          ",,,,PC,,,,IC,,,,,RC);
$display ("-4'd12  =",,,,,,,,,,PD,ID,,,RD);
$displayh("          ",,,,,,,,,,,PD,,,,ID,,,,,RD);
end
endmodule
           parameter    integer   reg[31:0]
-12     =         -12         -12  4294967284
            fffffff4   fffffff4    fffffff4
-'d12   = 4294967284         -12  4294967284
            fffffff4   fffffff4    fffffff4
-32'd12 = 4294967284         -12  4294967284
            fffffff4   fffffff4    fffffff4
-4'd12  =          4         -12  4294967284
                   4   fffffff4    fffffff4
```

Verilog only "keeps track" of the sign of a negative constant if it is (1) assigned to an `integer` or (2) assigned to a `parameter` without using a base (essentially the same thing). In other cases (even though the bit representations may be identical to the signed number--hexadecimal `fffffff4` in the previous example), a negative constant is treated as an unsigned number. Once Verilog "loses" the sign, keeping track of signed numbers becomes your responsibility (see also Section 11.3.1).

11.2.6 Strings

The code listings in this book use `Courier` font. The ISO/ANSI standard for the ASCII code defines the characters, but not the appearance of the graphic symbol in any particular font. The confusing characters are the quote and accent characters:

```
module characters; /*
" is ASCII 34 (hex 22), double quote.
’ is ASCII 39 (hex 27), tick or apostrophe.
/ is ASCII 47 (hex 2F), forward slash.
\ is ASCII 92 (hex 5C), back slash.
‘ is ASCII 96 (hex 60), accent grave.
| is ASCII 124 (hex 7C), vertical bar.
There are no standards for the graphic symbols for codes above 128.
´ is 171 (hex AB), accent acute in almost all fonts.
" is 210 (hex D2), open  double quote, like 66 (in some fonts).
" is 211 (hex D3), close double quote, like 99 (in some fonts).
‘ is 212 (hex D4), open  single quote, like  6 (in some fonts).
’ is 213 (hex D5), close single quote, like  9 (in some fonts).
*/ endmodule
```

Here is an example showing the use of **string constants** [Verilog LRM 2.6]:

```
module text;
parameter A_String = "abc"; // string constant, must be on one line
parameter Say = "Say \"Hey!\"";
// use escape quote \" for an embedded quote
parameter Tab = "\t"; // tab character
parameter NewLine = "\n"; // newline character
parameter BackSlash = "\\"; // back slash
parameter Tick = "\047"; // ASCII code for tick in octal
// parameter Illegal = "\500"; // illegal - no such ASCII code
initial begin$display("A_String(str) = %s ",A_String," (hex) = %h ",A_String);
$display("Say = %s ",Say," Say \"Hey!\"");
$display("NewLine(str) = %s ",NewLine," (hex) = %h ",NewLine);
$display("\\(str) = %s ",BackSlash," (hex) = %h ",BackSlash);
$display("Tab(str) = %s ",Tab," (hex) = %h ",Tab,"1 newline...");
$display("\n");
$display("Tick(str) = %s ",Tick," (hex) = %h ",Tick);
#1.23; $display("Time is %t", $time);
end
endmodule
A_String(str) = abc  (hex) = 616263
Say = Say \"Hey!\"  Say "Hey!"
NewLine(str) = \n  (hex) = 0a
\(str) = \\  (hex) = 5c
Tab(str) = \t  (hex) = 09 1 newline...


Tick(str) = ’  (hex) = 27
Time is                    1
```

Instead of parameters you may use a **define directive** that is a **compiler directive**, and not a statement [Verilog LRM 16]. The `define` directive has **global scope**:

```
module define;
define G_BUSWIDTH 32 // Bus width parameter (G_ for global).
/* Note: there is no semicolon at end of a compiler directive. The character ‘ is AS
wire [‘G_BUSWIDTH:0]MyBus; // A 32-bit bus.
endmodule
```

Chapter start

Previous page

Chapter start

Previous page

Next page

11.3 Operators

An expression uses any of the three types of operators: unary operators, binary operators, and a single ternary operator [Verilog LRM 4.1]. The Verilog operators are similar to those in the C programming language--except there is no autoincrement (`++`) or autodecrement (`--`) in Verilog. Table 11.1 shows the operators in their (increasing) order of precedence and Table 11.2 shows the unary operators. Here is an example that illustrates the use of the Verilog operators:

TABLE 11.1 Verilog operators (in increasing order of precedence).
?: (conditional) [legal for real; associates right to left (others associate left to right)]
|| (logical or) [A smaller operand is zero-filled from its msb (0-fill); legal for real]
&& (logical and)[0-fill, legal for real]
| (bitwise or) ~| (bitwise nor) [0-fill]
^ (bitwise xor) ^~ ~^ (bitwise xnor, equivalence) [0-fill]
& (bitwise and) ~& (bitwise nand) [0-fill]
== (logical) != (logical) === (case) !== (case) [0-fill, logical versions are legal for real]
< (lt) <= (lt or equal) > (gt) >= (gt or equal) [0-fill, all arelegal for real]
<< (shift left) >> (shift right) [zero fill; no -ve shifts; shift by x or z results in unknown]
+ (addition) - (subtraction) [if any bit is x or z for + - * / % then entire result is unknown]
* (multiply) / (divide) % (modulus) [integer divide truncates fraction; + - * / legal for real]
Unary operators: ! ~ & ~& | ~| ^ ~^ ^~ + - [see Table 11.2 for precedence]

TABLE 11.2 Verilog unary operators.		
Operator	**Name**	**Examples**
!	logical negation	!123 is ’b0 [0, 1, or x for ambiguous; legal for real]
~	bitwise unary negation	~1’b10xz is 1’b01xx
&	unary reduction and	& 4’b1111 is 1’b1, & 2’bx1 is 1’bx, & 2’bz1 is 1’bx
~&	unary reduction nand	~& 4’b1111 is 1’b0, ~& 2’bx1 is 1’bx
|	unary reduction or	Note:
~|	unary reduction nor	Reduction is performed left (first bit) to right
^	unary reduction xor	Beware of the non-associative reduction operators
~^ ^~	unary reduction xnor	z is treated as x for all unary operators
+	unary plus	+2’bxz is +2’bxz [+m is the same as m; legal for real]
-	unary minus	-2’bxz is x [-m is unary minus m; legal for real]

```
module operators;
parameter A10xz = {1’b1,1’b0,1’bx,1’bz}; // Concatenation and
parameter A01010101 = {4{2’b01}}; // replication, illegal for real.
// Arithmetic operators: +, -, *, /, and modulus %
parameter A1 = (3+2) %2; // The sign of a % b is the same as sign of a.
// Logical shift operators: << (left), >> (right)
parameter A2 = 4 >> 1; parameter A4 = 1 << 2; // Note: zero fill.
// Relational operators: <, <=, >, >=
initial if (1 > 2) $stop;
// Logical operators: ! (negation), && (and), || (or)
parameter B0 = !12; parameter B1 = 1 && 2;
reg [2:0] A00x; initial begin A00x = ’b111; A00x = !2’bx1; end
parameter C1 = 1 || (1/0); /* This may or may not cause an
error: the short-circuit behavior of && and || is undefined. An
```

```
evaluation including && or || may stop when an expression is known
to be true or false. */
// == (logical equality), != (logical inequality)
parameter Ax = (1==1'bx); parameter Bx = (1'bx!=1'bz);
parameter D0 = (1==0); parameter D1 = (1==1);
// === case equality, !== (case inequality)
// The case operators only return true (1) or false (0).
parameter E0 = (1===1'bx); parameter E1 = 4'b01xz === 4'b01xz;
parameter F1 = (4'bxxxx === 4'bxxxx);
// Bitwise logical operators:
// ~ (negation), & (and), | (inclusive or),
// ^ (exclusive or), ~^ or ^~ (equivalence)
parameter A00 = 2'b01 & 2'b10;
// Unary logical reduction operators:
// & (and), ~& (nand), | (or), ~| (nor),
// ^ (xor), ~^ or ^~ (xnor)
parameter G1= & 4'b1111;
// Conditional expression f = a ? b : c [if (a) then f=b else f=c]
// if a=(x or z), then (bitwise) f=0 if b=c=0, f=1 if b=c=1, else f=x
reg H0, a, b, c; initial begin a=1; b=0; c=1; H0=a?b:c; end
reg[2:0] J01x, Jxxx, J01z, J011;
initial begin Jxxx = 3'bxxx; J01z = 3'b01z; J011 = 3'b011;
J01x = Jxxx ? J01z : J011; end // A bitwise result.
initial begin #1;
$display("A10xz=%b",A10xz,"  A01010101=%b",A01010101);
$display("A1=%0d",A1,"  A2=%0d",A2,"  A4=%0d",A4);
$display("B1=%b",B1,"  B0=%b",B0,"  A00x=%b",A00x);
$display("C1=%b",C1,"  Ax=%b",Ax,"  Bx=%b",Bx);
$display("D0=%b",D0,"  D1=%b",D1);
$display("E0=%b",E0,"  E1=%b",E1,"  F1=%b",F1);
$display("A00=%b",A00,"  G1=%b",G1,"  H0=%b",H0);
$display("J01x=%b",J01x); end
endmodule
A10xz=10xz  A01010101=01010101
A1=1  A2=2  A4=4
B1=1  B0=0  A00x=00x
C1=1  Ax=x  Bx=x
D0=0  D1=1
E0=0  E1=1  F1=1
A00=00  G1=1  H0=0
J01x=01x
```

11.3.1 Arithmetic

Arithmetic operations on n-bit objects are performed modulo 2^n in Verilog,

```
module modulo; reg [2:0] Seven;
initial begin
#1 Seven = 7; #1 $display("Before=", Seven);
#1 Seven = Seven + 1; #1 $display("After =", Seven);
end
endmodule
Before=7
After =0
```

Arithmetic operations in Verilog (addition, subtraction, comparison, and so on) on vectors (`reg` or `wire`) are predefined (Tables 11.1 and 11.2 show which operators are legal for `real`). This is a very important difference for ASIC designers from the situation in VHDL. However, there are some

subtleties with Verilog arithmetic and negative numbers that are illustrated by the following example (based on an example in the LRM [Verilog LRM4.1]):

```
module LRM_arithmetic;
integer IA, IB, IC, ID, IE; reg [15:0] RA, RB, RC;
initial begin
IA = -4'd12;     RA =  IA / 3; // reg is treated as unsigned.
RB = -4'd12;     IB =  RB / 3; //
IC = -4'd12 / 3; RC = -12 / 3; // real is treated as signed
ID =    -12 / 3; IE =  IA / 3; // (two's complement).
end
initial begin #1;
$display("                        hex    default");
$display("IA = -4'd12     = %h%d",IA,IA);
$display("RA = IA / 3     =     %h      %d",RA,RA);
$display("RB = -4'd12     =     %h      %d",RB,RB);
$display("IB = RB / 3     = %h%d",IB,IB);
$display("IC = -4'd12 / 3 = %h%d",IC,IC);
$display("RC = -12 / 3    =     %h      %d",RC,RC);
$display("ID = -12 / 3    = %h%d",ID,ID);
$display("IE =  IA / 3    = %h%d",IE,IE);
end
endmodule
                      hex    default
IA = -4'd12     = fffffff4        -12
RA = IA / 3     =     fffc      65532
RB = -4'd12     =     fff4      65524
IB = RB / 3     = 00005551      21841
IC = -4'd12 / 3 = 55555551 1431655761
RC = -12 / 3    =     fffc      65532
ID = -12 / 3    = fffffffc         -4
IE =  IA / 3    = fffffffc         -4
```

We might expect the results of all these divisions to be - 4 = -12/3. For integer assignments, the results are correctly signed (`ID` and `IE`). Hex `fffc` (decimal 65532) is the 16-bit two's complement of - 4, so `RA` and `RC` are also correct if we keep track of the signs ourselves. The integer result `IB` is incorrect because Verilog treats `RB` as an unsigned number. Verilog also treats `-4'd12` as an unsigned number in the calculation of `IC` . Once Verilog "loses" a sign, it cannot get it back (see also Section 11.2.5).

Chapter start

Previous page

Chapter start

Previous page

Next page

11.4 Hierarchy

The **module** is the basic unit of code in the Verilog language [Verilog LRM 12.1],

```
module holiday_1(sat, sun, weekend);
  input sat, sun; output weekend;
  assign weekend = sat | sun;
endmodule
```

We do not have to explicitly declare the scalar wires: `saturday` , `sunday` , `weekend` because, since these wires appear in the module interface, they must be declared in an `input` , `output` , or `inout` statement and are thus implicitly declared. The **module interface** provides the means to interconnect two Verilog modules using **ports** [Verilog LRM 12.3]. Each port must be explicitly declared as one of **input**, **output**, or **inout**. Table 11.3 shows the characteristics of ports. Notice that a `reg` cannot be an `input` port or an `inout` port. This is to stop us trying to connect a `reg` to another `reg` that may hold a different value.

TABLE 11.3 Verilog ports.

Verilog port	**input**	**output**	**inout**
Characteristics	**wire** (or other net)	reg or wire (or other net) We can read an output port inside a module	wire (or other net)

Within a module we may **instantiate** other modules, but we cannot declare other modules. Ports are linked using **named association** or **positional association**,

```
‘timescale 100s/1s // Units are 100 seconds with precision of 1s.
module life; wire [3:0] n; integer days;
  wire wake_7am, wake_8am; // Wake at 7 on weekdays else at 8.
  assign n = 1 + (days % 7); // n is day of the week (1-7)
always@(wake_8am or wake_7am)
  $display("Day=",n," hours=%0d ",($time/36)%24,"  8am = ",
    wake_8am,"  7am = ",wake_7am,"  m2.weekday = ", m2.weekday);
  initial days = 0;
  initial begin #(24*36*10);$finish; end // Run for 10 days.
  always #(24*36) days = days + 1; // Bump day every 24hrs.
  rest m1(n, wake_8am); // Module instantiation.
// Creates a copy of module rest with instance name m1,
// ports are linked using positional notation.
  work m2(.weekday(wake_7am), .day(n));
// Creates a copy of module work with instance name m2,
// Ports are linked using named association.
endmodule
module rest(day, weekend); // Module definition.
// Notice the port names are different from the parent.
  input [3:0] day; output weekend; reg weekend;
  always begin #36 weekend = day > 5; end // Need a delay here.
endmodule
module work(day, weekday);
  input [3:0] day; output weekday; reg weekday;
  always begin #36 weekday = day < 6; end // Need a delay here.
endmodule
Day= 1 hours=0   8am = 0  7am = 0  m2.weekday = 0
Day= 1 hours=1   8am = 0  7am = 1  m2.weekday = 1
Day= 6 hours=1   8am = 1  7am = 0  m2.weekday = 0
Day= 1 hours=1   8am = 0  7am = 1  m2.weekday = 1
```

The port names in a module definition and the port names in the parent module may be different. We can **associate** (link or map) ports using the same order in the instantiating statement as we use in the module definition--such as instance `m1` in module `life` . Alternatively we can associate the ports by naming them--such as instance `m2` in module `life` (using a period `'.'` before the port name that we declared in the module definition). Identifiers in a module have local scope. If we want to refer to an identifier outside a module, we use a **hierarchical name** [Verilog LRM12.4] such as `m1.weekend` or `m2.weekday` (as in module `life`), for example. The compiler will first search downward (or inward) then upward (outward) to resolve a hierarchical name [Verilog LRM 12.4-12.5].

Chapter start

Previous page

Chapter start

Previous page

Next page

11.5 Procedures and Assignments

A Verilog **procedure** [Verilog LRM 9.9] is an `always` or `initial` statement, a `task` , or a `function` . The statements within a sequential block (statements that appear between a `begin` and an `end`) that is part of a procedure execute sequentially in the order in which they appear, but the procedure executes concurrently with other procedures. This is a fundamental difference from computer programming languages. Think of each procedure as a microprocessor running on its own and at the same time as all the other microprocessors (procedures). Before I discuss procedures in more detail, I shall discuss the two different types of assignment statements:

- continuous assignments that appear outside procedures
- procedural assignments that appear inside procedures

To illustrate the difference between these two types of assignments, consider again the example used in Section 11.4:

```
module holiday_1(sat, sun, weekend);
  input sat, sun; output weekend;
  assign weekend = sat | sun; // Assignment outside a procedure.
endmodule
```

We can change weekend to a `reg` instead of a `wire` , but then we must declare weekend and use a procedural assignment (inside a procedure--an `always` statement, for example) instead of a continuous assignment. We also need to add some delay (one time tick in the example that follows); otherwise the computer will never be able to get out of the `always` procedure to execute any other procedures:

```
module holiday_2(sat, sun, weekend);
```

```
  input sat, sun; output weekend; reg weekend;
  always #1 weekend = sat | sun; // Assignment inside a procedure.
endmodule
```

We shall cover the continuous assignment statement in the next section, which is followed by an explanation of sequential blocks and procedural assignment statements. Here is some skeleton code that illustrates where we may use these assignment statements:

```
module assignments
//... Continuous assignments go here.
always // beginning of a procedure
  begin // beginning of sequential block
  //... Procedural assignments go here.
  end
endmodule
```

Table 11.4 at the end of Section 11.6 summarizes assignment statements, including two more forms of assignment--you may want to look at this table now.

11.5.1 Continuous Assignment Statement

A **continuous assignment statement** [Verilog LRM 6.1] assigns a value to a `wire` in a similar way that a real logic gate drives a real wire,

```
module assignment_1();
wire pwr_good, pwr_on, pwr_stable; reg Ok, Fire;
assign pwr_stable = Ok & (!Fire);
assign pwr_on = 1;
assign pwr_good = pwr_on & pwr_stable;
initial begin Ok = 0; Fire = 0; #1 Ok = 1; #5 Fire = 1; end
initial begin $monitor("TIME=%0d",$time," ON=",pwr_on, " STABLE=",
    pwr_stable," OK=",Ok," FIRE=",Fire," GOOD=",pwr_good);
  #10 $finish; end
endmodule
TIME=0 ON=1 STABLE=0 OK=0 FIRE=0 GOOD=0
TIME=1 ON=1 STABLE=1 OK=1 FIRE=0 GOOD=1
TIME=6 ON=1 STABLE=0 OK=1 FIRE=1 GOOD=0
```

The assignment statement in this next example models a three-state bus:

```
module assignment_2; reg Enable; wire [31:0] Data;
/* The following single statement is equivalent to a declaration and continuous assi
wire [31:0] DataBus = Enable ? Data : 32'bz;
assign Data = 32'b10101101101011101111000010100001;
  initial begin
    $monitor("Enable=%b DataBus=%b ", Enable, DataBus);
    Enable = 0; #1; Enable = 1; #1; end
endmodule
Enable = 0 DataBus =zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
Enable = 1 DataBus =10101101101011101111000010100001
```

11.5.2 Sequential Block

A **sequential block** [Verilog LRM 9.8] is a group of statements between a **begin** and an **end**. We may declare new variables within a sequential block, but then we must name the block. A sequential block is

considered a statement, so that we may nest sequential blocks.

A sequential block may appear in an **always statement** [Verilog LRM9.9.2], in which case the block executes repeatedly. In contrast, an **initial statement** [Verilog LRM9.9.1] executes only once, so a sequential block within an `initial` statement only executes once--at the beginning of a simulation. It does not matter where the `initial` statement appears--it still executes first. Here is an example:

```
module always_1; reg Y, Clk;
always // Statements in an always statement execute repeatedly:
begin: my_block // Start of sequential block.
  @(posedge Clk) #5 Y = 1; // At +ve edge set Y=1,
  @(posedge Clk) #5 Y = 0; // at the NEXT +ve edge set Y=0.
end // End of sequential block.
always #10 Clk = ~ Clk; // We need a clock.
initial Y = 0; // These initial statements execute
initial Clk = 0; // only once, but first.
initial $monitor("T=%2g",$time,"  Clk=",Clk,"  Y=",Y);
initial #70 $finish;
endmodule
T= 0  Clk=0  Y=0
T=10  Clk=1  Y=0
T=15  Clk=1  Y=1
T=20  Clk=0  Y=1
T=30  Clk=1  Y=1
T=35  Clk=1  Y=0
T=40  Clk=0  Y=0
T=50  Clk=1  Y=0
T=55  Clk=1  Y=1
T=60  Clk=0  Y=1
```

11.5.3 Procedural Assignments

A **procedural assignment** [Verilog LRM 9.2] is similar to an assignment statement in a computer programming language such as C. In Verilog the value of an expression on the RHS of an assignment within a procedure (a procedural assignment) updates a `reg` (or memory element) on the LHS. In the absence of any *timing controls* (see Section 11.6), the `reg` is updated immediately when the statement executes. The `reg` holds its value until changed by another procedural assignment. Here is the BNF definition:

```
blocking_assignment ::= reg-lvalue = [delay_or_event_control] expression
```

(Notice this BNF definition is for a blocking assignment--a type of procedural assignment--see Section 11.6.4.) Here is an example of a procedural assignment (notice that a `wire` can only appear on the RHS of a procedural assignment):

```
module procedural_assign; reg Y, A;
always @(A)
  Y = A; // Procedural assignment.
initial begin A=0; #5; A=1; #5; A=0; #5; $finish; end
initial $monitor("T=%2g",$time,,"A=",A,,,"Y=",Y);
endmodule
T= 0 A=0  Y=0
T= 5 A=1  Y=1
T=10 A=0  Y=0
```

11.6 Timing Controls and Delay

The statements within a sequential block are executed in order, but, in the absence of any delay, they all execute at the same simulation time--the current **time step**. In reality there are delays that are modeled using a timing control.

11.6.1 Timing Control

A **timing control** is either a delay control or an event control [Verilog LRM 9.7]. A **delay control** delays an assignment by a specified amount of time. A **timescale compiler directive** is used to specify the units of time followed by the precision used to calculate time expressions,

```
‘timescale 1ns/10ps // Units of time are ns. Round times to 10 ps.
```

Time units may only be `s` , `ns` , `ps` , or `fs` and the multiplier must be 1, 10, or 100. We can delay an assignment in two different ways:

- Sample the RHS immediately and then delay the assignment to the LHS.
- Wait for a specified time and then assign the value of the LHS to the RHS.

Here is an example of the first alternative (an **intra-assignment delay**):

```
x = #1 y; // intra-assignment delay
```

The second alternative is **delayed assignment**:

```
#1 x = y; // delayed assignment
```

These two alternatives are not the same. The intra-assignment delay is equivalent to the following code:

```
begin // Equivalent to intra-assignment delay.
hold = y; // Sample and hold y immediately.
#1; // Delay.
x = hold; // Assignment to x. Overall same as x = #1 y.
end
```

In contrast, the delayed assignment is equivalent to a delay followed by an assignment as follows:

```
begin // Equivalent to delayed assignment.
#1; // Delay.
x = y; // Assign y to x. Overall same as #1 x = y.
end
```

The other type of timing control, an **event control**, delays an assignment until a specified event occurs. Here is the formal definition:

```
event_control ::= @ event_identifier | @ (event_expression)
event_expression ::= expression | event_identifier
  | posedge expression | negedge expression
  | event_expression or event_expression
```

(Notice there are two different uses of `'or'` in this simplified BNF definition--the last one, in bold, is part of the Verilog language, a keyword.) A positive edge (denoted by the keyword `posedge`) is a transition from `'0'` to `'1'` or `'x'` , or a transition from `'x'` to `'1'`. A negative edge (`negedge`) is a transition from `'1'` to `'0'` or `'x'` , or a transition from `'x'` to `'0'`. Transitions to or from `'z'` do not count. Here are examples of event controls:

```
module delay_controls; reg X, Y, Clk, Dummy;
always #1 Dummy=!Dummy; // Dummy clock, just for graphics.
// Examples of delay controls:
always begin #25 X=1;#10 X=0;#5; end
// An event control:
always @(posedge Clk) Y=X; // Wait for +ve clock edge.
always #10 Clk = !Clk; // The real clock.
initial begin Clk = 0;
  $display("T   Clk X Y");
  $monitor("%2g",$time,,,Clk,,,,X,,Y);
  $dumpvars;#100 $finish; end
endmodule
T   Clk X Y
 0  0   x x
10  1   x x
20  0   x x
25  0   1 x
30  1   1 1
35  1   0 1
40  0   0 1
50  1   0 0
60  0   0 0
65  0   1 0
70  1   1 1
75  1   0 1
80  0   0 1
90  1   0 0
```

The dummy clock in delay_controls helps in the graphical waveform display of the results (it provides a one-time-tick timing grid when we zoom in, for example). Figure 11.1 shows the graphical output from the Waves viewer in VeriWell (white is used to represent the initial unknown values). The assignment statements to `'X'` in the `always` statement repeat (every 25 + 10 + 5 = 40 time ticks).

FIGURE 11.1 Output from the module `delay_controls`.	

Events can be declared (as named events), triggered, and detected as follows:

```
module show_event;
reg clock;
event event_1, event_2; // Declare two named events.
always @(posedge clock) -> event_1; // Trigger event_1.
always @ event_1
begin $display("Strike 1!!"); -> event_2; end // Trigger event_2.
always @ event_2 begin $display("Strike 2!!");
$finish; end // Stop on detection of event_2.
always #10 clock = ~ clock; // We need a clock.
initial clock = 0;
endmodule
Strike 1!!
Strike 2!!
```

11.6.2 Data Slip

Consider this model for a shift register and the simulation output that follows:

```
module data_slip_1 (); reg Clk, D, Q1, Q2;
/************** bad sequential logic below ***************/
always @(posedge Clk)  Q1 = D;
always @(posedge Clk)  Q2 = Q1; // Data slips here!
/************** bad sequential logic above ***************/
initial begin Clk = 0; D = 1; end always #50 Clk = ~Clk;
initial begin $display("t   Clk D Q1 Q2");
$monitor("%3g",$time,,Clk,,,,D,,Q1,,,Q2); end
initial #400 $finish; // Run for 8 cycles.
initial $dumpvars;
endmodule
t   Clk D Q1 Q2
  0 0   1 x  x
 50 1   1 1  1
100 0   1 1  1
150 1   1 1  1
200 0   1 1  1
250 1   1 1  1
300 0   1 1  1
350 1   1 1  1
```

The first clock edge at `t = 50` causes `Q1` to be updated to the value of `D` at the clock edge (a `'1'`), and at the same time `Q2` is updated to this new value of `Q1` . The data, `D` , has passed through both `always` statements. We call this problem **data slip**.

If we include delays in the `always` statements (labeled 3 and 4) in the preceding example, like this--

```
always @(posedge Clk)  Q1 = #1 D;  // The delays in the assignments
always @(posedge Clk)  Q2 = #1 Q1; // fix the data slip.
```

--we obtain the correct output:

```
t   Clk D Q1 Q2
  0 0   1 x  x
 50 1   1 x  x
 51 1   1 1  x
100 0   1 1  x
150 1   1 1  x
151 1   1 1  1
200 0   1 1  1
250 1   1 1  1
300 0   1 1  1
350 1   1 1  1
```

11.6.3 Wait Statement

The **wait statement** [Verilog LRM9.7.5] suspends a procedure until a condition becomes true. There must be another concurrent procedure that alters the condition (in this case the variable `Done` --in general the condition is an expression) in the following `wait` statement; otherwise we are placed on "infinite hold":

```
wait (Done) $stop; // Wait until Done = 1 then stop.
```

Notice that the Verilog `wait` statement does not look for an event or a change in the condition; instead it is level-sensitive--it only cares that the condition is true.

```
module test_dff_wait;
reg D, Clock, Reset; dff_wait u1(D, Q, Clock, Reset);
initial begin D=1; Clock=0;Reset=1'b1; #15 Reset=1'b0; #20 D=0; end
always #10 Clock = !Clock;
initial begin $display("T  Clk D Q Reset");
  $monitor("%2g",$time,,Clock,,,,D,,Q,,Reset); #50 $finish; end
endmodule
module dff_wait(D, Q, Clock, Reset);
output Q; input D, Clock, Reset; reg Q; wire D;
always @(posedge Clock) if (Reset !== 1) Q = D;
always begin wait (Reset == 1) Q = 0; wait (Reset !== 1); end
endmodule
T  Clk D Q Reset
 0 0   1 0 1
10 1   1 0 1
15 1   1 0 0
20 0   1 0 0
30 1   1 1 0
35 1   0 1 0
40 0   0 1 0
```

We must include `wait` statements in module `dff_wait` above to wait for both `Reset==1` and `Reset==0` . If we were to omit the `wait` statement for `Reset==0` , as in the following code:

```
module dff_wait(D,Q,Clock,Reset);
output Q; input D,Clock,Reset; reg Q; wire D;
always @(posedge Clock) if (Reset !== 1) Q = D;
// We need another wait statement here or we shall spin forever.
always begin wait (Reset == 1) Q = 0; end
endmodule
```

the simulator would cycle endlessly, and we would need to press the 'Stop' button or 'CTRL-C' to halt the simulator. Here is the console window in VeriWell:

```
C1> .
T  Clk D Q Reset                  <- at this point nothing happens, so press CTRL-C
Interrupt at time 0
C1>
```

11.6.4 Blocking and Nonblocking Assignments

If a procedural assignment in a sequential block contains a timing control, then the execution of the following statement is delayed or **blocked**. For this reason a procedural assignment statement is also known as a **blocking procedural assignment statement** [Verilog LRM 9.2]. We covered this type of statement in Section 11.5.3. The **nonblocking procedural assignment statement** allows execution in a sequential block to continue and registers are all updated together at the end of the current time step. Both types of procedural assignment may contain timing controls. Here is an artificially complicated example that illustrates the different types of assignment:

```
module delay;
reg a,b,c,d,e,f,g,bds,bsd;
initial begin
a = 1; b = 0; // No delay control.
#1 b = 1;     // Delayed assignment.
c = #1 1;     // Intra-assignment delay.
#1;           // Delay control.
d = 1;        //
e <= #1 1;    // Intra-assignment delay, nonblocking assignment
#1 f <= 1;    // Delayed nonblocking assignment.
g <= 1;       // Nonblocking assignment.
end
initial begin #1 bds = b; end // Delay then sample (ds).
initial begin bsd = #1 b; end // Sample then delay (sd).
initial begin $display("t a b c d e f g bds bsd");
$monitor("%g",$time,,a,,b,,c,,d,,e,,f,,g,,bds,,,,bsd); end
endmodule
t a b c d e f g bds bsd
0 1 0 x x x x x x   x
1 1 1 x x x x x 1   0
2 1 1 1 x x x x 1   0
3 1 1 1 1 x x x 1   0
4 1 1 1 1 1 1 1 1   0
```

Many synthesis tools will not allow us to use blocking and nonblocking procedural assignments to the same `reg` within the same sequential block.

11.6.5 Procedural Continuous Assignment

A **procedural continuous assignment statement** [Verilog LRM 9.3] (sometimes called a quasicontinuous assignment statement) is a special form of the `assign` statement that we use within a sequential block. For example, the following flip-flop model assigns to `q` depending on the clear, clr_, and preset, pre_, inputs (in general it is considered very bad form to use a trailing underscore to signify active-low signals as I have done to save space; you might use " _n " instead).

```
module dff_procedural_assign;
reg d,clr_,pre_,clk; wire q; dff_clr_pre dff_1(q,d,clr_,pre_,clk);
always #10 clk = ~clk;
initial begin clk = 0; clr_ = 1; pre_ = 1; d = 1;
  #20; d = 0; #20; pre_ = 0; #20; pre_ = 1; #20; clr_ = 0;
  #20; clr_ = 1; #20; d = 1; #20; $finish; end
initial begin
  $display("T  CLK PRE_ CLR_ D Q");
  $monitor("%3g",$time,,,clk,,,,pre_,,,,clr_,,,,d,,q); end
endmodule
module dff_clr_pre(q,d,clear_,preset_,clock);
output q; input d,clear_,preset_,clock; reg q;
always @(clear_ or preset_)
  if (!clear_) assign q = 0; // active-low clear
  else if(!preset_) assign q = 1; // active-low preset
  else deassign q;
always @(posedge clock) q = d;
endmodule
T  CLK PRE_ CLR_ D Q
  0  0   1   1   1 x
 10  1   1   1   1 1
 20  0   1   1   0 1
 30  1   1   1   0 0
 40  0   0   1   0 1
 50  1   0   1   0 1
 60  0   1   1   0 1
 70  1   1   1   0 0
 80  0   1   0   0 0
 90  1   1   0   0 0
100  0   1   1   0 0
110  1   1   1   0 0
120  0   1   1   1 0
130  1   1   1   1 1
```

We have now seen all of the different forms of Verilog assignment statements. The following skeleton code shows where each type of statement belongs:

```
module all_assignments
//... continuous assignments.
always // beginning of procedure
  begin // beginning of sequential block
  //... blocking procedural assignments.
  //... nonblocking procedural assignments.
  //... procedural continuous assignments.
  end
endmodule
```

Table 11.4 summarizes the different types of assignments.

TABLE 11.4 Verilog assignment statements.				
Type of Verilog assignment	**Continuous assignment statement**	**Procedural assignment statement**	**Nonblocking procedural assignment statement**	**Procedural continuous assignment statement**
Where it can occur	outside an always or initial statement, task, or function	inside an always or initial statement, task, or function	inside an always or initial statement, task, or function	always or initial statement, task, or function

Example	`wire [31:0] DataBus;`
	`assign DataBus =`
	`Enable ? Data :`
	`32'bz`
		`reg Y;`
		`always`
		`@(posedge clock) Y = 1;`
			`reg Y;`
			`always Y <= 1;`
				`always`
				`@(Enable)`
				`if(Enable)`
				`assign Q = D;`
				`else deassign Q;`

Valid LHS of assignment	net	register or memory element	register or memory element	net
Valid RHS of assignment	<expression> net, reg or memory element	<expression> net, reg or memory element	<expression> net, reg or memory element	<expression> net, reg or memory element
Book	11.5.1	11.5.3	11.6.4	11.6.5
Verilog LRM	6.1	9.2	9.2.2	9.3

Chapter start

Previous page

Previous page

Next page

11.7 Tasks and Functions

A **task** [Verilog LRM 10.2] is a type of procedure, called from another procedure. A task has both inputs and outputs but does not return a value. A task may call other tasks and functions. A **function** [Verilog LRM 10.3] is a procedure used in any expression, has at least one input, no outputs, and returns a single value. A function may not call a task. In Section 11.5 we covered all of the different Verilog procedures except for tasks and functions. Now that we have covered timing controls, we can explain the difference between tasks and functions: Tasks may contain timing controls but functions may not. The following two statements help illustrate the difference between a function and a task:

```
Call_A_Task_And_Wait (Input1, Input2, Output);
Result_Immediate = Call_A_Function (All_Inputs);
```

Functions are useful to model combinational logic (rather like a subroutine):

```
module F_subset_decode; reg [2:0]A, B, C, D, E, F;
initial begin A = 1; B = 0; D = 2; E = 3;
  C = subset_decode(A, B); F = subset_decode(D,E);
  $display("A B C D E F"); $display(A,,B,,C,,D,,E,,F); end
function [2:0] subset_decode; input [2:0] a, b;
  begin if (a <= b) subset_decode = a; else subset_decode = b; end
endfunction
endmodule
A B C D E F
1 0 0 2 3 2
```

Chapter start

Previous page

Previous page

Next page

11.8 Control Statements

In this section we shall discuss the Verilog `if` , `case` , `loop` , `disable` , `fork` , and `join` statements that control the flow of code execution.

11.8.1 Case and If Statement

An **if statement** [Verilog LRM 9.4] represents a two-way branch. In the following example, `switch` has to be true to execute `'Y = 1'` ; otherwise `'Y = 0'` is executed:

```
if(switch) Y = 1; else Y = 0;
```

The **case statement** [Verilog LRM 9.5] represents a multiway branch. A controlling expression is matched with case **expressions** in each of the **case items** (or arms) to determine a match,

```
module test_mux; reg a, b, select; wire out;
mux mux_1(a, b, out, select);
initial begin #2; select = 0; a = 0; b = 1;
  #2; select = 1'bx; #2; select = 1'bz; #2; select = 1; end
initial $monitor("T=%2g",$time,"  Select=",select,"  Out=",out);
initial #10 $finish;
endmodule
module mux(a, b, mux_output, mux_select); input a, b, mux_select;
```

```
output mux_output; reg mux_output;
always begin
case(mux_select)
  0: mux_output = a;
  1: mux_output = b;
  default mux_output = 1'bx; // If select = x or z set output to x.
endcase
#1; // Need some delay, otherwise we'll spin forever.
end
endmodule
T= 0  Select=x  Out=x
T= 2  Select=0  Out=x
T= 3  Select=0  Out=0
T= 4  Select=x  Out=0
T= 5  Select=x  Out=x
T= 6  Select=z  Out=x
T= 8  Select=1  Out=x
T= 9  Select=1  Out=1
```

Notice that the `case` statement must be inside a sequential block (inside an `always` statement). Because the `case` statement is inside an `always` statement, it needs some delay; otherwise the simulation runs forever without advancing simulation time. The **casex statement** handles both `'z'` and `'x'` as don't care (so that they match any bit value), the **casez statement** handles `'z'` bits, and only `'z'` bits, as don't care. Bits in case expressions may be set to `'?'` representing don't care values, as follows:

```
casex (instruction_register[31:29])
  3b'??1 : add;
  3b'?1? : subtract;
  3b'1?? : branch;
endcase
```

11.8.2 Loop Statement

A **loop statement** [Verilog LRM 9.6] is a **for**, **while**, **repeat**, or **forever** statement. Here are four examples, one for each different type of loop statement, each of which performs the same function. The comments with each type of loop statement illustrate how the controls work:

```
module loop_1;
integer i; reg [31:0] DataBus; initial DataBus = 0;
initial begin
/************** Insert loop code after here. ******************/
/* for(Execute this assignment once before starting loop; exit loop if this expressi
for(i = 0; i <= 15; i = i+1) DataBus[i] = 1;
/*************** Insert loop code before here. ***************/
end
initial begin
$display("DataBus = %b",DataBus);
#2; $display("DataBus = %b",DataBus); $finish;
end
endmodule
```

Here is the `while` statement code (to replace line 4 in module `loop_1`):

```
i = 0;
/* while(Execute next statement while this expression is true.) */
while(i <= 15) begin DataBus[i] = 1; i = i+1; end
```

Here is the `repeat` statement code (to replace line 4 in module `loop_1`):

```
i = 0;
/* repeat(Execute next statement the number of times corresponding to the evaluation
repeat(16) begin DataBus[i] = 1; i = i+1; end
```

Here is the `forever` statement code (to replace line 4 in module `loop_1`):

```
i = 0;
/* A forever statement loops continuously. */
forever begin : my_loop
  DataBus[i] = 1;
  if (i == 15) #1 disable my_loop; // Need to let time advance to exit.
  i = i+1;
end
```

The output for all four forms of looping statement is the same:

```
DataBus = 00000000000000000000000000000000
DataBus = 00000000000000001111111111111111
```

11.8.3 Disable

The **disable statement** [Verilog LRM 11] stops the execution of a labeled sequential block and skips to the end of the block:

```
forever
begin: microprocessor_block // Labeled sequential block.
  @(posedge clock)
  if (reset) disable microprocessor_block; // Skip to end of block.
  else Execute_code;
end
```

Use the `disable` statement with caution in ASIC design. It is difficult to implement directly in hardware.

11.8.4 Fork and Join

The **fork statement** and **join statement** [Verilog LRM 9.8.2] allows the execution of two or more parallel threads in a **parallel block**:

```
module fork_1
event eat_breakfast, read_paper;
initial begin
  fork
  @eat_breakfast; @read_paper;
  join
end
endmodule
```

This is another Verilog language feature that should be used with care in ASIC design, because it is difficult to implement in hardware.

11.9 Logic-Gate Modeling

Verilog has a set of built-in logic models and you may also define your own models.

11.9.1 Built-in Logic Models

Verilog's built-in logic models are the following **primitives** [Verilog LRM7]:

- `and, nand, nor, or, xor, xnor`

You may use these primitives as you use modules. For example:

```
module primitive;
nand (strong0, strong1) #2.2
  Nand_1(n001, n004, n005),
  Nand_2(n003, n001, n005, n002);
nand (n006, n005, n002);
endmodule
```

This module models three NAND gates (Figure 11.2). The first gate (line 3) is a two-input gate named `Nand_1` ; the second gate (line 4) is a three-input gate named `Nand_2` ; the third gate (line 5) is unnamed. The first two gates have strong drive strengths [Verilog LRM3.4] (these are the defaults anyway) and 2.2 ns delay; the third gate takes the default values for drive strength (strong) and delay (zero). The first port of a primitive gate is always the output port. The remaining ports for a primitive gate (any number of them) are the input ports.

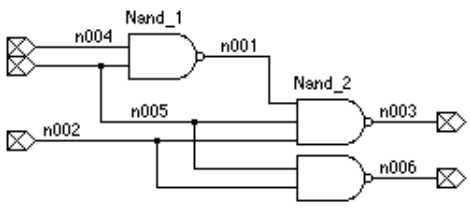

FIGURE 11.2 An example schematic (drawn with Capilano's DesignWorks) to illustrate the use of Verilog primitive gates.

Table 11.5 shows the definition of the and gate primitive (I use lowercase `'and'` as the name of the Verilog primitive, rather than `'AND'`, since Verilog is case-sensitive). Notice that if one input to the primitive `'and'` gate is zero, the output is zero, no matter what the other input is.

TABLE 11.5 Definition of the Verilog primitive 'and' gate.				
'and'	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

11.9.2 User-Defined Primitives

We can define primitive gates (a **user-defined primitive** or **UDP**) using a truth-table specification [Verilog LRM8]. The first port of a UDP must be an `output` port, and this must be the only o `utput` port (we may not use vector or `inout` ports):

```
primitive Adder(Sum, InA, InB);
output Sum; input Ina, InB;
table
// inputs : output
00 : 0;
01 : 1;
10 : 1;
11 : 0;
endtable
endprimitive
```

We may only specify the values `'0'`, `'1'`, and `'x'` as inputs in a **UDP truth table**. Any `'z'` input is treated as an `'x'`. If there is no entry in a UDP truth table that exactly matches a set of inputs, the output is `'x'` (unknown).

We can construct a UDP model for sequential logic by including a state in the UDP truth-table definition. The state goes between an input and an output in the table and the output then represents the next state. The following sequential UDP model also illustrates the use of shorthand notation in a UDP truth table:

```
primitive DLatch(Q, Clock, Data);
output Q; reg Q; input Clock, Data;
table
//inputs : present state : output (next state)
1 0 : ? : 0; // ? represents 0,1, or x (input or present state).
1 1 : b : 1; // b represents 0 or 1 (input or present state).
1 1 : x : 1; // Could have combined this with previous line.
0 ? : ? : -; // - represents no change in an output.
endtable
endprimitive
```

Be careful not to confuse the `'?'` in a UDP table (shorthand for `'0'`, `'1'`, or `'x'`) with the `'?'` in a

constant that represents an extension to `'z'` (Section 11.2.4) or the `'?'` in a `case` statement that represents don't care values (Section 11.8.1).

For sequential UDP models that need to detect edge transitions on inputs, there is another special truth-table notation `(ab)` that represents a change in logic value from `a` to `b` . For example, `(01)` represents a rising edge. There are also shorthand notations for various edges:

- `*` is `(??)`
- `r` is `(01)`
- `f` is `(10)`
- `p` is `(01)`, `(0x)`, or `(x1)`
- `n` is `(10)`, `(1x)`, or `(x0)`

```
primitive DFlipFlop(Q, Clock, Data);
output Q; reg Q; input Clock, Data;
table
//inputs : present state : output (next state)
r    0 : ? : 0 ; // rising edge, next state = output = 0
r    1 : ? : 1 ; // rising edge, next state = output = 1
(0x) 0 : 0 : 0 ; // rising edge, next state = output = 0
(0x) 1 : 1 : 1 ; // rising edge, next state = output = 1
(?0) ? : ? : - ; // falling edge, no change in output
? (??) : ? : - ; // no clock edge, no change in output
endtable
endprimitive
```

Chapter start

Previous page

Chapter start

Previous page

Next page

11.10 Modeling Delay

Verilog has a set of built-in methods to define delays. This is very important in ASIC physical design. Before we start layout, we can use ASIC cell library models written in Verilog that include logic delays as a function of fanout and estimated wiring loads. After we have completed layout, we can extract the wiring capacitance, allowing us to calculate the exact delay values. Using the techniques described in this section, we can then back-annotate our Verilog netlist with postlayout delays and complete a postlayout simulation.

We can complete this back-annotation process in a standard fashion since delay specification is part of the Verilog language. This makes working with an ASIC cell library and the ASIC foundry that will fabricate our ASIC much easier. Typically an ASIC library company might sell us a cell library

complete with Verilog models that include all the minimum, typical, and maximum delays as well as the different values for rising and falling transitions. The ASIC foundry will provide us with a delay calculator that calculates the net delays (this is usually proprietary technology) from the layout. These delays are held in a separate file (the **Standard Delay Format**, **SDF**, is widely used) and then mapped to parameters in the Verilog models. If we complete back-annotation and a postlayout simulation using an approved cell library, the ASIC foundry will "sign off" on our design. This is basically a guarantee that our chip will work according to the simulation. This ability to design sign-off quality ASIC cell libraries is very important in the ASIC design process.

11.10.1 Net and Gate Delay

We saw how to specify a delay control for any statement in Section 11.6. In fact, Verilog allows us to specify minimum, typical, and maximum values for the delay as follows [Verilog LRM7.15]:

```
#(1.1:1.3:1.7) assign delay_a = a; // min:typ:max
```

We can also specify the delay properties of a `wire` in a similar fashion:

```
wire #(1.1:1.3:1.7) a_delay; // min:typ:max
```

We can specify delay in a `wire` declaration together with a continuous assignment as in the following example:

```
wire #(1.1:1.3:1.7) a_delay = a; // min:typ:max
```

but in this case the delay is associated with the driver and not with the `wire` .

In Section 11.9.1 we explained that we can specify a delay for a logic primitive. We can also specify minimum, typical, and maximum delays as well as separate delays for rising and falling transitions for primitives as follows [Verilog LRM4.3]:

```
nand #3.0 nd01(c, a, b);
nand #(2.6:3.0:3.4) nd02(d, a, b); // min:typ:max
nand #(2.8:3.2:3.4, 2.6:2.8:2.9) nd03(e, a, b);
// #(rising, falling) delay
```

The first NAND gate, `nd01` , has a delay of 3 ns (assuming we specified nanoseconds as the timescale) for both rising and falling delays. The NAND gate `nd02` has a triplet for the delay; this corresponds to a minimum (2.6 ns), typical (3.0 ns), and a maximum delay (3.4 ns). The NAND gate `nd03` has two triplets for the delay: The first triplet specifies the min/typ/max rising delay (`'0'` or `'x'` or `'z'` to `'1'`), and the second triplet specifies the min/typ/max falling delay (`'1'` or `'x'` or `'z'` to `'0'`).

Some primitives can produce a high-impedance output, `'z'` . In this case we can specify a triplet of delay values corresponding to rising transition, falling transition, and the delay to transition to `'z'` (from `'0'` or `'1'` to `'z'` --this is usually the delay for a three-state driver to turn off or float). We can do the same thing for net types,

```
wire #(0.5,0.6,0.7) a_z = a; // rise/fall/float delays
```

11.10.2 Pin-to-Pin Delay

The **specify block** [Verilog LRM 13] is a special construct in Verilog that allows the definition of **pin-to-pin delays** across a module. The use of a specify block can include the use of built-in system functions to check setup and hold times, for example. The following example illustrates how to specify pin-to-pin timing for a D flip-flop. We declare the timing parameters first followed by the paths. This example uses the UDP from Section 11.9.2, which does not include preset and clear (so only part of the flip-flop function is modeled), but includes the timing for preset and clear for illustration purposes.

```
module DFF_Spec; reg D, clk;
DFF_Part DFF1 (Q, clk, D, pre, clr);
initial begin D = 0; clk = 0; #1; clk = 1; end
initial $monitor("T=%2g", $time," clk=", clk," Q=", Q);
endmodule
module DFF_Part(Q, clk, D, pre, clr);
  input clk, D, pre, clr; output Q;
  DFlipFlop(Q, clk, D); // No preset or clear in this UDP.
  specify
    specparam
    tPLH_clk_Q = 3, tPHL_clk_Q = 2.9,
    tPLH_set_Q = 1.2, tPHL_set_Q = 1.1;
  (clk => Q) = (tPLH_clk_Q, tPHL_clk_Q);
  (pre, clr *> Q) = (tPLH_set_Q, tPHL_set_Q);
  endspecify
endmodule
T= 0 clk=0 Q=x
T= 1 clk=1 Q=x
T= 4 clk=1 Q=0
```

There are the following two ways to specify paths (module `DFF_part` above uses both) [Verilog LRM13.3]:

- `x => y` specifies a **parallel connection** (or parallel path) between `x` and `y` (`x` and `y` must have the same number of bits).
- `x *> y` specifies a **full connection** (or full path) between `x` and `y` (every bit in `x` is connected to `y`) . In this case `x` and `y` may be different sizes.

The delay of some logic cells depends on the state of the inputs. This can be modeled using a **state-dependent path delay**. Here is an example:

```
‘timescale 1 ns / 100 fs
module M_Spec; reg A1, A2, B; M M1 (Z, A1, A2, B);
initial begin A1=0;A2=1;B=1;#5;B=0;#5;A1=1;A2=0;B=1;#5;B=0; end
initial
  $monitor("T=%4g",$realtime," A1=",A1," A2=",A2," B=",B," Z=",Z);
endmodule
‘timescale 100 ps / 10 fs
module M(Z, A1, A2, B); input A1, A2, B; output Z;
or (Z1, A1, A2); nand (Z, Z1, B); // OAI21
/*A1 A2 B Z  Delay=10*100 ps unless indicated in the table below.
  0  0  0 1
  0  0  1 1
  0  1  0 1  B:0->1 Z:1->0 delay=t2
  0  1  1 0  B:1->0 Z:0->1 delay=t1
  1  0  0 1  B:0->1 Z:1->0 delay=t4
  1  0  1 0  B:1->0 Z:0->1 delay=t3
  1  1  0 1
```

```
  1  1  1 0 */
specify specparam t1 = 11, t2 = 12; specparam t3 = 13, t4 = 14;
  (A1 => Z) = 10; (A2 => Z) = 10;
  if (~A1) (B => Z) = (t1, t2);if (A1) (B => Z) = (t3, t4);
endspecify
endmodule
T=   0 A1=0 A2=1 B=1 Z=x
T=   1 A1=0 A2=1 B=1 Z=0
T=   5 A1=0 A2=1 B=0 Z=0
T= 6.1 A1=0 A2=1 B=0 Z=1
T=  10 A1=1 A2=0 B=1 Z=1
T=  11 A1=1 A2=0 B=1 Z=0
T=  15 A1=1 A2=0 B=0 Z=0
T=16.3 A1=1 A2=0 B=0 Z=1
```

Chapter start

Previous page

Previous page

Next page

11.11 Altering Parameters

Here is an example of a module that uses a parameter [Verilog LRM3.10, 12.2]:

```
module Vector_And(Z, A, B);
  parameter CARDINALITY = 1;
  input [CARDINALITY-1:0] A, B;
  output [CARDINALITY-1:0] Z;
  wire [CARDINALITY-1:0] Z = A & B;
endmodule
```

We can override this parameter when we instantiate the module as follows:

```
module Four_And_Gates(OutBus, InBusA, InBusB);
  input [3:0] InBusA, InBusB; output [3:0] OutBus;
  Vector_And #(4) My_AND(OutBus, InBusA, InBusB); // 4 AND gates
endmodule
```

The parameters of a module have local scope, but we may override them using a **defparam** statement and a hierarchical name, as in the following example:

```
module And_Gates(OutBus, InBusA, InBusB);
  parameter WIDTH = 1;
  input [WIDTH-1:0] InBusA, InBusB; output [WIDTH-1:0] OutBus;
  Vector_And #(WIDTH) My_And(OutBus, InBusA, InBusB);
endmodule
module Super_Size; defparam And_Gates.WIDTH = 4; endmodule
```

11.12 A Viterbi Decoder

This section describes an ASIC design for a Viterbi decoder using Verilog. Christeen Gray completed the original design as her MS thesis at the University of Hawaii (UH) working with VLSI Technology, using the Compass ASIC Synthesizer and a VLSI Technology cell library. The design was mapped from VLSI Technology design rules to Hewlett-Packard design rules; prototypes were fabricated by Hewlett-Packard (through Mosis) and tested at UH.

11.12.1 Viterbi Encoder

Viterbi encoding is widely used for satellite and other noisy **communications channels**. There are two important components of a channel using Viterbi encoding: the **Viterbi encoder** (at the transmitter) and the **Viterbi decoder** (at the receiver). A Viterbi encoder includes extra information in the transmitted signal to reduce the probability of errors in the received signal that may be corrupted by noise.

I shall describe an encoder in which every two bits of a data stream are encoded into three bits for transmission. The ratio of input to output information in an encoder is the **rate** of the encoder; this is a rate 2/3 encoder. The following equations relate the three encoder output bits (Y_n^2 , Y_n^1 , and Y_n^0) to the two encoder input bits (X_n^2 and X_n^1) at a time nT:

$Y_n^2 = X_n^2$

$Y_n^1 = X_n^1$ xor X_{n-2}^1

$Y_n^0 = X_{n-1}^1$

We can write the input bits as a single number. Thus, for example, if $X_n^2 = 1$ and $X_n^2 = 0$, we can write $X_n = 2$. Equation 11.1 defines a state machine with two memory elements for the two last input values for X_n^1 : X_{n-1}^1 and X_{n-2}^1 . These two state variables define four states: $\{X_{n-1}^1, X_{n-2}^1\}$, with $S_0 = \{0, 0\}$, $S_1 = \{1, 0\}$, $S_2 = \{0, 1\}$, and $S_3 = \{1, 1\}$. The 3-bit output Y_n is a function of the state and current 2-bit input X .

2-bit input X_n .

The following Verilog code describes the rate 2/3 encoder. This model uses two D flip-flops as the state register. When reset (using active-high input signal `res`) the encoder starts in state S_0 . In Verilog I represent $Y_n{}^2$ by `Y2N` , for example.

```
/*****************************************************/
/* module viterbi_encode                             */
/*****************************************************/
/* This is the encoder. X2N (msb) and X1N form the 2-bit input
message, XN. Example: if X2N=1, X1N=0, then XN=2. Y2N (msb), Y1N, and
Y0N form the 3-bit encoded signal, YN (for a total constellation of 8
PSK signals that will be transmitted). The encoder uses a state
machine with four states to generate the 3-bit output, YN, from the
2-bit input, XN. Example: the repeated input sequence XN = (X2N, X1N)
= 0, 1, 2, 3 produces the repeated output sequence YN = (Y2N, Y1N,
Y0N) = 1, 0, 5, 4. */
module viterbi_encode(X2N,X1N,Y2N,Y1N,Y0N,clk,res);
input X2N,X1N,clk,res; output Y2N,Y1N,Y0N;
wire X1N_1,X1N_2,Y2N,Y1N,Y0N;
dff dff_1(X1N,X1N_1,clk,res); dff dff_2(X1N_1,X1N_2,clk,res);
assign Y2N=X2N; assign Y1N=X1N ^ X1N_2; assign Y0N=X1N_1;
endmodule
```

Figure 11.3 shows the state diagram for this encoder. The first four rows of Table 11.6 show the four different transitions that can be made from state S_0 . For example, if we reset the encoder and the input is $X_n = 3$ ($X_n{}^2 = 1$ and $X_n{}^1 = 1$), then the output will be $Y_n = 6$ ($Y_n{}^2 = 1$, $Y_n{}^1 = 1$, $Y_n{}^0 = 0$) and the next state will be S_1 .

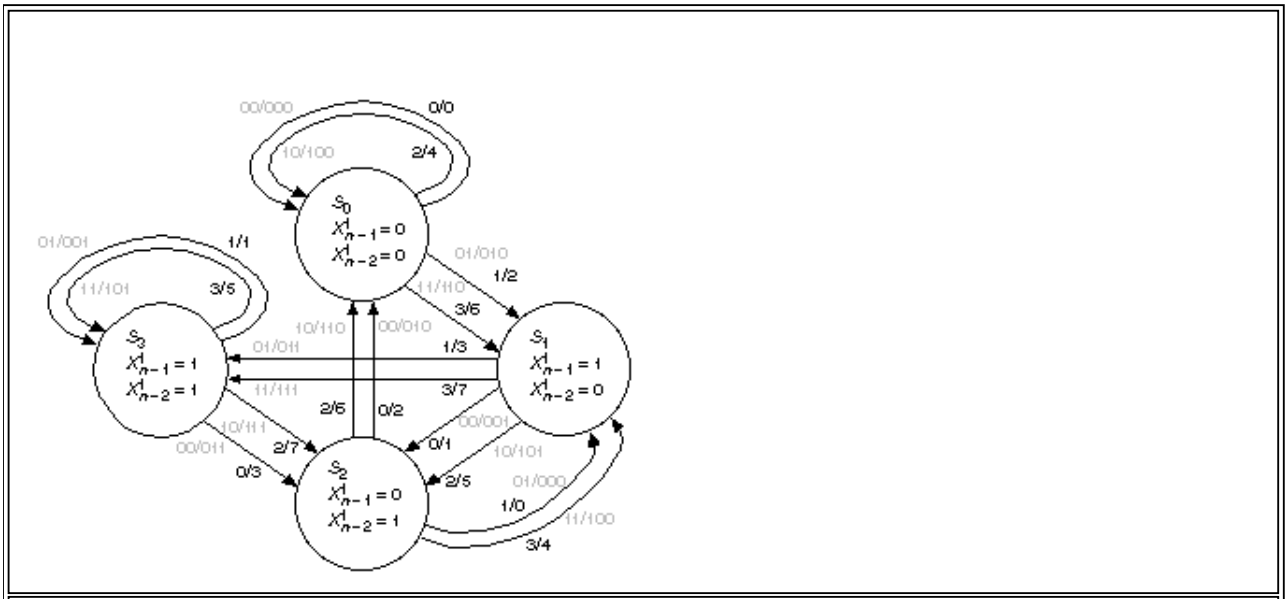

FIGURE 11.3 A state diagram for a rate 2/3 Viterbi encoder. The inputs and outputs are shown in binary as $X_n{}^2\, X_n{}^1 / Y_n{}^2 Y_n{}^1 Y_n{}^0$, and in decimal as X_n / Y_n .

TABLE 11.6 State table for the rate 2/3 Viterbi encoder.

Present state	**Inputs**		**State variables**		**Outputs**			**Next state**	
	X_n^2	X_n^1	X_{n-1}^1	X_{n-2}^1	Y_n^2	Y_n^1	Y_n^0		
	X_n^2	X_n^1	X_{n-1}^1	X_{n-2}^1	X_n^2	$= X_n^1$ xor X_{n-2}^1	$= X_{n-1}^1$	$\{X_{n-1}^1, X_{n-2}^1\}$	
S_0	0	0	0	0	0	0	0	00	S_0
S_0	0	1	0	0	0	1	0	10	S_1
S_0	1	0	0	0	1	0	0	00	S_0
S_0	1	1	0	0	1	1	0	10	S_1
S_1	0	0	1	0	0	0	1	01	S_2
S_1	0	1	1	0	0	1	1	11	S_3
S_1	1	0	1	0	1	0	1	01	S_2
S_1	1	1	1	0	1	1	1	11	S_3
S_2	0	0	0	1	0	1	0	00	S_0
S_2	0	1	0	1	0	0	0	10	S_1
S_2	1	0	0	1	1	1	0	00	S_0
S_2	1	1	0	1	1	0	0	10	S_1
S_3	0	0	1	1	0	1	1	01	S_2
S_3	0	1	1	1	0	0	1	11	S_3
S_3	1	0	1	1	1	1	1	01	S_2
S_3	1	1	1	1	1	0	1	11	S_3

As an example, the repeated encoder input sequence X_n = 0, 1, 2, 3, ... produces the encoder output sequence Y_n = 1, 0, 5, 4, ... repeated. Table 11.7 shows the state transitions for this sequence, including the initialization steps.

FIGURE 11.4 The signal constellation for an 8 PSK (phase-shift keyed) code.

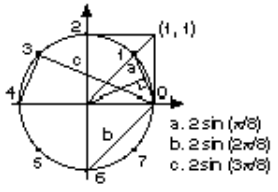

TABLE 11.7 A sequence of transmitted signals for the rate 2/3 Viterbi encoder										
Time	**Inputs**		State variables		Outputs			**Present state**	**Next state**
ns	X_n^2	X_n^1	X_{n-1}^1	X_{n-2}^1	Y_n^2	Y_n^1	Y_n^0		
0	1	1	x	x	1	x	x	$S_?$	$S_?$
10	1	1	0	0	1	1	0	S_0	S_1
50	0	0	1	0	0	0	1	S_1	S_2
150	0	1	0	1	0	0	0	S_2	S_1
250	1	0	1	0	1	0	1	S_1	S_2
350	1	1	0	1	1	0	0	S_2	S_1
450	0	0	1	0	0	0	1	S_1	S_2
550	0	1	0	1	0	0	0	S_2	S_1
650	1	0	1	0	1	0	1	S_1	S_2
750	1	1	0	1	1	0	0	S_2	S_1
850	0	0	1	0	0	0	1	S_1	S_2
950	0	1	0	1	0	0	0	S_2	S_1

Next we transmit the eight possible encoder outputs (Y_n = 0-7) as **signals** over our noisy communications channel (perhaps a microwave signal to a satellite) using the **signal constellation** shown in Figure 11.4. Typically this is done using **phase-shift keying** (**PSK**) with each signal position corresponding to a different phase shift in the transmitted carrier signal.

11.12.2 The Received Signal

The noisy signal enters the receiver. It is now our task to discover which of the eight possible signals were transmitted at each time step. First we calculate the distance of each received signal from each of the known eight positions in the signal constellation. Table 11.8 shows the distances between signals in the 8PSK constellation. We are going to assume that there is no noise in the channel to illustrate the operation of the Viterbi decoder, so that the distances in Table 11.8 represent the possible distance measures of our received signal from the 8PSK signals.

The distances, **X**, in the first column of Table 11.8 are the geometric or algebraic distances. We measure the **Euclidean distance**, $\mathbf{E} = \mathbf{X}^2$ shown as **B** (the binary quantized value of **E**) in Table 11.8. The rounding errors that result from conversion to fixed-width binary are **quantization errors** and are important in any practical implementation of the Viterbi decoder. The effect of the quantization error is to add a form of noise to the received signal.

The following code models the receiver section that digitizes the noisy analog received signal and computes the binary distance measures. Eight binary-distance measures, `in0-in7` , are generated each time a signal is received. Since each of the distance measures is 3 bits wide, there are a total of 24 bits (8 ¥ 3) that form the digital inputs to the Viterbi decoder.

TABLE 11.8 Distance measures for Viterbi encoding (8PSK).

Signal	Algebraic distance from signal 0	**X** = Distance from signal 0	Euclidean distance $\mathbf{E} = \mathbf{X}^2$	**B** = binary quantized value of **E**	**D** = decimal value of B	Quantization error **Q** = **D** - 1.75 **E**
0	2 sin (0 π / 8)	0.00	0.00	000	0	0
1	2 sin (1 π / 8)	0.77	0.59	001	1	-0.0325
2	2 sin (2 π / 8)	1.41	2.00	100	4	0.5
3	2 sin (3 π / 8)	1.85	3.41	110	6	0.0325
4	2 sin (4 π / 8)	2.00	4.00	111	7	0
5	2 sin (5 π / 8)	1.85	3.41	110	6	0.0325
6	2 sin (6 π / 8)	1.41	2.00	100	4	0.5
7	2 sin (7 π / 8)	0.77	0.59	001	1	-0.0325

```
/*****************************************************/
/* module viterbi_distances                          */
/*****************************************************/
/* This module simulates the front end of a receiver. Normally the
received analog signal (with noise) is converted into a series of
distance measures from the known eight possible transmitted PSK
signals: s0,...,s7. We are not simulating the analog part or noise in
this version, so we just take the digitally encoded 3-bit signal, Y,
from the encoder and convert it directly to the distance measures.
d[N] is the distance from signal = N to signal = 0
d[N] = (2*sin(N*PI/8))**2 in 3-bit binary (on the scale 2=100)
Example: d[3] = 1.85**2 = 3.41 = 110
inN is the distance from signal = N to encoder signal.
Example: in3 is the distance from signal = 3 to encoder signal.
d[N] is the distance from signal = N to encoder signal = 0.
If encoder signal = J, shift the distances by 8-J positions.
Example: if signal = 2, in0 is d[6], in1 is D[7], in2 is D[0], etc. */
module viterbi_distances
  (Y2N,Y1N,Y0N,clk,res,in0,in1,in2,in3,in4,in5,in6,in7);
input clk,res,Y2N,Y1N,Y0N; output in0,in1,in2,in3,in4,in5,in6,in7;
reg [2:0] J,in0,in1,in2,in3,in4,in5,in6,in7; reg [2:0] d [7:0];
```

```
initial begin d[0]='b000;d[1]='b001;d[2]='b100;d[3]='b110;
d[4]='b111;d[5]='b110;d[6]='b100;d[7]='b001; end
always @(Y2N or Y1N or Y0N) begin
J[0]=Y0N;J[1]=Y1N;J[2]=Y2N;
J=8-J;in0=d[J];J=J+1;in1=d[J];J=J+1;in2=d[J];J=J+1;in3=d[J];
J=J+1;in4=d[J];J=J+1;in5=d[J];J=J+1;in6=d[J];J=J+1;in7=d[J];
end endmodule
```

As an example, Table 11.9 shows the distance measures for the transmitted encoder output sequence Y_n = 1, 0, 5, 4, ... (repeated) corresponding to an encoder input of X_n = 0, 1, 2, 3, ... (repeated).

TABLE 11.9 Receiver distance measures for an example transmission sequence.

Time ns	**Input Xn**	Output **Yn**	**Present state**	**Next state**	in0	in1	in2	in3	in4	in5	in6	in7
0	3	x	$S_?$	$S_?$	x	x	x	x	x	x	x	x
10	3	6	S_0	S_1	4	6	7	6	4	1	0	1
50	0	1	S_1	S_2	1	0	1	4	6	7	6	4
150	1	0	S_2	S_1	0	1	4	6	7	6	4	1
250	2	5	S_1	S_2	6	7	6	4	1	0	1	4
350	3	4	S_2	S_1	7	6	4	1	0	1	4	6
450	0	1	S_1	S_2	1	0	1	4	6	7	6	4
550	1	0	S_2	S_1	0	1	4	6	7	6	4	1
650	2	5	S_1	S_2	6	7	6	4	1	0	1	4
750	3	4	S_2	S_1	7	6	4	1	0	1	4	6
850	0	1	S_1	S_2	1	0	1	4	6	7	6	4
950	1	0	S_2	S_1	0	1	4	6	7	6	4	1

11.12.3 Testing the System

Here is a testbench for the entire system: encoder, receiver front end, and decoder:

```
/*****************************************************/
/* module viterbi_test_CDD                           */
/*****************************************************/
/* This is the top-level module, viterbi_test_CDD, that models the
communications link. It contains three modules: viterbi_encode,
viterbi_distances, and viterbi. There is no analog and no noise in
this version. The 2-bit message, X, is encoded to a 3-bit signal, Y.
In this module the message X is generated using a simple counter.
The digital 3-bit signal Y is transmitted, received with noise as an
```

```
analog signal (not modeled here), and converted to a set of eight
3-bit distance measures, in0, ..., in7. The distance measures form
the input to the Viterbi decoder that reconstructs the transmitted
signal Y, with an error signal if the measures are inconsistent.
CDD = counter input, digital transmission, digital reception */
module viterbi_test_CDD;
wire Error;         // decoder out
wire [2:0] Y, Out; // encoder out, decoder out
reg [1:0] X;        // encoder inputs
reg Clk, Res;       // clock and reset
wire [2:0] in0,in1,in2,in3,in4,in5,in6,in7;
always #500 $display("t    Clk X Y Out Error");
initial $monitor("%4g",$time,,Clk,,,,X,,Y,,Out,,,,Error);
initial $dumpvars; initial #3000 $finish;
always #50 Clk = ~Clk;initial begin Clk = 0;
X = 3; // No special reason to start at 3.
#60 Res = 1;#10 Res = 0; end // Hit reset after inputs are stable.
always @(posedge Clk) #1 X = X + 1; // Drive the input with a counter.
viterbi_encode v_1
  (X[1],X[0],Y[2],Y[1],Y[0],Clk,Res);
viterbi_distances v_2
  (Y[2],Y[1],Y[0],Clk,Res,in0,in1,in2,in3,in4,in5,in6,in7);
viterbi v_3
  (in0,in1,in2,in3,in4,in5,in6,in7,Out,Clk,Res,Error);
endmodule
```

The Viterbi decoder takes the distance measures and calculates the most likely transmitted signal. It does this by keeping a running history of the previously received signals in a path memory. The path-memory length of this decoder is 12. By keeping a history of possible sequences and using the knowledge that the signals were generated by a state machine, it is possible to select the most likely sequences.

TABLE 11.10 Output from the Viterbi testbench

t	Clk	X	Y	Out	Error	t	Clk	X	Y	Out	Error
0	0	3	x	x	0	1351	1	1	0	0	0
50	1	3	x	x	0	1400	0	1	0	0	0
51	1	0	x	x	0	1450	1	1	0	0	0
60	1	0	0	0	0	1451	1	2	5	2	0
100	0	0	0	0	0	1500	0	2	5	2	0
150	1	0	0	0	0	1550	1	2	5	2	0
151	1	1	2	0	0	1551	1	3	4	5	0

Table 11.10 shows part of the simulation results from the testbench, viterbi_test_CDD, in tabular form. Figure 11.5 shows the Verilog simulator output from the testbench (displayed using VeriWell from Wellspring).

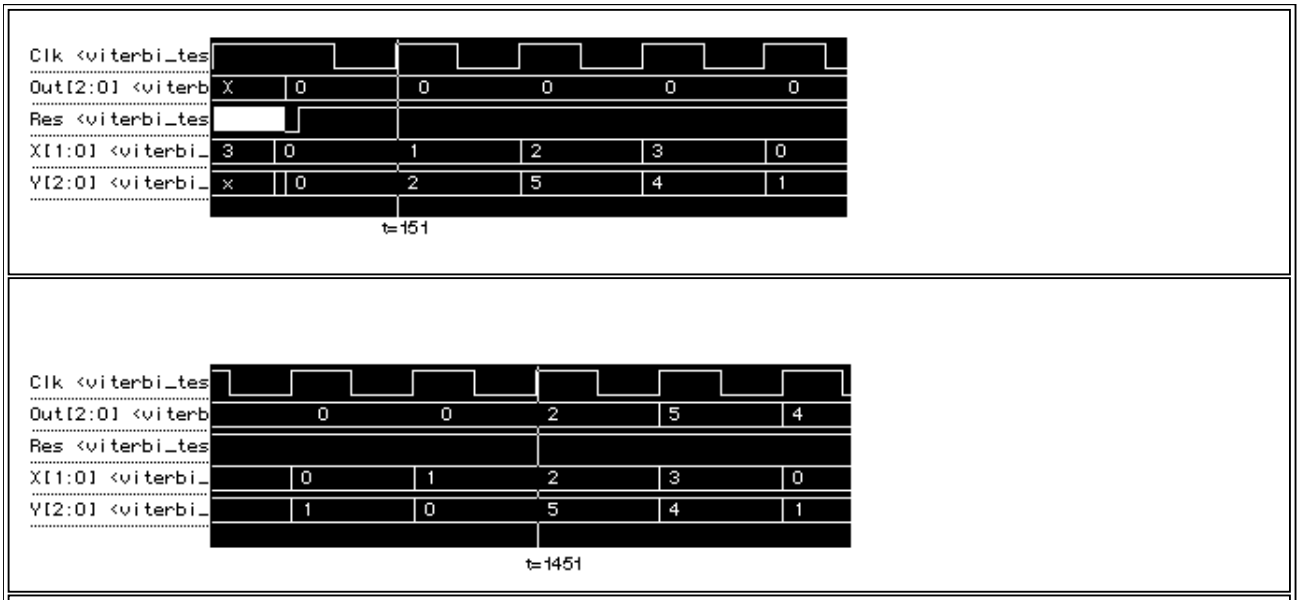

FIGURE 11.5 Viterbi encoder testbench simulation results. (Top) Initialization and the start of the encoder output sequence 2, 5, 4, 1, 0, ... on Y[2:0] at t = 151. (Bottom) The appearance of the same encoder output sequence at the output of the decoder, Out[2:0], at t = 1451, 1300 time units (13 positive clock edges) later.

The system input or message, `X[1:0]` , is driven by a counter that repeats the sequence 0, 1, 2, 3, ... incrementing by 1 at each positive clock edge (with a delay of one time unit), starting with `X` equal to 3 at t = 0. The active-high reset signal, `Res` , is asserted at t = 60 for 10 time units. The encoder output, `Y[2:0]` , changes at t = 151, which is one time unit (the positive-edge-triggered D flip-flop model contains a one-time-unit delay) after the first positive clock edge (at t = 150) following the deassertion of the reset at t = 70. The encoder output sequence beginning at t = 151 is 2, 5, 4, 1, 0, ... and then the sequence 5, 4, 1, 0, ... repeats. This encoder output sequence is then imagined to be transmitted and received. The receiver module calculates the distance measures and passes them to the decoder. After 13 positive clock-edges (1300 time ticks) the transmitted sequence appears at the output, `Out[2:0]` , beginning at t = 1451 with 2, 5, 4, 1, 0, ..., exactly the same as the encoder output.

11.12.4 Verilog Decoder Model

The Viterbi decoder model presented in this section is written for both simulation and synthesis. The Viterbi decoder makes extensive use of vector D flip-flops (registers). Early versions of Verilog-XL did not support vector instantiations of modules. In addition the inputs of UDPs may not be vectors and there are no primitive D flip-flops in Verilog. This makes instantiation of a register difficult other than by writing a separate module instance for each flip-flop.

The first solution to this problem is to use flip-flop models supplied with the synthesis tool such as the following:

```
asDff #(3) subout0(in0, sub0, clk, reset);
```

The asDff is a model in the Compass ASIC Synthesizer standard component library. This statement triggers the synthesis of three D flip-flops, with an input vector `ina` (with a range of three) connected to

the D inputs, an output vector `sub0` (also with a range of three) connected to the Q flip-flop outputs, a common scalar clock signal, `clk` , and a common scalar `reset` signal. The disadvantage of this approach is that the names, functional behavior, and interfaces of the standard components are different for every software system.

The second solution, in new versions of Verilog-XL and other tools that support the IEEE standard, is to use vector instantiation as follows [LRM 7.5.1, 12.1.2]:

```
myDff subout0[0:2] (in0, sub0, clk, reset);
```

This instantiates three copies of a user-defined module or UDP called my `Dff` . The disadvantage of this approach is that not all simulators and synthesizers support vector instantiation.

The third solution (which is used in the Viterbi decoder model) is to write a model that supports vector inputs and outputs. Here is an example D flip-flop model:

```
/*****************************************************/
/*       module dff                                  */
/*****************************************************/
/* A D flip-flop module. */
module dff(D,Q,Clock,Reset); // N.B. reset is active-low.
output Q; input D,Clock,Reset;
parameter CARDINALITY = 1; reg [CARDINALITY-1:0] Q;
wire [CARDINALITY-1:0] D;
always @(posedge Clock) if (Reset !== 0) #1 Q = D;
always begin wait (Reset == 0); Q = 0; wait (Reset == 1); end
endmodule
```

We use this model by defining a parameter that specifies the bus width as follows:

dff #(3) subout0(in0, sub0, clk, reset);

The code that models the entire Viterbi decoder is listed below (Figure 12.6 on page 578 shows the block digram). Notice the following:

- Comments explain the function of each module.
- Each module is about a page or less of code.
- Each module can be tested by itself.
- The code is as simple as possible avoiding clever coding techniques.

The code is not flexible, because bit widths are fixed rather than using parameters. A model with parameters for rate, signal constellation, distance measure resolution, and path memory length is considerably more complex. We shall use this Viterbi decoder design again when we discuss logic synthesis in Chapter 12, test in Chapter 14, floorplanning and placement in Chapter 16, and routing in Chapter 17.

```
/* Verilog code for a Viterbi decoder. The decoder assumes a rate
2/3 encoder, 8 PSK modulation, and trellis coding. The viterbi module
contains eight submodules: subset_decode, metric, compute_metric,
compare_select, reduce, pathin, path_memory, and output_decision.
  The decoder accepts eight 3-bit measures of ||r-si||**2 and, after
an initial delay of thirteen clock cycles, the output is the best
estimate of the signal transmitted. The distance measures are the
```

```
Euclidean distances between the received signal r (with noise) and
each of the (in this case eight) possible transmitted signals s0 to s7.
  Original by Christeen Gray, University of Hawaii. Heavily modified
by MJSS; any errors are mine. Use freely. */
/*****************************************************/
/*   module viterbi                                  */
/*****************************************************/
/* This is the top level of the Viterbi decoder. The eight input
signals {in0,...,in7} represent the distance measures, ||r-si||**2.
The other input signals are clk and reset. The output signals are
out and error. */
module viterbi
    (in0,in1,in2,in3,in4,in5,in6,in7,
    out,clk,reset,error);
input [2:0] in0,in1,in2,in3,in4,in5,in6,in7;
output [2:0] out; input clk,reset; output error;
wire sout0,sout1,sout2,sout3;
wire [2:0] s0,s1,s2,s3;
wire [4:0] m_in0,m_in1,m_in2,m_in3;
wire [4:0] m_out0,m_out1,m_out2,m_out3;
wire [4:0] p0_0,p2_0,p0_1,p2_1,p1_2,p3_2,p1_3,p3_3;
wire ACS0,ACS1,ACS2,ACS3;
wire [4:0] out0,out1,out2,out3;
wire [1:0] control;
wire [2:0] p0,p1,p2,p3;
wire [11:0] path0;
  subset_decode u1(in0,in1,in2,in3,in4,in5,in6,in7,
    s0,s1,s2,s3,sout0,sout1,sout2,sout3,clk,reset);
  metric u2(m_in0,m_in1,m_in2,m_in3,m_out0,
    m_out1,m_out2,m_out3,clk,reset);
  compute_metric u3(m_out0,m_out1,m_out2,m_out3,s0,s1,s2,s3,
    p0_0,p2_0,p0_1,p2_1,p1_2,p3_2,p1_3,p3_3,error);
  compare_select u4(p0_0,p2_0,p0_1,p2_1,p1_2,p3_2,p1_3,p3_3,
    out0,out1,out2,out3,ACS0,ACS1,ACS2,ACS3);
  reduce u5(out0,out1,out2,out3,
    m_in0,m_in1,m_in2,m_in3,control);
  pathin u6(sout0,sout1,sout2,sout3,
    ACS0,ACS1,ACS2,ACS3,path0,clk,reset);
  path_memory u7(p0,p1,p2,p3,path0,clk,reset,
    ACS0,ACS1,ACS2,ACS3);
  output_decision u8(p0,p1,p2,p3,control,out);
endmodule
/*****************************************************/
/* module subset_decode                              */
/*****************************************************/
/* This module chooses the signal corresponding to the smallest of
each set {||r-s0||**2,||r-s4||**2}, {||r-s1||**2, ||r-s5||**2},
{||r-s2||**2,||r-s6||**2}, {||r-s3||**2,||r-s7||**2}. Therefore
there are eight input signals and four output signals for the
distance measures. The signals sout0, ..., sout3 are used to control
the path memory. The statement dff #(3) instantiates a vector array
of 3 D flip-flops. */
module subset_decode
    (in0,in1,in2,in3,in4,in5,in6,in7,
    s0,s1,s2,s3,
    sout0,sout1,sout2,sout3,
    clk,reset);
input [2:0] in0,in1,in2,in3,in4,in5,in6,in7;
output [2:0] s0,s1,s2,s3;
output sout0,sout1,sout2,sout3;
input clk,reset;
```

```
wire [2:0] sub0,sub1,sub2,sub3,sub4,sub5,sub6,sub7;
  dff #(3) subout0(in0, sub0, clk, reset);
  dff #(3) subout1(in1, sub1, clk, reset);
  dff #(3) subout2(in2, sub2, clk, reset);
  dff #(3) subout3(in3, sub3, clk, reset);
  dff #(3) subout4(in4, sub4, clk, reset);
  dff #(3) subout5(in5, sub5, clk, reset);
  dff #(3) subout6(in6, sub6, clk, reset);
  dff #(3) subout7(in7, sub7, clk, reset);
  function [2:0] subset_decode; input [2:0] a,b;
    begin
      subset_decode = 0;
      if (a<=b) subset_decode = a; else subset_decode = b;
    end
  endfunction
  function set_control; input [2:0] a,b;
    begin
      if (a<=b) set_control = 0; else set_control = 1;
    end
  endfunction
assign s0 = subset_decode (sub0,sub4);
assign s1 = subset_decode (sub1,sub5);
assign s2 = subset_decode (sub2,sub6);
assign s3 = subset_decode (sub3,sub7);
assign sout0 = set_control(sub0,sub4);
assign sout1 = set_control(sub1,sub5);
assign sout2 = set_control(sub2,sub6);
assign sout3 = set_control(sub3,sub7);
endmodule
/*****************************************************/
/*   module compute_metric                           */
/*****************************************************/
/* This module computes the sum of path memory and the distance for
each path entering a state of the trellis. For the four states,
there are two paths entering it; therefore eight sums are computed
in this module. The path metrics and output sums are 5 bits wide.
The output sum is bounded and should never be greater than 5 bits
for a valid input signal. The overflow from the sum is the error
output and indicates an invalid input signal.*/
module compute_metric
    (m_out0,m_out1,m_out2,m_out3,
    s0,s1,s2,s3,p0_0,p2_0,
    p0_1,p2_1,p1_2,p3_2,p1_3,p3_3,
    error);
  input [4:0] m_out0,m_out1,m_out2,m_out3;
  input [2:0] s0,s1,s2,s3;
  output [4:0] p0_0,p2_0,p0_1,p2_1,p1_2,p3_2,p1_3,p3_3;
  output error;
  assign
    p0_0 = m_out0 + s0,
    p2_0 = m_out2 + s2,
    p0_1 = m_out0 + s2,
    p2_1 = m_out2 + s0,
    p1_2 = m_out1 + s1,
    p3_2 = m_out3 + s3,
    p1_3 = m_out1 + s3,
    p3_3 = m_out3 + s1;
  function is_error; input x1,x2,x3,x4,x5,x6,x7,x8;
  begin
    if (x1||x2||x3||x4||x5||x6||x7||x8) is_error = 1;
    else is_error = 0;
```

```
  end
  endfunction
  assign error = is_error(p0_0[4],p2_0[4],p0_1[4],p2_1[4],
    p1_2[4],p3_2[4],p1_3[4],p3_3[4]);
endmodule
/*****************************************************/
/*   module compare_select                           */
/*****************************************************/
/* This module compares the summations from the compute_metric
module and selects the metric and path with the lowest value. The
output of this module is saved as the new path metric for each
state. The ACS output signals are used to control the path memory of
the decoder. */
module compare_select
    (p0_0,p2_0,p0_1,p2_1,p1_2,p3_2,p1_3,p3_3,
    out0,out1,out2,out3,
    ACS0,ACS1,ACS2,ACS3);
  input [4:0] p0_0,p2_0,p0_1,p2_1,p1_2,p3_2,p1_3,p3_3;
  output [4:0] out0,out1,out2,out3;
  output ACS0,ACS1,ACS2,ACS3;
  function [4:0] find_min_metric; input [4:0] a,b;
    begin
      if (a <= b) find_min_metric = a; else find_min_metric = b;
    end
  endfunction
  function set_control; input [4:0] a,b;
    begin
      if (a <= b) set_control = 0; else set_control = 1;
    end
  endfunction
assign out0 = find_min_metric(p0_0,p2_0);
assign out1 = find_min_metric(p0_1,p2_1);
assign out2 = find_min_metric(p1_2,p3_2);
assign out3 = find_min_metric(p1_3,p3_3);
assign ACS0 = set_control (p0_0,p2_0);
assign ACS1 = set_control (p0_1,p2_1);
assign ACS2 = set_control (p1_2,p3_2);
assign ACS3 = set_control (p1_3,p3_3);
endmodule
/*****************************************************/
/*   module path                                     */
/*****************************************************/
/* This is the basic unit for the path memory of the Viterbi
decoder. It consists of four 3-bit D flip-flops in parallel. There
is a 2:1 mux at each D flip-flop input. The statement dff #(12)
instantiates a vector array of 12 flip-flops. */
module path(in,out,clk,reset,ACS0,ACS1,ACS2,ACS3);
input [11:0] in; output [11:0] out;
input clk,reset,ACS0,ACS1,ACS2,ACS3; wire [11:0] p_in;
dff #(12) path0(p_in,out,clk,reset);
  function [2:0] shift_path; input [2:0] a,b; input control;
    begin
      if (control == 0) shift_path = a; else shift_path = b;
    end
  endfunction
assign p_in[11:9] = shift_path(in[11:9],in[5:3],ACS0);
assign p_in[ 8:6] = shift_path(in[11:9],in[5:3],ACS1);
assign p_in[ 5:3] = shift_path(in[8: 6],in[2:0],ACS2);
assign p_in[ 2:0] = shift_path(in[8: 6],in[2:0],ACS3);
endmodule
/*****************************************************/
```

```
/*   module path_memory                                    */
/*********************************************************/
/* This module consists of an array of memory elements (D
flip-flops) that store and shift the path memory as new signals are
added to the four paths (or four most likely sequences of signals).
This module instantiates 11 instances of the path module. */
module path_memory
    (p0,p1,p2,p3,
    path0,clk,reset,
    ACS0,ACS1,ACS2,ACS3);
output [2:0] p0,p1,p2,p3; input [11:0] path0;
input clk,reset,ACS0,ACS1,ACS2,ACS3;
wire [11:0]out1,out2,out3,out4,out5,out6,out7,out8,out9,out10,out11;
    path x1 (path0,out1 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
         x2 (out1, out2 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
         x3 (out2, out3 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
         x4 (out3, out4 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
         x5 (out4, out5 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
         x6 (out5, out6 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
         x7 (out6, out7 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
         x8 (out7, out8 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
         x9 (out8, out9 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
         x10(out9, out10,clk,reset,ACS0,ACS1,ACS2,ACS3),
         x11(out10,out11,clk,reset,ACS0,ACS1,ACS2,ACS3);
assign p0 = out11[11:9];
assign p1 = out11[ 8:6];
assign p2 = out11[ 5:3];
assign p3 = out11[ 2:0];
endmodule
/*********************************************************/
/*   module pathin                                       */
/*********************************************************/
/* This module determines the input signal to the path for each of
the four paths. Control signals from the subset decoder and compare
select modules are used to store the correct signal. The statement
dff #(12) instantiates a vector array of 12 flip-flops. */
module pathin
    (sout0,sout1,sout2,sout3,
    ACS0,ACS1,ACS2,ACS3,
    path0,clk,reset);
  input sout0,sout1,sout2,sout3,ACS0,ACS1,ACS2,ACS3;
  input clk,reset; output [11:0] path0;
  wire [2:0] sig0,sig1,sig2,sig3; wire [11:0] path_in;
  dff #(12) firstpath(path_in,path0,clk,reset);
  function [2:0] subset0; input sout0;
    begin
      if(sout0 == 0) subset0 = 0; else subset0 = 4;
    end
  endfunction
  function [2:0] subset1; input sout1;
    begin
      if(sout1 == 0) subset1 = 1; else subset1 = 5;
    end
  endfunction
  function [2:0] subset2; input sout2;
    begin
      if(sout2 == 0) subset2 = 2; else subset2 = 6;
    end
  endfunction
  function [2:0] subset3; input sout3;
    begin
```

```
      if(sout3 == 0) subset3 = 3; else subset3 = 7;
    end
  endfunction
  function [2:0] find_path; input [2:0] a,b; input control;
    begin
      if(control==0) find_path = a; else find_path = b;
    end
  endfunction
assign sig0 = subset0(sout0);
assign sig1 = subset1(sout1);
assign sig2 = subset2(sout2);
assign sig3 = subset3(sout3);
assign path_in[11:9] = find_path(sig0,sig2,ACS0);
assign path_in[ 8:6] = find_path(sig2,sig0,ACS1);
assign path_in[ 5:3] = find_path(sig1,sig3,ACS2);
assign path_in[ 2:0] = find_path(sig3,sig1,ACS3);
endmodule
/*****************************************************/
/*   module metric                                   */
/*****************************************************/
/* The registers created in this module (using D flip-flops) store
the four path metrics. Each register is 5 bits wide. The statement
dff #(5) instantiates a vector array of 5 flip-flops. */
module metric
    (m_in0,m_in1,m_in2,m_in3,
    m_out0,m_out1,m_out2,m_out3,
    clk,reset);
input [4:0] m_in0,m_in1,m_in2,m_in3;
output [4:0] m_out0,m_out1,m_out2,m_out3;
input clk,reset;
  dff #(5) metric3(m_in3, m_out3, clk, reset);
  dff #(5) metric2(m_in2, m_out2, clk, reset);
  dff #(5) metric1(m_in1, m_out1, clk, reset);
  dff #(5) metric0(m_in0, m_out0, clk, reset);
endmodule
/*****************************************************/
/*   module output_decision                          */
/*****************************************************/
/* This module decides the output signal based on the path that
corresponds to the smallest metric. The control signal comes from
the reduce module. */
module output_decision(p0,p1,p2,p3,control,out);
  input [2:0] p0,p1,p2,p3; input [1:0] control; output [2:0] out;
  function [2:0] decide;
  input [2:0] p0,p1,p2,p3; input [1:0] control;
  begin
    if(control == 0) decide = p0;
    else if(control == 1) decide = p1;
    else if(control == 2) decide = p2;
    else decide = p3;
    end
  endfunction
assign out = decide(p0,p1,p2,p3,control);
endmodule
/*****************************************************/
/*   module reduce                                   */
/*****************************************************/
/* This module reduces the metrics after the addition and compare
operations. This algorithm selects the smallest metric and subtracts
it from all the other metrics. */
module reduce
```

```
    (in0,in1,in2,in3,
    m_in0,m_in1,m_in2,m_in3,
    control);
  input [4:0] in0,in1,in2,in3;
  output [4:0] m_in0,m_in1,m_in2,m_in3;
  output [1:0] control; wire [4:0] smallest;
  function [4:0] find_smallest;
    input [4:0] in0,in1,in2,in3; reg [4:0] a,b;
      begin
        if(in0 <= in1) a = in0; else a = in1;
        if(in2 <= in3) b = in2; else b = in3;
        if(a <= b) find_smallest = a;
        else find_smallest = b;
      end
  endfunction
  function [1:0] smallest_no;
  input [4:0] in0,in1,in2,in3,smallest;
    begin
      if(smallest == in0) smallest_no = 0;
      else if (smallest == in1) smallest_no = 1;
      else if (smallest == in2) smallest_no = 2;
      else smallest_no = 3;
    end
  endfunction
assign smallest = find_smallest(in0,in1,in2,in3);
assign m_in0 = in0 - smallest;
assign m_in1 = in1 - smallest;
assign m_in2 = in2 - smallest;
assign m_in3 = in3 - smallest;
assign control = smallest_no(in0,in1,in2,in3,smallest);
endmodule
```

Chapter start

Previous page

Previous page

Next page

11.13 Other Verilog Features

This section covers some of the more advanced Verilog features. **System tasks** and functions are defined as part of the IEEE Verilog standard [Verilog LRM14].

11.13.1 Display Tasks

The following code illustrates the **display system tasks** [Verilog LRM 14.1]:

```
module test_display; // display system tasks:
initial begin $display ("string, variables, or expression");
/* format specifications work like printf in C:
        %d=decimal %b=binary %s=string %h=hex %o=octal
        %c=character %m=hierarchical name %v=strength %t=time format
        %e=scientific %f=decimal %g=shortest
examples: %d uses default width %0d uses minimum width
        %7.3g uses 7 spaces with 3 digits after decimal point */
// $displayb, $displayh, $displayo print in b, h, o formats
// $write, $strobe, $monitor also have b, h, o versions
$write("write"); // as $display, but without newline at end of line
$strobe("strobe"); // as $display, values at end of simulation cycle
$monitor(v); // disp. @change of v (except v= $time,$stime,$realtime)
$monitoron; $monitoroff; // toggle monitor mode on/off
end endmodule
```

11.13.2 File I/O Tasks

The following example illustrates the **file I/O system tasks** [Verilog LRM 14.2]:

```
module file_1; integer f1, ch; initial begin f1 = $fopen("f1.out");
if(f1==0) $stop(2); if(f1==2)$display("f1 open");
ch = f1|1; $fdisplay(ch,"Hello"); $fclose(f1); end endmodule
> vlog file_1.v
> vsim -c file_1
# Loading work.file_1
VSIM 1> run 10
# f1 open
# Hello
VSIM 2> q
> more f1.out
Hello
>
```

The `$fopen` system task returns a 32-bit unsigned integer called a **multichannel descriptor** (`f1` in this example) unique to each file. The multichannel descriptor contains 32 flags, one for each of 32 possible channels or files (subject to limitations of the operating system). Channel 0 is the standard output (normally the screen), which is always open. The first call to `$fopen` opens channel 1 and sets bit 1 of the multichannel descriptor. Subsequent calls set higher bits. The file I/O system tasks: `$fdisplay` , `$fwrite` , `$fmonitor` , and `$fstrobe` ; correspond to their display counterparts. The first parameter for the file system tasks is a multichannel descriptor that may have multiple bits set. Thus, the preceding example writes the string `"Hello"` to the screen and to `file1.out` . The task $fclose closes a file and allows the channel to be reused.

The file I/O tasks `$readmemb` and `$readmemh` read a text file into a memory. The file may contain only spaces, new lines, tabs, form feeds, comments, addresses, and binary (for `$readmemb`) or hex (for `$readmemh`) numbers, as in the following example:

```
mem.dat
@2 1010_1111 @4 0101_1111 1010_1111 // @address in hex
x1x1_zzzz 1111_0000 /* x or z is OK */
module load; reg [7:0] mem[0:7]; integer i; initial begin
$readmemb("mem.dat", mem, 1, 6); // start_address=1, end_address=6
for (i= 0; i<8; i=i+1) $display("mem[%0d] %b", i, mem[i]);
end endmodule
> vsim -c load
```

```
# Loading work.load
VSIM 1> run 10
# ** Warning: $readmem (memory mem) file mem.dat line 2:
#    More patterns than index range (hex 1:6)
#    Time: 0 ns  Iteration: 0  Instance:/
# mem[0] xxxxxxxx
# mem[1] xxxxxxxx
# mem[2] 10101111
# mem[3] xxxxxxxx
# mem[4] 01011111
# mem[5] 10101111
# mem[6] x1x1zzzz
# mem[7] xxxxxxxx
VSIM 2> q
>
```

11.13.3 Timescale, Simulation, and Timing-Check Tasks

There are two **timescale tasks**, `$printtimescale` and `$timeformat` [Verilog LRM 14.3]. The `$timeformat` specifies the `%t` format specification for the display and file I/O system tasks as well as the time unit for delays entered interactively and from files. Here are examples of the timescale tasks:

```
// timescale tasks:
module a; initial $printtimescale(b.c1); endmodule
module b; c c1 (); endmodule
‘timescale 10 ns / 1 fs
module c_dat; endmodule
‘timescale 1 ms / 1 ns
module Ttime; initial $timeformat(-9, 5, " ns", 10); endmodule
/* $timeformat [ ( n, p, suffix , min_field_width ) ] ;
units = 1 second ** (-n), n = 0->15, e.g. for n = 9, units = ns
p = digits after decimal point for %t e.g. p = 5 gives 0.00000
suffix for %t (despite timescale directive)
min_field_width is number of character positions for %t */
```

The **simulation control tasks** are $stop and $finis `h` [Verilog LRM 14.4]:

```
module test_simulation_control; // simulation control system tasks:
initial begin $stop; // enter interactive mode (default parameter 1)
$finish(2); // graceful exit with optional parameter as follows:
// 0 = nothing 1 = time and location 2 = time, location, and statistics
end endmodule
```

The **timing-check tasks** [Verilog LRM 14.5] are used in specify blocks. The following code and comments illustrate the definitions and use of timing-check system tasks. The arguments to the tasks are defined and explained in Table 11.11.

TABLE 11.11 Timing-check system task parameters.		
Timing task argument	**Description of argument**	**Type of argument**
`reference_event`	to establish reference time	module input or inout (scalar or vector net)
`data_event`	signal to check against reference_event	module input or inout (scalar or vector net)
`limit`	time limit to detect timing violation on data_event	constant expression or specparam
`threshold`	largest pulse width ignored by timing check $width	constant expression or specparam
`notifier`	flags a timing violation (before -> after): `x->0, 0->1, 1->0, z->z`	register

```
module timing_checks (data, clock, clock_1,clock_2);
input data,clock,clock_1,clock_2; reg tSU,tH,tHIGH,tP,tSK,tR;
specify // timing check system tasks:
/* $setup (data_event, reference_event, limit [, notifier]);
violation = (T_reference_event)-(T_data_event) < limit */
$setup(data, posedge clock, tSU);
/* $hold (reference_event, data_event, limit [, notifier]);
violation =
  (time_of_data_event)-(time_of_reference_event) < limit */
$hold(posedge clock, data, tH);
/* $setuphold (reference_event, data_event, setup_limit,
    hold_limit [, notifier]);
parameter_restriction = setup_limit + hold_limit > 0 */
$setuphold(posedge clock, data, tSU, tH);
/* $width (reference_event, limit, threshold [, notifier]);
violation =
  threshold < (T_data_event) - (T_reference_event) < limit
reference_event = edge
data_event = opposite_edge_of_reference_event */
$width(posedge clock, tHIGH);
/* $period (reference_event, limit [, notifier]);
violation = (T_data_event) - (T_reference_event) < limit
reference_event = edge
data_event = same_edge_of_reference event */
$period(posedge clock, tP);
/* $skew (reference_event, data_event, limit [, notifier]);
violation = (T_data_event) - (T_reference_event) > limit */
$skew(posedge clock_1, posedge clock_2, tSK);
/* $recovery (reference_event, data_event, limit, [, notifier]);
violation = (T_data_event) - (T_reference_event) < limit */
$recovery(posedge clock, posedge clock_2, tR);
/* $nochange (reference_event, data_event, start_edge_offset,
  end_edge_offset [, notifier]);
reference_event = posedge | negedge
```

```
violation = change while reference high (posedge)/low (negedge)
+ve start_edge_offset moves start of window later
+ve end_edge_offset moves end of window later */
$nochange (posedge clock, data, 0, 0);
endspecify endmodule
```

You can use **edge specifiers** as parameters for the timing-check events (except for the reference event in $nochange):

```
edge_control_specifier ::= edge [edge_descriptor {, edge_descriptor}]
edge_descriptor ::= 01 | 0x | 10 | 1x | x0 | x1
```

For example, `'edge [01, 0x, x1] clock'` is equivalent to `'posedge clock'` . Edge transitions with `'z'` are treated the same as transitions with `'x'` .

Here is a D flip-flop model that uses timing checks and a **notifier register**. The register, notifier, is changed when a timing-check task detects a violation and the last entry in the table then sets the flip-flop output to unknown.

```
primitive dff_udp(q, clock, data, notifier);
output q; reg q; input clock, data, notifier;
table //   clock data  notifier:state:  q
           r     0     ?      :  ?  :  0 ;
           r     1     ?      :  ?  :  1 ;
           n     ?     ?      :  ?  :  - ;
           ?     *     ?      :  ?  :  - ;
           ?     ?     *      :  ?  :  x ; endtable // notifier
endprimitive
‘timescale 100 fs / 1 fs
module dff(q, clock, data); output q; input clock, data; reg notifier;
dff_udp(q1, clock, data, notifier); buf(q, q1);
specify
  specparam tSU = 5, tH = 1, tPW = 20, tPLH = 4:5:6, tPHL = 4:5:6;
     (clock *> q) = (tPLH, tPHL);
  $setup(data, posedge clock, tSU, notifier); // setup: data to clock
  $hold(posedge clock, data, tH, notifier); // hold: clock to data
  $period(posedge clock, tPW, notifier); // clock: period
endspecify
endmodule
```

11.13.4 PLA Tasks

The **PLA modeling tasks** model two-level logic [Verilog LRM 14.6]. As an example, the following `eqntott` logic equations can be implemented using a PLA:

```
b1 = a1 & a2; b2 = a3 & a4 & a5 ; b3 = a5 & a6 & a7;
```

The following module loads a PLA model for the equations above (in AND logic) using the **array format** (the array format allows only `'1'` or `'0'` in the PLA memory, or **personality array**). The file `array.dat` is similar to the `espresso` input plane format.

```
array.dat
1100000
0011100
0000111
```

```
module pla_1 (a1,a2,a3,a4,a5,a6,a7,b1,b2,b3);
input a1, a2, a3, a4, a5, a6, a7 ; output b1, b2, b3;
reg [1:7] mem[1:3]; reg b1, b2, b3;
initial begin
  $readmemb("array.dat", mem);
  #1; b1=1; b2=1; b3=1;
  $async$and$array(mem,{a1,a2,a3,a4,a5,a6,a7},{b1,b2,b3});
end
initial $monitor("%4g",$time,,b1,,b2,,b3);
endmodule
```

The next example illustrates the use of the **plane format**, which allows `'1'` , `'0'` , as well as `'?'` or `'z'` (either may be used for don't care) in the personality array.

```
b1 = a1 & !a2; b2 = a3; b3 = !a1 & !a3; b4 = 1;
module pla_2; reg [1:3] a, mem[1:4]; reg [1:4] b;
initial begin
  $async$and$plane(mem,{a[1],a[2],a[3]},{b[1],b[2],b[3],b[4]});
  mem[1] = 3'b10?; mem[2] = 3'b??1; mem[3] = 3'b0?0; mem[4] = 3'b???;
  #10 a = 3'b111; #10 $displayb(a, " -> ", b);
  #10 a = 3'b000; #10 $displayb(a, " -> ", b);
  #10 a = 3'bxxx; #10 $displayb(a, " -> ", b);
  #10 a = 3'b101; #10 $displayb(a, " -> ", b);
end endmodule
111 -> 0101
000 -> 0011
xxx -> xxx1
101 -> 1101
```

11.13.5 Stochastic Analysis Tasks

The **stochastic analysis tasks** model queues [Verilog LRM 14.7]. Each of the tasks return a status as shown in Table 11.12.

TABLE 11.12 Status values for the stochastic analysis tasks.

Status value	Meaning
0	OK
1	queue full, cannot add
2	undefined `q_id`
3	queue empty, cannot remove
4	unsupported `q_type` , cannot create queue
5	`max_length` <= 0, cannot create queue
6	duplicate `q_id` , cannot create queue
7	not enough memory, cannot create queue

The following module illustrates the interface and parameters for these tasks:

```
module stochastic; initial begin // stochastic analysis system tasks:
/* $q_initialize (q_id, q_type, max_length, status) ;
```

```
q_id is an integer that uniquely identifies the queue
q_type 1=FIFO 2=LIFO
max_length is an integer defining the maximum number of entries */
$q_initialize (q_id, q_type, max_length, status) ;
/* $q_add (q_id, job_id, inform_id, status) ;
job_id = integer input
inform_id = user-defined integer input for queue entry */
$q_add (q_id, job_id, inform_id, status) ;
/* $q_remove (q_id, job_id, inform_id, status) ; */
$q_remove (q_id, job_id, inform_id, status) ;
/* $q_full (q_id, status) ;
status = 0 = queue is not full, status = 1 = queue full */
$q_full (q_id, status) ;
/* $q_exam (q_id, q_stat_code, q_stat_value, status) ;
q_stat_code is input request as follows:
1=current queue length 2=mean inter-arrival time 3=max. queue length
4=shortest wait time ever
5=longest wait time for jobs still in queue 6=ave. wait time in queue
q_stat_value is output containing requested value */
$q_exam (q_id, q_stat_code, q_stat_value, status) ;
end endmodule
```

11.13.6 Simulation Time Functions

The **simulation time functions** return the time as follows [Verilog LRM 14.8]:

```
module test_time; initial begin // simulation time system functions:
$time ;
// returns 64-bit integer scaled to timescale unit of invoking module
$stime ;
// returns 32-bit integer scaled to timescale unit of invoking module
$realtime ;
// returns real scaled to timescale unit of invoking module
end endmodule
```

11.13.7 Conversion Functions

The **conversion functions for reals** handle real numbers [Verilog LRM 14.9]:

```
module test_convert; // conversion functions for reals:
integer i; real r; reg [63:0] bits;
initial begin #1 r=256;#1 i = $rtoi(r);
#1; r = $itor(2 * i) ; #1 bits = $realtobits(2.0 * r) ;
#1; r = $bitstoreal(bits) ; end
initial $monitor("%3f",$time,,i,,r,,bits); /*
$rtoi converts reals to integers w/truncation e.g. 123.45 -> 123
$itor converts integers to reals e.g. 123 -> 123.0
$realtobits converts reals to 64-bit vector
$bitstoreal converts bit pattern to real
Real numbers in these functions conform to IEEE Std 754. Conversion rounds to the ne
endmodule
# 0.000000           x 0                    x
# 1.000000           x 256                    x
# 2.000000         256 256                    x
# 3.000000         256 512                    x
# 4.000000         256 512  4652218415073722368
# 5.000000         256 1024  465221841507372368
```

Here is an example using the conversion functions in port connections:

```
module test_real;wire [63:0]a; driver d (a); receiver r (a);
initial $monitor("%3g",$time,,a,,d.r1,,r.r2); endmodule
module driver (real_net);
output real_net; real r1; wire [64:1] real_net = $realtobits(r1);
initial #1 r1 = 123.456; endmodule
module receiver (real_net);
input real_net; wire [64:1] real_net; real r2;
initial assign r2 = $bitstoreal(real_net);
endmodule
# 0                     0 0 0
# 1  4638387860618067575 123.456 123.456
```

11.13.8 Probability Distribution Functions

The probability distribution functions are as follows [Verilog LRM 14.10]:

```
module probability; // probability distribution functions:
/* $random [ ( seed ) ] returns random 32-bit signed integer
seed = register, integer, or time */
reg [23:0] r1,r2; integer r3,r4,r5,r6,r7,r8,r9;
integer seed, start, \end , mean, standard_deviation;
integer degree_of_freedom, k_stage;
initial begin seed=1; start=0; \end =6; mean=5;
standard_deviation=2; degree_of_freedom=2; k_stage=1; #1;
r1 = $random % 60; // random -59 to 59
r2 = {$random} % 60; // positive value 0-59
r3=$dist_uniform (seed, start, \end ) ;
r4=$dist_normal (seed, mean, standard_deviation) ;
r5=$dist_exponential (seed, mean) ;
r6=$dist_poisson (seed, mean) ;
r7=$dist_chi_square (seed, degree_of_freedom) ;
r8=$dist_t (seed, degree_of_freedom) ;
r9=$dist_erlang (seed, k_stage, mean) ; end
initial #2 $display ("%3f",$time,,r1,,r2,,r3,,r4,,r5);
initial begin #3; $display ("%3f",$time,,r6,,r7,,r8,,r9); end
/* All parameters are integer values.
Each function returns a pseudo-random number
e.g. $dist_uniform returns uniformly distributed random numbers
mean, degree_of_freedom, k_stage
(exponential, poisson, chi-square, t, erlang) > 0.
seed = inout integer initialized by user, updated by function
start, end ($dist_uniform) = integer bounding return values */
endmodule
2.000000        8       57           0           4           9
3.000000           7           3           0           2
```

11.13.9 Programming Language Interface

The C language **Programming Language Interface** (**PLI**) allows you to access the internal Verilog data structure [Verilog LRM17-23, A-E]. For example, you can use the PLI to implement the following extensions to a Verilog simulator:

- C language delay calculator for a cell library
- C language interface to a Verilog-based or other logic or fault simulator

- Graphical waveform display and debugging
- C language simulation models
- Hardware interfaces

There are three generations of PLI routines (see Appendix B for an example):

- Task/function (TF) routines (or utility routines), the first generation of the PLI, start with `'tf_'`.
- Access (ACC) routines, the second generation of the PLI, start with the characters `'acc_'` and access delay and logic values. There is some overlap between the ACC routines and TF routines.
- Verilog Procedural Interface (VPI) routines, the third generation of the PLI, start with the characters `'vpi_'` and are a superset of the TF and ACC routines.

Chapter start

Previous page

Previous page

Next page

11.14 Summary

Table 11.13 lists the key features of Verilog HDL. The most important concepts covered in this chapter are:

TABLE 11.13 Verilog on one page.

Verilog feature	Example

Comments	`a = 0; // comment ends with newline`
	`/* This is a multiline or block comment */`

Constants: string and numeric	`parameter BW = 32 // local, use BW`
	`‘define G_BUS 32 // global, use ‘G_BUS`
	`4’b2  1’bx`

Names (case-sensitive, start with letter or ’_’)	`_12name  A_name  $BAD  NotSame  notsame`
Two basic types of logic signals: wire and reg	`wire myWire; reg myReg;`
Use a continuous assignment statement with wire	`assign myWire = 1;`
Use a procedural assignment statement with reg	`always myReg = myWire;`

Buses and vectors use square brackets	`reg [31:0] DBus; DBus[12] = 1'bx;`
We can perform arithmetic on bit vectors	`reg [31:0] DBus; DBus = DBus + 2;`
Arithmetic is performed modulo 2^n	`reg [2:0] R; R = 7 + 1; // now R = 0`
Operators: as in C (but not ++ or - -)	
Fixed logic-value system	`1, 0, x (unknown), z (high-impedance)`

Basic unit of code is the module	`module bake (chips, dough, cookies);`
	`input chips, dough; output cookies;`
	`assign cookies = chips & dough;`
	`endmodule`

Ports	input or input/output ports are wire output ports are wire or reg

Procedures model things that happen at the same time
 	
and may be sensitive to an edge, **posedge**, **negedge**,
 	
or to a level.
	`always @rain sing; always @rain dance;`
	`always @(posedge clock) D = Q; // flop`
	`always @(a or b) c = a & b; // and gate`

Sequential blocks model repeating things:
 	
always: executes forever
 	
initial: executes once only at start of simulation
	`initial born;`
	`always @alarm_clock begin : a_day`
	`metro=commute; bulot=work; dodo=sleep;`
	`end`

Functions and tasks	`function ... endfunction` `task ... endtask`
Output	`$display("a=%f",a);$dumpvars;$monitor(a)`
Control simulation	`$stop; $finish // sudden or gentle halt`
Compiler directives	'timescale 1ns/1ps // units/resolution

Delay	#1 a = b; // delay then sample b
	
	a = #1 b; // sample b then delay

- Concurrent processes and sequential execution
- Difference between a `reg` and a `wire` , and between a scalar and a vector
- Arithmetic operations on `reg` and `wire`
- Data slip
- Delays and events

11.15 Problems

* = Difficult, ** = Very difficult, *** = Extremely difficult

11.1 (Counter, 30 min.) Download the VeriWell simulator from `http://www. wellspring.com` and simulate the counter from Section 11.1 (exclude the comments to save typing). Include the complete input and output listings in your report.

11.2 (Simulator, 30 min.) Build a "cheat sheet" for your simulator, listing the commands for running the simulator and using it in interactive mode.

11.3 (Verilog examples, 10 min.) The Cadence Verilog-XL simulator comes with a directory `examples` . Make a list of the examples from the `README` files in the various directories.

11.4 (Gotchas, 60 min.) Build a "most common Verilog mistakes" file. Start with:

- Extra or missing semicolon ’;’
- Forgetting to declare a `reg`
- Using a `reg` instead of a `wire` for an `input` or `inout` port
- Bad declarations: `reg bus[0:31]` instead of `reg [31:0]bus`
- Mixing vector declarations: `wire [31:0]BusA, [15:0]BusB`
- The case-sensitivity of Verilog
- No delay in an `always` statement (simulator loops forever)
- Mixing up `‘` (accent grave) for `‘define` and `’` (tick or apostrophe) for `1’b1` with ´ (accent acute) or `‘` (open single quote) or `’` (close single quote)
- Mixing `"` (double quote) with `"` (open quotes) or `"` (close quotes)

11.5 (Sensitivity, 10 min.) Explore and explain what happens if you write this:

```
always @(a or b or c) e = (a|b)&(c|d);
```

11.6 (Verilog `if` statement, 10 min.) Build test code to simulate the following Verilog fragment. Explain what is wrong and fix the problem.

```
if (i > 0)
  if (i < 2) $display ("i is 1");
```

```
else $display ("i is less than 0");
```

11.7 (Effect of delay, 30 min.). Write code to test the four different code fragments shown in Table 11.14 and print the value of `'a'` at time = 0 and time = 1 for each case. Explain the differences in your simulation results.

TABLE 11.14 Code fragments for Problem 11.7.

	(a)	(b)	(c)	(d)

Code fragment	`reg a;`
	`initial`
	`begin`
	`a = 0;`
	`a = a + 1;`
	`end`
		`reg a;`
		`initial`
		`begin`
		`#0 a = 0;`
		`#0 a = a + 1;`
		`end`
			`reg a;`
			`initial`
			`begin`
			`a <= 0;`
			`a <= a + 1;`
			`end`
				`reg a;`
				`initial`
				`begin`
				`#1 a = 0;`
				`#1 a = a + 1;`
				`end`

11.8 (Verilog events, 10 min.). Simulate the following and explain the results:

```
event event_1, event_2;
always @ event_1 -> event_2;
initial @event_2 $stop;
initial -> event_1;
```

11.9 (Blocking and nonblocking assignment statements, 30 min.). Write code to test the different code fragments shown in Table 11.15 and print the value of `'outp'` at time = 0 and time = 10 for each case. Explain the difference in simulation results.

TABLE 11.15 Code fragments for Problem 11.9.

	(a)	(b)	(c)	(d)

Code fragment	`reg outp;`
	`always`
	`begin`
	`#10 outp = 0;`
	`#10 outp = 1;`
	`end`
		`reg outp;`
		`always`
		`begin`
		`outp <= #10 1;`
		`outp <= #10 0;`
		`end`
			`reg outp;`
			`always`
			`begin`
			`#10 outp = 0;`
			`#10 outp <= 1;`
			`end`
				`reg outp;`
				`always`
				`begin`
				`#10 outp <= 0;`
				`#10 outp = 1;`
				`end`

11.10 (Verilog UDPs, 20 min.). Use this primitive to build a half adder:

```
primitive Adder(Sum, InA, InB); output Sum; input Ina, InB;
table 00 : 0; 01 : 1; 10 : 1; 11 : 0; endtable
endprimitive
```

Apply unknowns to the inputs. What is the output?

11.11 (Verilog UDPs, 30 min.). Use the following primitive model for a D latch:

```
primitive DLatch(Q, Clock, Data); output Q; reg Q; input Clock, Data;
table 1 0 : ? : 0; 1 1 : ? : 1; 0 1 : ? : -; endtable
endprimitive
```

Check to see what happens when you apply unknown inputs (including clock transitions to unknown). What happens if you apply high-impedance values to the inputs (again including transitions)?

11.12 (Propagation of unknowns in primitives, 45 min.) Use the following primitive model for a D flip-flop:

```
primitive DFF(Q, Clock, Data); output Q; reg Q; input Clock, Data;
table
r    0 : ? : 0 ;
r    1 : ? : 1 ;
(0x) 0 : 0 : 0 ;
(0x) 1 : 1 : 1 ;
(?0) ? : ? : - ;
? (??) : ? : - ;
endtable
endprimitive
```

Check to see what happens when you apply unknown inputs (including a clock transition to an unknown value). What happens if you apply high-impedance values to the inputs (again including transitions)?

11.13 (D flip-flop UDP, 60 min.) Table 11.16 shows a UDP for a D flip-flop with QN output and asynchronous reset and set.

TABLE 11.16 D flip-flop UDP for Problem 11.13.

```
primitive DFlipFlop2(QN, Data, Clock, Res, Set);
output QN; reg QN; input Data, Clock, Res, Set;
table
//   Data   Clock   Res   Set   :state   :next state
  1  (01)  0  0  :?  :0;  // line 1
  1  (01)  0  x  :?  :0;
  ?  ?  0  x  :0  :0;
  0  (01)  0  0  :?  :1;
  0  (01)  x  0  :?  :1;
  ?  ?  x  0  :1  :1;
  1  (x1)  0  0  :0  :0;
  0  (x1)  0  0  :1  :1;
  1  (0x)  0  0  :0  :0;
  0  (0x)  0  0  :1  :1;
  ?  ?  1  ?  :?  :1;
  ?  ?  0  1  :?  :0;
  ?  n  0  0  :?  :-;
  *  ?  ?  ?  :?  :-;
  ?  ?  (?0)  ?  :?  :-;
  ?  ?  ?  (?0)  :?  :-;
  ?  ?  ?  ?  :?  :x;  // line 17
endtable
endprimitive
```

a. Explain the purpose of each line in the truth table.

b. Write a module to test each line of the UDP.

c. Can you find any errors, omissions, or other problems in this UDP?

11.14 (JK flip-flop, 30 min.) Test the following model for a JK flip-flop:

```
module JKFF (Q, J, K, Clk, Rst);
parameter width = 1, reset_value = 0;
input [width-1:0] J, K; output [width-1:0] Q; reg [width-1:0] Q;
input Clk, Rst; initial Q = {width{1'bx}};
always @ (posedge Clk or negedge Rst )
if (Rst==0 ) Q <= #1 reset_value;
else Q <= #1 (J & ~K) | (J & K & ~Q) | (~J & ~K & Q);
endmodule
```

11.15 (Overriding Verilog parameters, 20 min.) The following module has a parameter specification that allows you to change the number of AND gates that it models (the cardinality or width):

```
module Vector_AND(Z, A, B);
  parameter card = 2; input [card-1:0] A,B; output [card-1:0] Z;
  wire [card-1:0] Z = A & B;
endmodule
```

The next module changes the parameter value by specifying an overriding value in the module instantiation:

```
module Four_AND_Gates(OutBus, InBusA, InBusB);
  input [3:0] InBusA, InBusB; output [3:0] OutBus;
  Vector_AND #(4) My_AND(OutBus, InBusA, InBusB);
endmodule
```

These next two modules change the parameter value by using a `defparam` statement, which overrides the declared parameter value:

```
module X_AND_Gates(OutBus, InBusA, InBusB);
  parameter X = 2;input [X-1:0] InBusA, InBusB;output [X-1:0] OutBus;
  Vector_AND #(X) My_AND(OutBus, InBusA, InBusB);
endmodule
module size; defparam X_AND_Gates.X = 4; endmodule
```

a. Check that the two alternative methods of specifying parameters are equivalent by instantiating the modules `Four_AND_Gates` and `X_AND_Gates` in another module and simulating.

b. List and comment on the advantages and disadvantages of both methods.

11.16 (Default Verilog delays, 10 min.). Demonstrate, using simulation, that the following NAND gates have the delays you expect:

```
nand (strong0, strong1) #1
  Nand_1(n001, n004, n005),
  Nand_2(n003, n001, n005, n002);
nand (n006, n005, n002);
```

11.17 (Arrays of modules, 30 min.) Newer versions of Verilog allow the instantiating of **arrays of modules** (in this book we usually call this a vector since we are only allowed one row). You specify the number in the array by using a **range** after the instance name as follows:

```
nand #2 nand_array[0:7](zn, a, b);
```

Create and test a model for an 8-bit register using an array of flip-flops.

11.18 (Assigning Verilog real to integer data types, 10 min.). What is the value of `ImInteger` in the following code?

```
real ImReal; integer ImInteger;
initial begin ImReal = -1.5; ImInteger = ImReal; end
```

11.19 (BNF syntax, 10 min.) Use the BNF syntax definitions in Appendix B to answer the following questions. In each case explain how you arrive at the answer:

a. What is the highest-level construct?

b. What is the lowest-level construct?

c. Can you nest `begin` and `end` statements?

d. Where is a legal place for a `case` statement?

e. Is the following code legal: **reg** `[31:0] rega, [32:1] regb;`

f. Where is it legal to include sequential statements?

11.20 (Old syntax definitions, 10 min.) Prior to the IEEE LRM, Verilog BNF was expressed using a different notation. For example, an event expression was defined as follows:

```
<event_expression> ::= <expression>
  or <<posedge or negedge> <SCALAR_EVENT_EXPRESSION>>
  or <<event_expression> or <event_expression>>
```

Notice that we are using `'or'` as part of the BNF to mean "alternatively" and also `' or '` as a Verilog keyword. The keyword `' or '` is in bold--the difference is fairly obvious. Here is an alternative definition for an event expression:

```
<event_expression> ::= <expression>
||= posedge <SCALAR_EVENT_EXPRESSION>
||= negedge <SCALAR_EVENT_EXPRESSION>
||= <event_expression> <or <event_expression>>*
```

Are these definitions equivalent (given, of course, that we replaced `||=` with `or` in the simplified syntax)? Explain carefully how you would attempt to prove that they are the same.

11.21 (Operators, 20 min.) Explain Table 11.17 (see next page).

TABLE 11.17 Unary operators (Problem 11.21).			
	(a)	(b)	(c)

Code	`module unary;`
	`reg [4:0] u;`
	`initial u=!'b011z;`
	`initial $display("%b",u);`
	`endmodule`
		`module unary;`
		`wire u;`
		`assign u=!'b011z;`
		`initial $display("%b",u);`
		`endmodule`
			`module unary;`
			`wire u;`
			`assign u=!'b011z;`
			`initial #1 $display("%b`
			`endmodule`

Output	`0000x`	`z`	`x`

11.22 (Unary reduction, 10 min.) Complete Table 11.18 (see next page).

TABLE 11.18 Unary reduction (Problem 11.22).						
Operand	**&**	**~&**	**|**	**~|**	**^**	**~^**
`4'b0000`						
`4'b1111`						
`4'b01x0`						
`4'bz000`						

11.23 (Coerced ports, 20 min.) Perform some experiments to test the behavior of your Verilog simulator in the following situation: "NOTE--A port that is declared as input (output) but used as an output (input) or inout may be coerced to inout. If not coerced to inout, a warning must be issued" [Verilog LRM 12.3.6].

11.24 (*Difficult delay code, 20 min.) Perform some experiments to explain what this difficult to interpret statement does:

```
#2 a <= repeat(2) @(posedge clk) d;
```

11.25 (Fork-join, 20 min.) Write some test code to compare the behavior of the code fragments shown in Table 11.19.

TABLE 11.19 Fork-and-join examples for Problem 11.25.				
	(a)	(b)	(c)	(d)

Code fragment	`fork`
	`a = b;`
	`b = a;`
	`join`
		`fork`
		`a <= b;`
		`b <= a;`
		`join`
			`fork`
			`#1 a = b;`
			`#1 b = a;`
			`join`
				`fork`
				`a = #1 b;`
				`b = #1 a;`
				`join`

11.26 (Blocking and nonblocking assignments, 20 min.) Simulate the following code and explain the results:

```
module nonblocking; reg Y;
  always begin Y <= #10 1;Y <= #20 0;#10; end
```

```
  always begin $display($time,,"Y=",Y); #10; end
  initial #100 $finish;
endmodule
```

11.27 (*Flip-flop code, 10 min.) Explain why this flip-flop does not work:

```
module Dff_Res_Bad(D,Q,Clock,Reset);
output Q; input D,Clock,Reset; reg Q; wire D;
always @(posedge Clock) if (Reset !== 1) Q = D; always if (Reset == 1) Q = 0;
end endmodule
```

11.28 (D flip-flop, 10 min.) Test the following D flip-flop model:

```
module DFF (D, Q, Clk, Rst);
parameter width = 1,  reset_value = 0;
input [width-1:0] D; output [width-1:0] Q; reg [width-1:0] Q;
input Clk,Rst;
initial Q = {width{1'bx}};
always @ ( posedge Clk or negedge Rst )
if ( Rst == 0 ) Q <= #1 reset_value; else Q <= #1 D;
endmodule
```

11.29 (D flip-flop with scan, 10 min.) Explain the following model:

```
module DFFSCAN (D, Q, Clk, Rst, ScEn, ScIn, ScOut);
parameter width = 1,  reset_value = 0;
input [width-1:0] D; output [width-1:0] Q; reg [width-1:0] Q;
input Clk,Rst,ScEn,ScIn; output ScOut;
initial Q = {width{1'bx}};
always @ ( posedge Clk or negedge Rst ) begin
  if ( Rst == 0 )                Q <= #1 reset_value;
  else if (ScEn)                Q <= #1 {Q,ScIn};
  else                 Q <= #1 D;
end
assign ScOut=Q[width-1];
endmodule
```

11.30 (Pads, 30 min.) Test the following model for a bidirectional I/O pad:

```
module PadBidir (C, Pad, I, Oen); // active low enable
parameter width=1, pinNumbers="", \strength =1, level="CMOS",
pull="none", externalVdd=5;
output [width-1:0] C; inout  [width-1:0] Pad; input  [width-1:0] I;
input Oen;
assign #1 Pad = Oen ? {width{1'bz}} : I;
assign #1 C = Pad;
endmodule
```

Construct and test a model for a three-state pad from the above.

11.31 (Loops, 15 min.) Explain and correct the problem in the following code:

```
module Loop_Bad; reg [3:0] i; reg [31:0] DBus;
initial DBus = 0;
initial begin #1; for (i=0; i<=15; i=i+1) DBus[i]=1; end
initial begin
$display("DBus = %b",DBus); #2; $display("DBus = %b",DBus); $stop;
end endmodule
```

11.32 (Arithmetic, 10 min.) Explain the following:

```
integer IntA;
IntA = -12 / 3; // result is -4
IntA = -'d 12 / 3; // result is 1431655761
```

Determine and explain the values of `intA` and `regA` after each assignment statement in the following code:

```
integer intA; reg [15:0] regA;
intA = -4'd12; regA = intA/3; regA = -4'd12;
intA = regA/3; intA = -4'd12/3; regA = -12/3;
```

11.33 (Arithmetic overflow, 30 min.) Consider the following:

```
reg [7:0] a, b, sum; sum = (a + b) >> 1;
```

The intent is to add `a` and `b` , which may cause an overflow, and then shift `sum` to keep the carry bit. However, because all operands in the expression are of an 8-bit width, the expression `(a + b)` is only 8 bits wide, and we lose the carry bit before the shift. One solution forces the expression `(a + b)` to use at least 9 bits. For example, adding an integer value of 0 to the expression will cause the evaluation to be performed using the bit size of integers [LRM 4.4.2]. Check to see if the following alternatives produce the intended result:

```
sum = (a + b + 0) >> 1;
sum = {0,a} + {0,b} >> 1;
```

11.34 (*Data slip, 60 min.) Table 11.20 shows several different ways to model the connection of a 2-bit shift register. Determine which of these models suffer from data slip. In each case show your simulation results.

TABLE 11.20 Data slip (Problem 11.34).		
	Alternative	**Data slip?**
1	`always @(posedge Clk) begin Q2 = Q1; Q1 = D1; end`	
2	`always @(posedge Clk) begin Q1 = D1; Q2 = Q1; end`	
3	`always @(posedge Clk) begin Q1 <= #1 D1; Q2 <= #1 Q1; end`	
4	`always @(posedge Clk) Q1 = D1; always @(posedge Clk) Q2 = Q1;`	Y
5	`always @(posedge Clk) Q1 = #1 D1; always @(posedge Clk) Q2 = #1 Q1;`	N
6	`always @(posedge Clk) #1 Q1 = D1; always @(posedge Clk) #1 Q2 = Q1;`	
7	`always @(posedge Clk) Q1 <= D1; always @(posedge Clk) Q2 <= Q1;`	
8	`module FF_1 (Clk, D1, Q1); always @(posedge Clk) Q1 = D1; endmodule` `module FF_2 (Clk, Q1, Q2); always @(posedge Clk) Q2 = Q1; endmodule`	
9	`module FF_1 (Clk, D1, Q1); always @(posedge Clk) Q1 <= D1; endmodule` `module FF_2 (Clk, Q1, Q2); always @(posedge Clk) Q2 <= Q1; endmodule`	

11.35 (**Timing, 30 min.) What does a simulator display for the following?

```
assign p = q; initial begin q = 0; #1 q = 1; $display(p); end
```

What is the problem here? Conduct some experiments to illustrate your answer.

11.36 (Port connections, 10 min.) Explain the following declaration:

```
module test (.a(c), .b(c));
```

11.37 (**Functions and tasks, 30 min.) Experiment to determine whether invocation of a function (or task) behaves as a blocking or nonblocking assignment.

11.38 (Nonblocking assignments, 10 min.) Predict the output of the following model:

```
module e1; reg a, b, c;
initial begin a = 0; b = 1; c = 0; end
always c = #5 ~c;always @(posedge c) begin a <= b; b <= a; end
endmodule
```

11.39 (Assignment timing, 20 min.) Predict the output of the following module and explain the timing of the assignments:

```
module e2; reg a, b, c, d, e, f;
initial begin a = #10 1; b = #2 0; c = #4 1; end
initial begin d <= #10 1; e <= #2 0; f <= #4 1; end
endmodule
```

11.40 (Swap, 10 min.) Explain carefully what happens in the following code:

```
module e3; reg a, b;
initial begin a = 0; b = 1; a <= b; b <= a; end
endmodule
```

11.41 (*Overwriting, 30 min.) Explain the problem in the following code, determine what happens, and conduct some experiments to explore the problem further:

```
module m1; reg a;
initial a = 1;
initial begin a <= #4 0; a <= #4 1; end
endmodule
```

11.42 (*Multiple assignments, 30 min.) Explain what happens in the following:

```
module m2; reg r1; reg [2:0] i;
initial begin
r1 = 0; for (i = 0; i <= 5; i = i+1) r1 <= # (i*10) i[0]; end
endmodule
```

11.43 (Timing, 30 min) Write a model to mimic the behavior of a traffic light signal. The clock input is 1 MHz. You are to drive the lights as follows (times that the lights are on are shown in parentheses): green (60 s), yellow (1 s), red (60 s).

11.44 (Port declarations, 30 min.) The rules for port declarations are as follows: "The port expression in the port definition can be one of the following:

- a simple identifier
- a bit-select of a vector declared within the module
- a part-select of a vector declared within the module
- a concatenation of any of the above

Each port listed in the module definition's list of ports shall be declared in the body of the module as an input, output, or inout (bidirectional). This is in addition to any other declaration for a particular port--for example, a reg, or wire. A port can be declared in both a port declaration and a net or register declaration. If a port is declared as a vector, the range specification between the two declarations of a port shall be identical" [Verilog LRM 12.3.2].

Compile the following and comment (you may be surprised at the results):

```
module stop (); initial #1 $finish; endmodule
module Outs_1 (a); output [3:0] a; reg [3:0] a;
initial a <= 4'b10xz; endmodule
module Outs_2 (a); output [2:0] a; reg [3:0] a;
initial a <= 4'b10xz; endmodule
module Outs_3 (a); output [3:0] a; reg [2:0] a;
initial a <= 4'b10xz; endmodule
module Outs_4 (a); output [2:0] a; reg [2:0] a;
initial a <= 4'b10xz; endmodule
module Outs_5 (a); output a; reg [3:0] a;
initial a <= 4'b10xz; endmodule
module Outs_6 (a[2:0]); output [3:0] a; reg [3:0] a;
initial a <= 4'b10xz; endmodule
```

```
module Outs_7 (a[1]); output [3:0] a; reg [3:0] a;
initial a <= 4'b10xz; endmodule
module Outs_8 (a[1]); output a; reg [3:0] a;
always a <= 4'b10xz; endmodule
```

11.45 (Specify blocks, 30 min.)

a. Describe the pin-to-pin timing of the following module. Build a testbench to demonstrate your explanation.

```
module XOR_spec (a, b, z); input a, b: output z; xor x1 (z, a, b);
specify
  specparam tnr = 1, tnf = 2 specparam tir = 3, tif = 4;
  if ( a)(b => z) = (tir, tif); if ( b)(a => z) = (tir, tif);
  if (~a)(b=> z) = (tnr, tnf); if (~b)(a=> z) = (tnr, tnf);
endspecify
endmodule
```

b. Write and test a module for a 2:1 MUX with inputs `A0` , `A1` , and `sel` ; output `Z` ; and the following delays: `A0` to `Z` : 0.3 ns (rise) and 0.4 ns (fall); `A1` to `Z` : 0.2 ns (rise) and 0.3 ns (fall); `sel` to `Z` = 0.5 ns.

11.46 (Design contest, **60 min.) In 1995 John Cooley organized a contest between VHDL and Verilog for ASIC designers. The goal was to design the fastest 9-bit counter in under one hour using Synopsys synthesis tools and an LSI Logic vendor technology library. The Verilog interface is as follows:

```
module counter (data_in, up, down, clock,
  count_out, carry_out, borrow_out, parity_out);
output [8:0]  count_out;
output carry_out, borrow_out, parity_out;
input  [8:0]  data_in; input clock, up, down;
reg    [8:0]  count_out; reg carry_out,  borrow_out, parity_out;
// Insert your design here.
endmodule
```

The counter is positive-edge triggered, counts up with `up='1'` and down with `down='1'` . The contestants had the advantage of a predefined testbench with a set of test vectors; you do not. Design a model for the counter and a testbench.

11.47 (Timing checks, ***60 min.+) Flip-flops with preset and clear require more complex timing-check constructs than those described in Section 11.13.3. The following BNF defines a **controlled timing-check event**:

```
controlled_timing_check_event ::= timing_check_event_control specify_terminal_descri
timing_check_condition ::=
  scalar_expression | ~scalar_expression
| scalar_expression == scalar_constant
| scalar_expression === scalar_constant
| scalar_expression != scalar_constant
| scalar_expression !== scalar_constant
```

The scalar expression that forms the conditioning signal must be a scalar net, or else the least significant bit of a vector net or a multibit expression value is used. The comparisons in the timing check condition may be **deterministic** (using `===` , `!==` , `~`, or no operator) or **nondeterministic** (using `==` or `!=`). For deterministic comparisons, an `'x'` result disables the timing check. For nondeterministic comparisons,

an 'x' result enables the timing check.

As an example the following **unconditioned timing check**,

```
$setup(data, posedge clock, 10);
```

performs a setup timing check on every positive edge of `clock` , as was explained in Section 11.13.3. The following controlled timing check is enabled only when `clear` is high, which is what is required in a flip-flop model, for example.

```
$setup(data, posedge clock &&& clear, 10);
```

The next example shows two alternative ways to enable a timing check only when `clear` is low. The second method uses a nondeterministic operator.

```
$setup(data,posedge clock &&&(~clear),10); // clear=x disables check
$setup(data,posedge clock &&&(clear==0),10); // clear=x enables check
```

To perform the setup check only when `clear` and `preset` signals are high, you can add a gate outside the specify block, as follows:

```
and g1(clear_and_preset, clear, set);
```

A controlled timing check event can then use this `clear_and_preset` signal:

```
$setup(data, posedge clock &&& clear_and_preset, 10);
```

Use the preceding techniques to expand the D flip-flop model, dff_udp, from Section 11.13.3 to include asynchronous active-low preset and clear signals as well as an output, `qbar` . Use the following module interface:

```
module dff(q, qbar, clock, data, preset, clear);
```

11.48 (Verilog BNF, 30 min.) Here is the "old" BNF definition of a sequential block (used in the Verilog reference manuals and the OVI LRM). Are there any differences from the "new" version?

```
<sequential_block> ::=
  begin <statement>* end
  or
  begin: <block_IDENTIFIER> <block_declaration>*
    <statement>*
  end
<block_declaration> ::= parameter <list_of_param_assignment>;
  or reg <range>? <attribute_decl>*
    <list_of_register_variable>;
  or integer <attribute_decl>* <list_of_register_variable>;
  or real <attribute_decl>* <list_of_variable_IDENTIFIER>;
  or time <attribute_decl>* <list_of_register_variable>;
  or event <attribute_decl>* <list_of_event_IDENTIFIER>;
<statement> ::=
  <blocking_assignment>;
  or <non-blocking_assignment>;
  or if(<expression>) <statement_or_null>
    <else <statement_or_null> >?
  or <case or casez or casex>
```

```
    (<expression>) <case item>+ endcase
  or forever <statement>
  or repeat(<expression>) <statement>
  or while(<expression>) <statement>
  or for(<assignment>;
    <expression>; <assignment>) <statement>
  or wait(<expression>) <statement_or_null>
  or disable <task_IDENTIFIER>;
  or disable <block_IDENTIFIER>;
  or force <assignment>; or release <value>;
  or <timing_control> <statement_or_null>
  or -> <event_IDENTIFIER>;
  or <sequential_block> or <parallel_block>
  or <task_enable> or <system_task_enable>
```

11.49 (Conditional compiler directives, 30 min.) The conditional compiler directives: `‘define` , `‘ifdef` , `‘else` , `‘endif` , and `‘undef` ; work much as in C. Write and compile a module that models an AND gate as ’z = a&b’ if the variable behavioral is defined. If behavioral is not defined, then model the AND gate as ’and a1 (z, a, b)’.

11.50 (*Macros, 30 min.) According to the IEEE Verilog LRM [16.3.1] you can create a **macro** with parameters using `‘define` , as the following example illustrates. This is a particularly difficult area of compliance. Does your software allow the following? You may have to experiment considerably to get this to work. Hint: Check to see if your software is substituting for the macro text literally or if it does in fact substitute for parameters.

```
‘define M_MAX(a, b)((a) > (b) ? (a) : (b))
‘define M_ADD(a,b) (a+b)
module macro;
reg m1, m2, m3, s0, s1;
‘define var_nand(delay) nand #delay
‘var_nand (2) g121 (q21, n10, n11);
‘var_nand (3) g122 (q22, n10, n11);
initial begin s0=0; s1=1;
m1 = ‘M_MAX (s0, s1); m2 = ‘M_ADD (s0,s1); m3 = s0 > s1 ? s0 : s1;
end
initial #1 $display(" m1=",m1," m2=",m2," m3=",m3);
endmodule
```

11.51 (**Verilog hazards, 30 min.) The MTI simulator, VSIM, is capable of detecting the following kinds of Verilog hazards:

1. WRITE/WRITE: Two processes writing to the same variable at the same time.
2. READ/WRITE: One process reading a variable at the same time it is being written to by another process. VSIM calls this a READ/WRITE hazard if it executed the read first.
3. WRITE/READ: Same as a READ/WRITE hazard except that VSIM executed the write first.

For example, the following log shows how to simulate Verilog code in hazard mode for the example in Section 11.6.2:

```
> vlib work
> vlog -hazards data_slip_1.v
> vsim -c -hazards data_slip_1
...(lines omitted)...
# 100 0   1 1  x
```

```
# ** Error: Write/Read hazard detected on Q1 (ALWAYS 3 followed by ALWAYS 4)
#    Time: 150 ns  Iteration: 1  Instance:/
# 150 1   1 1  1
...(lines omitted)...
```

There are a total of five hazards in the module data_slip_1, four are on Q1, but there is another. If you correct the code as suggested in Section 11.6.2 and run VSIM, you will find this fifth hazard. If you do not have access to MTI's simulator, can you spot this additional read/write hazard? Hint: It occurs at time zero on Clk. Explain.

11.15.1 The Viterbi Decoder

11.52 (Understanding, 20 min.) Calculate the values shown in Table 11.8 if we use 4 bits for the distance measures instead of 3.

11.53 (Testbenches)

a. (30 min.) Write a testbench for the encoder, viterbi_encode, in Section 11.12 and reproduce the results of Table 11.7.

b. (30 min.) Write a testbench for the receiver front-end viterbi_distances and reproduce the results of Table 11.9 (you can write this stand-alone or use the answer to part a to generate the input). Hint: You will need a model for a D flip-flop. The sequence of results is more important than the exact timing. If you do have timing differences, explain them carefully.

11.54 (Things go wrong, 60 min.) Things do not always go as smoothly as the examples in this book might indicate. Suppose you accidentally invert the sense of the reset for the D flip-flops in the encoder. Simulate the output of the faulty encoder with an input sequence $X_n = 0, 1, 2, 3, ...$ (in other words run the encoder with the flip-flops being reset continually). The output sequence looks reasonable (you should find that it is $Y_n = 0, 2, 4, 6, ...$). Explain this result using the state diagram of Figure 11.3. If you had constructed a testbench for the entire decoder and did not check the intermediate signals against expected values you would probably never find this error.

11.55 (Subset decoder) Table 11.21 shows the inputs and outputs from the first-stage of the Viterbi decoder, the subset decoder. Calculate the expected output and then confirm your predictions using simulation.

TABLE 11.21 Subset decoder (Problem 11.55).

input	in0	in1	in2	in3	in4	in5	in6	in7	s0	s1	s2	s3	sout0	sout1	sout2	sout3
5	6	7	6	4	1	0	1	4	1	0	1	4				
4	7	6	4	1	0	1	4	6	0	1	4	1				
1	1	0	1	4	6	7	6	4	1	0	1	4				
0	0	1	4	6	7	6	4	1	0	1	4	1				

11.16 Bibliography

The IEEE Verilog LRM [1995] is less intimidating than the IEEE VHDL LRM, because it is based on the OVI LRM, which in turn was based on the Verilog-XL simulator reference manual. Thus it has more of a "User's Guide" flavor and is required reading for serious Verilog users. It is the only source for detailed information on the PLI.

Phil Moorby was one of the original architects of the Verilog language. The Thomas and Moorby text is a good introduction to Verilog [1991]. The code examples from this book can be obtained from the World Wide Web. Palnitkar's book includes an example of the use of the PLI routines [1996].

Open Verilog International (OVI) has a Web site maintained by Chronologic (`http://www.chronologic.com/ovi`) with membership information and addresses and an ftp site maintained by META-Software (`ftp://ftp.metasw.com` in `/pub/OVI/`). OVI sells reference material, including proceedings from the International Verilog HDL Conference.

The newsgroup `comp.lang.verilog` (with a FAQ--frequently asked questions) is accessible from a number of online sources. The FAQ includes a list of reference materials and book reviews. Cray Research maintained an archive for `comp.lang.verilog` going back to 1993 but this was lost in January 1997 and is still currently unavailable. Cadence has a discussion group at `talkverilog@cadence.com` . Wellspring Solutions offers VeriWell, a no-cost, limited capability, Verilog simulator for UNIX, PC, and Macintosh platforms.

There is a free, "copylefted" Verilog simulator, `vbs` , written by Jimen Ching and Lay Hoon Tho as part of their Master's theses at the University of Hawaii, which is part of the `comp.lang.verilog` archive. The package includes explanations of the mechanics of a digital event-driven simulator, including event queues and time wheels.

More technical references are included as part of Appendix B.

11.17 References

IEEE Std 1364-95, Verilog LRM. 1995. The Institute of Electrical and Electronics Engineers. Available from The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017 USA. [cited on p. 479]

Palnitkar, S. 1996. Verilog HDL: A Guide to Digital Design and Synthesis. Upper Saddle River, NJ: Prentice-Hall, 396 p. ISBN 0-13-451675-3.

Thomas, D. E., and P. Moorby. 1991. The Verilog Hardware Description Language. 1st ed. Dordrecht, Netherlands: Kluwer, 223 p. ISBN 0-7923-9126-8, TK7885.7.T48 (1st ed.). ISBN 0-7923-9523-9 (2nd ed.).

Chapter start Previous page