3 Verilog for Simulation and Synthesis

This chapter presents Verilog from the point of view of a designer wanting to describe a design, perform pre-synthesis simulation, and synthesize his or her design for programming an FPGA or generating a layout. Many of the complex Verilog constructs related to timing and fine modeling features of this language will not be covered here. The chapter first describes Verilog with emphasis on design using simple examples. We will cover the basics, just enough to describe our examples. In a later section after a general familiarity with the language is gained, more complex features of the Verilog language with emphasis on testbench development will be described.

3.1 Design with Verilog

Verilog syntax and language constructs are designed to facilitate description of hardware components for simulation and synthesis. In addition, Verilog can be used to describe testbenches, specify test data and monitor circuit responses. Figure 3.1 shows a simulation model that consists of a design and its testbench in Verilog. Simulation output is generated in form of a waveform for visual inspection or data files for machine readability.

After a design passes basic functional validations, it must be synthesized into a netlist of components of a target library. Constructs used for verification of a design, or timing checks and timing specifications are not synthesizable. A Verilog design that is to be synthesized must use language constructs that have a clear hardware correspondence. Figure 3.2 shows a block diagram specifying the synthesis process.

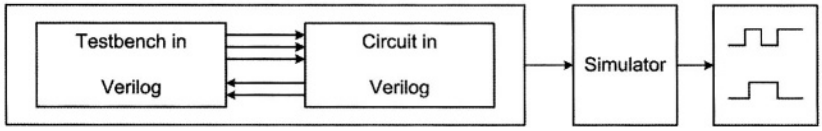

Figure 3.1 Simulation in Verilog

The output of synthesis is a netlist of components of the target library. Often synthesis tools have an option to generate this netlist in Verilog. In this case, the same testbench prepared for pre-synthesis simulation can be used with the netlist generated by the synthesis tool.

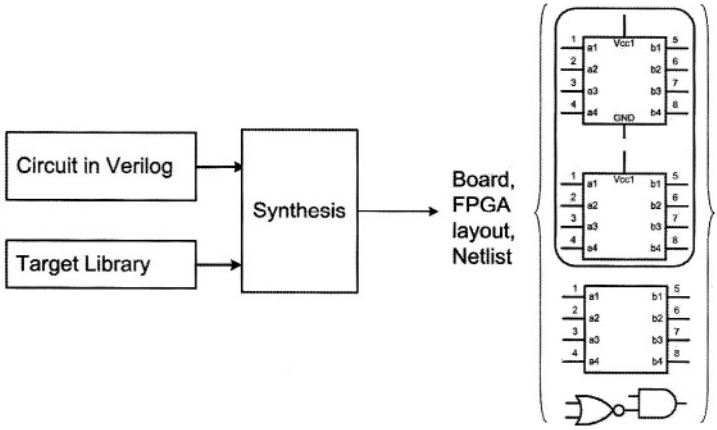

Figure 3.2 Synthesis

3.1.1 Modules

The entity used in Verilog for description of hardware components is a **module.** A module can describe a hardware component as simple as a transistor or a network of complex digital systems. As shown in Figure 3.3, modules begin with the **module** keyword and end with **endmodule.**

```
module
…
…
endmodule
```

Figure 3.3 Module

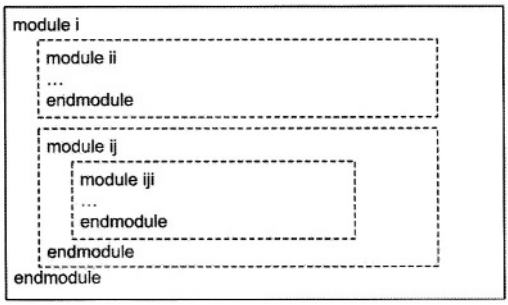

Figure 3.4 Module Hierarchy

A design may be described in a hierarchy of other modules. The top-level module is the complete design, and modules lower in the hierarchy are the design's components. Module instantiation is the construct used for bringing a lower level module into a higher level one. Figure 3.4 shows a hierarchy of several nested modules.

As shown in Figure 3.5, in addition to the **module** keyword, a module header also includes the module name and list of its ports. Following the module header, its ports and internal signals and variables are declared. Specification of the operation of a module follows module declarations.

```
module name (ports);
  port declarations;
  other declarations;
  . . .
  statements
  . . .
endmodule
```

Figure 3.5 Module Outline

```
module simple1a (i1, i2, i3, w1, w2);
  input i1, i2;
  output w1, w2;
  . . .
  and g1 (w1, i1, i2);
endmodule

module simple1b (i1, i2, i3, w1, w2);
  input i1, i2;
  output w1, w2;
  . . .
  assign w1 = i1 & i2;
  assign w2 = i1 & i2 | ~i3;
endmodule

module simple1c (i1, i2, i3, w1, w2);
  input i1, i2;
  output w1, w2;
  reg w1;
  always @ (i1 or i2) begin
    if (i1 == 1) w1 = i2; else w1 = 0;
  end
endmodule
```

Figure 3.6 Module Definition Alternatives

Operation of a module can be described at the gate level, using Boolean expressions, at the behavioral level, or a mixture of various levels of abstraction. Figure 3.6 shows three ways operation of a module may be described. Module *simple1a i*n Figure 3.6 uses Verilog's gate primitives, *simple1b* uses concurrent statements, and *simple1c* uses a procedural statement.

The subsections that follow describe details of module ports and description styles. In the examples in this chapter Verilog keywords and reserved words are shown in bold. Verilog is case sensitive. It allows letters, numbers and special character "_" to be used for names. Names are used for modules, parameters, ports, variables, and instance of gates and modules.

For readability of graphics, we use the symbol shown in Figure 3.7 for representing a Verilog module. Inputs are shown as hollow boxes, and outputs as solid ones. The name of the module appears inside the module box on its upper side.

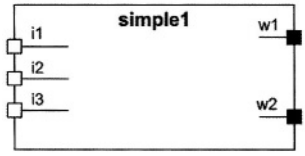

Figure 3.7 Module Notation

3.1.2 Module Ports

Following the name of a module is a set of parenthesis with a list of module ports. This list includes inputs, outputs and bidirectional input lines. Ports may be listed in any order. This ordering can only become significant when a module is instantiated, and does not affect the way its operation is described. Top-level modules used for testbenches have no ports.

```
module acircuit (a, b, c, av, bv, cv, w, wv);
  input a, b;
  output w;
  inout c;
  input [7:0] av, bv;
  output [7:0] wv;
  inout [7:0] cv;
  . . .
endmodule
```

Figure 3.8 Module Ports

Following module header, ports of a module are declared. In this part, size and direction of each port listed in the module header are specified. A port may be **input, output** or **inout.** The latter type is used for bidirectional input/output lines. Size of vectored ports of a module is also declared in the

module port declaration part. Size and indexing of a port is specified after its type name within square brackets. Figure 3.8 shows an example circuit with scalar, vectored, input, output and inout ports. Ports named *a*, and *b* are one-bit inputs. Ports *av* and *bv* are 8-bit inputs of *acircuit.* The set of square brackets that follow the **input** keyword applies to all ports that follow it. Port *w* of *acircuit* is declared as a 1-bit output, and *wv* is an 8-bit bi-directional port of this module.

```
module bcircuit (a, b, av, bv, w, wv);
  input a, b;
  output w;
  Input [7:0] av, bv;
  output [7:0] wv;
  wire d;
  wire [7:0] dv;
  reg e;
  reg [7:0] ev;
  . . .
endmodule
```

Figure 3.9 Wire and Variable Declaration

In addition to port declarations, a module declarative part may also include wire and variable declarations that are to be used inside the module. Wires (that are called **net** in Verilog) are declared by their types, **wire, wand** or **wor**; and variables are declared as **reg.** Wires are used for interconnections and have properties of actual signals in a hardware component. Variables are used for behavioral descriptions and are very much like variables in software languages. Figure 3.9 shows several wire and variable declarations.

```
module vcircuit (av, bv, cv, wv);
  input [7:0] av, bv, cv;
  output [7:0] wv;
  wire [7 :0] iv, jv;
  assign iv = av & cv;
  assign jv = av | cv;
  assign wv = iv ^ jv;
endmodule
```

Figure 3.10 Using Wires

Wires represent simple interconnection wires, busses, and simple gate or complex logical expression outputs. When wires are used on the left hand sides of **assign** statements, they represent outputs of logical structures. Wires can be used in scalar or vector form. Figure 3.10 shows several examples of wires used on the right and left hand sides of **assign** statements.

In the vector form, inputs, outputs, wires and variables may be used as a complete vector, part of a vector, or a bit of the vector. The latter two are referred to as part-select and bit-select.

3.1.3 Logic Value System

Verilog uses a 4-value logic value system. Values in this system are **0**, **1**, **Z**, and **X**. Value **0** is for logical **0** which in most cases represent a path to ground (Gnd). Value **1** is logical **1** and it represents a path to supply (Vdd). Value **Z** is for float, and **X** is used for un-initialized, undefined, un-driven, unknown, and value conflicts. Values **Z** and **X** are used for wired-logic, busses, initialization values, tri-state structures, and switch-level logic.

For more logic precision, Verilog uses strengths values as well as logic values. Our dealing with Verilog is for design and synthesis, and these issues will not be discussed here.

3.2 Combinational Circuits

A combinational circuit can be represented by its gate level structure, its Boolean functionality, or description of its behavior. At the gate level, interconnection of its gates are shown; at the functional level, Boolean expressions representing its outputs are written; and at the behavioral level a software-like procedural description represents its functionality. This section shows these three levels of abstraction for describing combinational circuits.

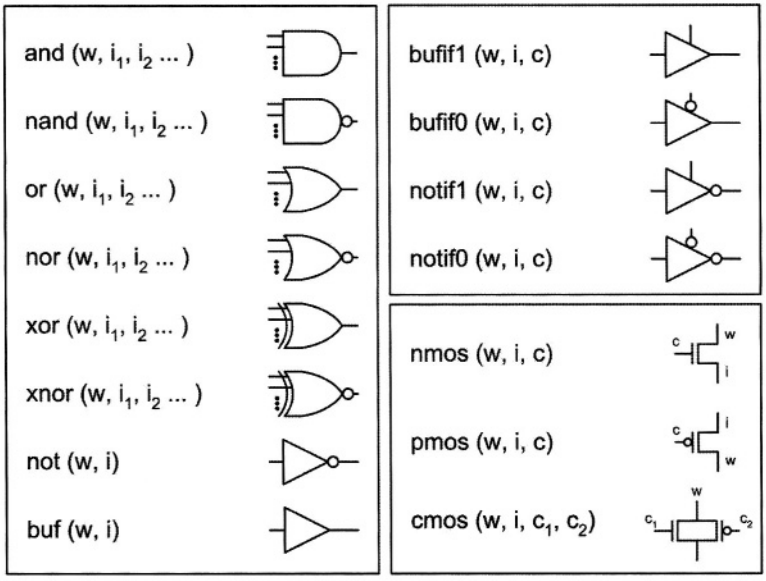

Figure 3.11 Basic Primitives

3.2.1 Gate Level Combinational Circuits

Verilog provides primitive gates and transistors. Some of the more important Verilog primitives and their logical representations are shown in Figure 3.11. In this figure *w* is used for gate outputs, *i* for inputs and *c* for control inputs.

Basic logic gates are **and, nand, or, nor, xor, xnor.** These gates can be used with one output and any number of inputs. The other two structures shown, are **not** and **buf.** These gates can be used with one input and any number of outputs.

Another group of primitives shown in this figure are three-state (tri-state is also used to refer to these structures) gates. Gates shown have *w* for their outputs, *i* for data inputs, and *c* for their control inputs. These primitives are **bufif1, notif1, bufif0,** and **notif0.** When control *c* for such gates is active (**1** for first and third, and **0** for the others), the data input, *i*, or its complement appears on the output of the gate. When control input of a gate is not active, its output becomes high-impedance, or **Z**.

Also shown in Figure 3.11 are NMOS, PMOS and CMOS structures. These are switches that are used in switch level description of gates, complex gates, and busses. The **nmos (pmos)** primitive is a simple switch with an active high (low) control input. The **cmos** switch is usually used with two complementary control inputs. These switches behave like the three-state gates. They are different in their output voltage levels and drive strengths. These parameters are modeled by wire strengths and are not discussed in this book.

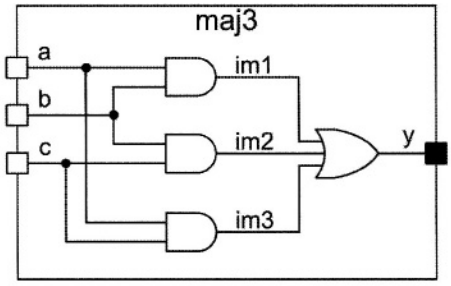

Figure 3.12 A Majority Circuit

Majority Example. We use the majority circuit of Figure 3.12 to illustrate how primitive gates are used in a design. The description shown in Figure 3.13 corresponds to this circuit. The module description has inputs and outputs according to the schematic of Figure 3.12.

Line 1 of the code shown is the **timescale** directive. This defines all time units in the description and their precision. For our example, *1ns/100Ps* means that all numbers in the code that represent a time value are in nanoseconds and they can have up to one fractional digit (100 Ps).

The statement that begins in Line 6 and ends in Line 9 instantiates three **and** primitives. The construct that follows the primitive name specifies rise and

fall delays for the instantiated primitive **(t_{plh}=2, t_{phl}=4).** This part is optional and if eliminated, 0 values are assumed for rise and fall delays. Line 7 shows inputs and outputs of one of the three instances of the **and** primitive. The output is *im1* and inputs are module input ports *a* and *b*. The port list on Line 7 must be followed by a comma if other instances of the same primitive are to follow, otherwise a semicolon should be used, like the end of Line 9. Line 8 and Line 9 specify input and output ports of the other two instances of the **and** primitive. Line 10 is for instantiation of the **or** primitive at the output of the majority gate. The output of this gate is *y* that comes first in the port list, and is followed by inputs of the gate. In this example, intermediate signals for interconnection of gates are *im1, im2,* and *im3.* Scalar interconnecting wires need not be explicitly declared in Verilog.

```
`timescale 1ns/100ps                  // Line 1
module maj3 ( a, b, c, y );
  input a, b, c;
  output y;

  and #(2,4)                          // Line 6
    (im1, a, b),                      // Line 7
    (im2, b, c ),                     // Line 8
    (im3, c, a );                     // Line 9
  or  #(3,5) (y, im1, im2, im3);      //Line 10

endmodule
```

Figure 3.13 Verilog Code for the Majority Circuit

The three **and** instances could be written as three separate statements, like instantiation of the **or** primitive. If we were to specify different delay values for the three instances of the **and** primitive, we had to have three separate primitive instantiation statements.

Three-state gates are instantiated in the same way as the regular logic gates. Outputs of three-state gates can be wired to form wired-and, wired-or, or wiring logic. For various wiring functions, Verilog uses **wire, wand, wor, tri, tri0** and **tri1 net** types. When two wires (**net**s) are connected, the resulting value depends on the two **net** values, as well as the type of the interconnecting **net**. Figure 3.14 shows **net** values for **net** types **wire, wand** and **wor.**

	Driving **net** values									
	net values									
net type	00	01	0Z	0X	11	1Z	1X	ZZ	ZX	XX
wire	0	X	0	X	1	1	X	Z	X	X
wand	0	0	0	0	1	X	X	Z	X	X
wor	0	1	X	X	1	1	1	X	X	X

Figure 3.14 "net" Type Resolutions

The table shown in Figure 3.14 is called a **net** resolution table. Several examples of **net** resolutions are shown in Figure 3.15. The **tri net** type mentioned above is the same as the **wire** type. **tri0** and **tri1** types resolve to **0** and **1**, respectively, when driven by all **Z** values.

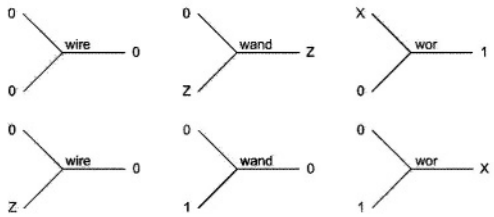

Figure 3.15 "net" Resolution Examples

Multiplexer Example. Figure 3.16 shows a 2-to-1 multiplexer using three-state gates. The Verilog code of this multiplexer is shown in Figure 3.17.

Lines 6 and 7 in Figure 3.17 instantiate two three-state gates. Their output is *y*, and since it is driven by both gates a wired-net is formed. Since *y* is not declared, its **net** type defaults to **wire.** When *s* is **1**, *bufif1* conducts and the value of *b* propagates to its output. At the same time, because *s* is **1**, *bufif0* does not conduct and its output becomes **Z**. Resolution of these values driving **net** *y* is determined by the **wire net** resolution as shown in Figure 3.14.

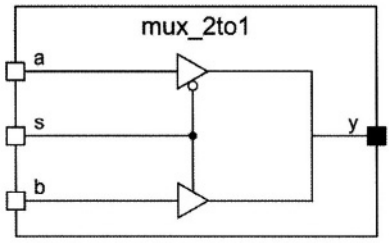

Figure 3.16 Multiplexer Using Three-state Gates

```
`timescale 1ns/100ps

module mux_2to1 ( a, b, s, y );
  input a, b, s;
  output y;
  bufif1 #(3) (y, b, s);          // Line 6
  bufif0 #(5) (y, a, s);          // Line 7
endmodule
```

Figure 3.17 Multiplexer Verilog Code

CMOS NAND Example. As another example of instantiation of primitives, consider the two-input CMOS NAND gate shown in Figure 3.18.

The Verilog code of Figure 3.19 describes this CMOS NAND gate. Logically, NMOS transistors in a CMOS structure push **0** into the output of the gate. Therefore, in the Verilog code of the CMOS NAND, input to output direction of NMOS transistors are from *Gnd* towards *w*. Likewise, PMOS transistors push a **1** value into *w*, and therefore, their inputs are considered the *Vdd* node and their outputs are connected to the *w* node. The *im1* signal is an intermediate **net** and is explicitly declared.

In the Verilog code of CMOS NAND gate, primitive gate instance names are used. This naming (*T1, T2, T3, T4*) is optional for primitives and mandatory when modules are instantiated. Examples of module instantiations are shown in the next section.

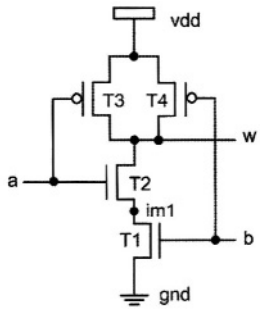

Figure 3.18 CMOS NAND Gate

```
module cmos_nand (a, b, w);
input a, b;
output w;
wire im1;
supply1 vdd;
supply0 gnd;

  nmos #(3, 4)
    T1 (im1, gnd, b),
    T2 (w, im1, a);

  pmos #(4, 5)
    T3 (w, vdd, a),
    T4 (w, vdd, b);

endmodule
```

Figure 3.19 CMOS NAND Verilog Description

3.2.2 Descriptions by Use of Equations

At a higher level than gates and transistors, a combinational circuit may be described by use of Boolean, logical, and arithmetic expressions. For this purpose the Verilog concurrent **assign** statement is used. Figure 3.20 shows Verilog operators that can be used with **assign** statements.

XOR Example. As our first example for using an **assign** statement consider the description of an XOR gate as shown in Figure 3.21. The **assign** statement uses *y* on the left-hand-side and equates it to Exclusive-OR of *a, b,* and c inputs.

Effectively, this **assign** statement is like driving *y* with the output of a 3-input **xor** primitive gate. The difference is that, the use of an **assign** statement gives us more flexibility and allows the use of more complex functions than what is available as primitive gates. Instead of being limited to the gates shown in Figure 3.11, we can write our own expressions using operators of Figure 3.20.

Bitwise Operators	&	|	^	~	~^	^~	
Reduction Operators	&	~&	|	~|	^	~^	^~
Arithmetic Operators	+	-	*	/	%		
Logical Operators	&&	||	!				
Compare Operators	<	>	<=	>=	==		
Shift Operators	>>	<<					
Concatenation Operators	{ }	{ n{ } }					

Figure 3.20 Verilog Operators

```
module xor3 ( a, b, c, y );
  input a, b, c;
  output y;

  assign y = a^b^c;

endmodule
```

Figure 3.21 XOR Verilog Code

Full-Adder Example. Figure 3.22 shows another example of using **assign** statements. This code corresponds to a full-adder circuit (see Chapter 2). The *s* output is the XOR result of *a*, *b* and *ci* inputs, and the *co* output is an AND-OR expression involving these inputs.

A delay value of 10 ns is used for the *s* output and 8 ns for the *co* output. As with the gate outputs, rise and fall delay values can be specified for a **net** that is used on the left-hand side of an **assign** statement. This construct allows the use of two delay values. If only one value is specified, it applies to both rise and fall transitions.

```
`timescale 1ns/100ps

module add_1bit ( a, b, ci, s, co );
  input a, b, ci;
  output s, co;
  assign #(10) s = a^b^ci;
  assign #(8) co = ( a & b ) | ( b & ci ) | ( a & ci );
endmodule
```

Figure 3.22 Full Adder Verilog Code

Another property of **assign** statements that also corresponds to gate instantiations is their concurrency. The statements in the Verilog module of Figure 3.22 are concurrent. This means that the order in which they appear in this module is not important. These statements are sensitive to events on their right hand sides. When a change of value occurs on any of the right hand side **net** or variables, the statement is evaluated and the resulting value is scheduled for the left hand side **net**.

Comparator Example. Figure 3.23 shows another example of using **assign** statements. This code describes a 4-bit comparator. The first **assign** statement uses a bitwise XOR operation on its right hand side. The result that is assigned to the *im* intermediate **net** is a 4-bit vector formed by XORing bits of *a* and *b* input vectors. The second **assign** statement uses the NOR reduction operator to NOR bits of *im* to generate the equal output for the 4-bit comparator.

The above describes the comparator using its Boolean function. However, using compare operators of Verilog, the *eq* output of the comparator may be written as:

```
assign eq = (a == b);
```

In this expression, (*a* == *b*) results in **1** if *a* and *b* are equal, and **0** if they are not. This result is simply assigned to *eq*.

The right-hand side expression of an **assign** statement can have a condition expression using the **?** and **:** operators. These operators are like if-then-else. In reading expressions that involve a condition operator, **?** and **:** take places of **then** and **else** respectively. The if-condition appears to the left of **?**.

```
module comp_4bit ( a, b, eq );
  input [3:0]a, b;
  output eq;
  wire [3:0] im;
  assign im = a^b;
  assign eq = ~| im;
endmodule
```

Figure 3.23 Four-Bit Comparator

Multiplexer Example. Figure 3.24 shows a 2-to-1 multiplexer using a condition operator. The expression shown reads as follows: **if** *s* is **1**, **then** *y* is *i1* **else** it becomes *i0.*

```
module mux2_1 ( i0, i1, s, y );
  input [3:0] i0, i1;
  input s;
  output [3:0] y;

  assign y = s ? i1 : i0;

endmodule
```

Figure 3.24 A 2-to-1 Mux Using Condition Operator

Decoder Example. Figure 3.25 shows another example using the condition operator. In this example a nesting of several **?:** operations are used to describe a decoder.

```
`timescale 1ns/100ps

module dcd2_4( a, b, d0, d1, d2, d3 );
   input a, b;
   output d0, d1, d2, d3;

   assign {d3, d2, d1, d0} =
      ( {a, b} == 2'b00 ) ? 4'b0001 :
      ( {a, b} == 2'b01 ) ? 4'b0010 :
      ( {a, b} == 2'b10 ) ? 4'b0100 :
      ( {a, b} == 2'b11 ) ? 4'b1000 :
        4'b0000;

endmodule
```

Figure 3.25 Decoder Using ?: and Concatenation

The decoder description also uses the concatenation operator **{ }** to form vectors from its scalar inputs and outputs. The decoder has four outputs, *d3, d2, d1* and *d0* and two inputs *a* and *b.* Input values **00, 01, 10,** and **11** produce **0001, 0010, 0100,** and **1000** outputs. In order to be able to compare *a* and *b* with their possible values, a two-bit vector is formed by concatenating *a* and *b.* The *{a, b}* vector is then compared with the four possible values it can take using a nesting of **?:** operations.

Similarly, in order to be able to place vector values on the outputs, the four outputs are concatenated using the **{ }** operator and used on the left-hand side of the **assign** statement shown in Figure 3.25.

This example also shows the use of sized numbers. Constants for the inputs and outputs have the general format of **n`bm.** In this format, **n** is the number of bits, **b** is the base specification and **m** is the number in base **b**. For calculation of the corresponding constant, number **m** in base **b** is translated to **n** bit binary. For example, *4 `hA* becomes 1010 in binary.

Adder Example. For another example using **assign** statements, consider an 8-bit adder circuit with a carry-in and a carry-out output. The Verilog code of this adder, shown in Figure 3.26, uses an **assign** statement to set concatenation of *co* on the left-hand side of *s* to the sum of *a*, *b* and *ci*. This sum results in nine bits with the left-most bit being the resulting carry. The sum is captured in the 9-bit left-hand side of the **assign** statement in {co, s}.

So far in this section we have shown the use of operators of Figure 3.20 in **assign** statements. A Verilog description may contain any number of **assign** statements and can use any mix of the operators discussed. The next example shows multiple **assign** statements.

```
module add_4bit ( a, b, ci, s, co );
  input [7:0] a, b;
  output [7:0] s;
  input ci;
  output co;

  assign {co, s} = a + b + ci;

endmodule
```

Figure 3.26 Adder with Carry-in and Carry-out

ALU Example. As our final example of **assign** statements, consider an ALU that performs add and subtract operations and has two flag outputs *gt* and *zero.* The *gt* output becomes **1** when input *a* is greater than input *b,* and the *zero* output becomes **1** when the result of the operation performed by the ALU is **0**.

Figure 3.27 shows the Verilog code of this ALU. Used in this description are arithmetic, concatenation, condition, compare and relational operations.

```
module ALU ( a, b, ci, addsub, gt, zero, co, r );
  input [7:0] a, b;
  output [7:0] r;
  input ci;
  output gt, zero, co;

  assign {co, s} = addsub ? (a + b + ci) : (a – b – ci);
  assign gt = (a>b);
  assign zero = (r == 0);

endmodule
```

Figure 3.27 ALU Verilog Code Using a Mix of Operations

3.2.3 Descriptions with Procedural Statements

At a higher level of abstraction than describing hardware with gates and expressions, Verilog provides constructs for procedural description of hardware. Unlike gate instantiations and **assign** statements that correspond to concurrent sub-structures of a hardware component, procedural statements describe the hardware by its behavior. Also, unlike concurrent statements that appear directly in a module body, procedural statements must be enclosed in procedural blocks before they can be put inside a module.

The main procedural block in Verilog is the **always** block. This is considered a concurrent statement that runs concurrent with all other statements in a module. Within this statement, procedural statements like **if-else** and **case** statements are used and are executed sequentially. If there are more than one procedural statement inside a procedural block, they must be bracketed by **begin** and **end** keywords.

Unlike assignments in concurrent bodies that model driving logic for left hand side wires, assignments in procedural blocks are assignments of values to variables that hold their assigned values until a different value is assigned to them. A variable used on the left hand side of a procedural assignment must be declared as **reg.**

An event control statement is considered a procedural statement, and is used inside an **always** block. This statement begins with an at-sign, and in its simplest form, includes a list of variables in the set of parenthesis that follow the at-sign, e.g., @ (*v1* **or** v2 ...); .

When the flow of the program execution within an **always** block reaches an event-control statement, the execution halts (suspends) until an event occurs on one of the variables in the enclosed list of variables. If an event-control statement appears at the beginning of an **always** block, the variable list it contains is referred to as the *sensitivity list* of the **always** block. For combinational circuit modeling all variables that are read inside a procedural block must appear on its sensitivity list.

Examples that follow show various ways combinational component may be modeled by procedural blocks.

Majority Example. Figure 3.28 shows a majority circuit described by use of an **always** block. In the declarative part of the module shown, the *y* output is declared as **reg** since this variable is to be assigned a value inside a procedural block.

The **always** block describing the behavior of this circuit uses an event control statement that encloses a list of variables that is considered as the sensitivity list of the **always** block. The **always** block is said to be sensitive to *a*, *b* and *c* variables. When an event occurs on any of these variables, the flow into the **always** block begins and as a result, the result of the Boolean expression shown will be assigned to variable *y*. This variable holds its value until the next time an event occurs on *a*, *b*, or *c* inputs.

In this example, since the **begin** and **end** bracketing only includes one statement, its use is not necessary. Furthermore, the syntax of Verilog allows elimination of semicolon after an event control statement. This effectively collapses the event control and the statement that follows it into one statement.

```
module maj3 ( a, b, c, y );
  input a, b, c;
  output y;
  reg y;

  always @( a or b or c)
  begin
    y = (a&b)|(b&c)|(a&c);
  end

endmodule
```

Figure 3.28 Procedural Block Describing a Majority Circuit

Majority Example with Delay. The Verilog code shown in Figure 3.29 is a majority circuit with a 5ns delay. Following the **always** keyword, the statements in this procedural block are an event-control, a delay-control and a procedural assignment. The delay-control statement begins with a sharp-sign and is followed by a delay value. This statement causes the flow into this procedural block to be suspended for 5ns. This means that after an event on one of the circuit inputs, evaluation and assignment of the output value to *y* takes place after 5 nanoseconds.

Note in the description of Figure 3.29 that **begin** and **end** bracketing is not used. As with the event-control statement, a delay-control statement can collapse into its next statement by removing their separating semicolon. The event-control, delay-control and assignment to *y* become a single procedural statement in the **always** block of *maj3* code.

```
`timescale 1ns/100ps

module maj3 ( a, b, c, y );
  input a, b, c;
  output y;
  reg y;

  always @( a or b or c ) #5 y = (a & b) | (b &c) | (a & c);

endmodule
```

Figure 3.29 Majority Gate with Delay

Full-Adder Example. Another example of using procedural assignments in a procedural block is shown in Figure 3.30. This example describes a full-adder with sum and carry-out outputs.

The **always** block shown is sensitive to *a, b,* and *ci* inputs. This means that when an event occurs on any of these inputs, the **always** block wakes up and executes all its statements in the order that they appear. Since assignments to *s* and *co* outputs are procedural, both these outputs are declared as **reg.**

The delay mechanism used in the full-adder of Figure 3.30 is called an intra-statement delay that is different than that of the majority circuit of Figure 3.29.

```
`timescale 1ns/100ps

module add_1 bit ( a, b, ci, s, co );
  input a, b, ci;
  output s, co;
  reg s, co;

  always @(a or b or ci )
  begin
    s = #5 a ^ b ^ ci;
    co = #3 (a & b) I (b &ci) I (a & ci);
  end
endmodule
```

Figure 3.30 Full-Adder Using Procedural Assignments

In the majority circuit, the delay simply delays execution of its next statement. However, the intra-statement delay of Figure 3.30 only delays the assignment of the calculated value of the right-hand side to the left-hand side variable. This means that in Figure 3.30, as soon as an event occurs on an input, the expression *a^b^c* is evaluated. But, the assignment of the evaluated value to *s* and proceeding to the next statement takes 5ns.

Because assignment to *co* follows that to *s*, the timing of the former depends on that of the latter, and evaluation of the right-hand side of *co* begins 5ns after an input change. Therefore, *co* receives its value 8ns after an input change occurs. To remove this timing dependency and be able to define the timing of each statement independent of its previous one, a different kind of assignment must be used.

```
`timescale 1ns/100ps

module add_1bit ( a, b, ci, s, co );
  input a, b, ci;
  output s, co;
  reg s, co;

  always @(a or b or ci)
  begin
    s <= #5 a ^ b ^ ci;
    co <= #8 (a & b) I (b &ci) I (a & ci);
  end
endmodule
```

Figure 3.31 Full-Adder Using Non-Blocking Assignments

Assignments in Figure 3.30 are of the blocking type. Such statements block the flow of the program until they are completed. A different assignment is of the non-blocking type. A different version of the full-adder that uses this construct is shown in Figure 3.31. This assignment schedules its right hand side value into its left hand side to take place after the specified delay. Program flow continues into the next statement while propagation of values into the first left hand side is still going on.

In the example of Figure 3.31, evaluation of the right hand side of *s* is done immediately after an input changes. Evaluation of the right hand side of *co* occurs 8ns after that. To make *s* and *co* delays match those of Figure 3.30, an 8 nanoseconds delay is used for assignment to *co*.

Since our focus is on synthesizable coding and gate delay timing issues are not of importance, we will mostly use blocking assignments in this book.

Procedural Multiplexer Example. For another example of a procedural block, consider the 2-to-1 multiplexer of Figure 3.32. This example uses an **if-else** construct to set *y* to *i0* or *i1* depending on the value of s.

As in the previous examples, all circuit variables that participate in determination of value of *y* appear on the sensitivity list of the **always** block. Also since *y* appears on the left hand side of a procedural assignment, it is declared as **reg.**

The **if-else** statement shown in Figure 3.32 has a condition part that uses an equality operator. If the condition is false (or equal to **0**), the block of statements that follow it will be taken, otherwise block of statements after the **else** are taken. In both cases, the block of statements must be bracketed by **begin** and **end** keywords if there is more than one statement in a block.

```
module mux2_1 ( i0, i1 , s, y );
  input i0, i1, s;
  output y;
  reg y;

  always @( i0 or i1 or s ) begin
    if (s==1'b0)
      y = i0;
    else
      y = i1;
  end
endmodule
```

Figure 3.32 Procedural Multiplexer

Procedural ALU Example. The **if-else** statement, used in the previous example, is easy to use, descriptive and expandable. However, when many choices exist, a **case**-statement which is more structured may be a better choice. The ALU description of Figure 3.33 uses a **case** statement to describe an ALU with add, subtract, AND and XOR functions.

```
module alu_4bit ( a, b, f, y );
  input [3:0] a, b;
  input [1:0] f;
  output [3:0] y;
  reg [3:0] y;

  always @ ( a or b or f ) begin
    case (f)
      2'b00 : y = a + b;
      2'b01 : y = a - b;
      2'b10 : y = a & b;
      2'b11 : y = a ^ b;
      default: y = 4'b0000;
    endcase
  end
endmodule
```

Figure 3.33 Procedural ALU

The ALU has *a* and *b* data inputs and a 2-bit *f* input that selects its function. The Verilog code shown in Figure 3.33 uses *a, b* and *f* on its sensitivity list. The **case**-statement shown in the **always** block uses *f* to select one of the **case** alternatives. The last alternative is the **default** alternative that is taken when *f* does not match any of the alternatives that appear before it. This is necessary to make sure that unspecified input values (here, those that contain **X** and/or **Z**) cause the assignment of the default value to the output and not leave it unspecified.

3.2.4 Combinational Rules

Completion of **case** alternatives or **if-else** conditions is an important issue in combinational circuit coding. In an **always** block, if there are conditions under which the output of a combinational circuit is not assigned a value, because of the property of **reg** variables the output retains its old value. The retaining of old value infers a latch on the output. Although, in some designs this latching is intentional, obviously it is unwanted when describing combinational circuits. With this, we have set two rules for coding combinational circuits with **always** blocks.

1. List all inputs of the combinational circuit in the sensitivity list of the **always** block describing it.
2. Make sure all combinational circuit outputs receive some value regardless of how the program flows in the conditions of **if-else** and/or **case** statements. If there are too many conditions to check, set all outputs to their inactive values at the beginning of the **always** block.

3.2.5 Bussing

Bus structures can be implemented by use of multiplexers or three-state logic. In Verilog, various methods of describing combinational circuits can be used for the description of a bus.

Figure 3.34 shows Verilog coding of *busout* that is a three-state bus and has three sources, *busin1, busin2*, and *busin3*. Sources of *busout* are put on this bus by active-high enabling control signals, *en1, en2* and *en3*. Using the value of an enabling signal, a condition statement either selects a bus driver or a 4-bit **Z** value to drive the *busout* output.

```
module bussing (busin1, busin2, busin3, en1, en2, en3, busout );
  input [3:0] busin1, busin2, busin3;
  input en1, en2, en3;
  output [3:0] busout;

   assign busout = en1 ? busin1 : 4'bzzzz;
   assign busout = en2 ? busin2 : 4'bzzzz;
   assign busout = en3 ? busin3 : 4'bzzzz;

endmodule
```

Figure 3.34 Implementing a 3-State Bus

Verilog allows multiple concurrent drivers for **net**s. However, a variable declared as a **reg** and used on a left hand side in a procedural block **(always** block), can only be driven by one source. This makes the use of **net**s more appropriate for representing busses.

3.3 Sequential Circuits

As with any digital circuit, a sequential circuit can be described in Verilog by use of gates, Boolean expressions, or behavioral constructs (e.g., the **always** statement). While gate level descriptions enable a more detailed description of timing and delays, because of complexity of clocking and register and flip-flop controls, these circuits are usually described by use of procedural **always** blocks. This section shows various ways sequential circuits are described in Verilog. The following discusses primitive structures like latch and flip-flops, and then generalizes coding styles used for representing these structures to more complex sequential circuits including counters and state machines.

3.3.1 Basic Memory Elements at the Gate Level

A clocked D-latch latches its input data during an active clock cycle. The latch structure retains the latched value until the next active clock cycle. This element is the basis of all static memory elements.

A simple implementation of the D-latch that uses cross-coupled NOR gates is shown in Figure 3.35. The Verilog code of Figure 3.36 corresponds to this D-latch circuit. This description uses primitive **and** and **nor** structures.

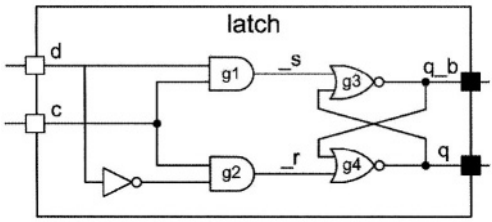

Figure 3.35 Clocked D-latch

```
`timescale 1ns/100ps

module latch ( d, c, q, q_b );
  input d, c;
  output q, q_b;
  wire _s, _r;

   and #(6)   g1(_s, c, d),
              g2(_r, c, ~d);
   nor #(4)   g3(q_b, _s, q),
              g4 ( q,  _r,  q_b );
endmodule
```

Figure 3.36 Verilog Code for a Clocked D-latch

As shown in this Verilog code, the tilde (~) operator is used to generate the complement of the *d* input of the latch. Using AND gates, the *d* input and its complement are gated to generate internal *_s* and *_r* inputs. These are inputs to the cross-coupled NOR structure that is the core of the memory in this latch.

Alternatively, the same latch can be described with an **assign** statement as shown below.

```
assign #(3) q = c ? d : q;
```

This statement simply describes what happens in a latch. The statement says that when *c* is **1**, the *q* output receives *d*, and when *c* is **0** it retains its old value. Using two such statements with complementary clock values describe a master-slave flip-flop. As shown in Figure 3.37, the *qm* **net** is the master output and *q* is the flip-flop output.

```
`timescale 1ns/100ps

module master_slave ( d, c, q );
  input d, c;
  output q;

  wire qm;

  assign #(3) qm = c ? d : qm;
  assign #(3) q = ~c ? qm : q;

endmodule
```

Figure 3.37 Master-Slave Flip-Flop

This code uses two concurrent **assign** statements. As discussed before, these statements model logic structures with **net** driven outputs (*qm* and q). The order in which the statements appear in the body of the *master_slave* **module** is not important.

3.3.2 Memory Elements Using Procedural Statements

Although latches and flip-flops can be described by primitive gates and **assign** statements, such descriptions are hard to generalize, and describing more complex register structures cannot be done this way. This section uses **always** statements to describe latches and flip-flops. We will show that the same coding styles used for these simple memory elements can be generalized to describe memories with complex control as well as functional register structures like counters and shift-registers.

```
module latch ( d, c, q, q_b );
  input d, c;
  output q, q_b;
  reg q, q_b;

  always @ ( c or d )
    if ( c ) begin
      #4 q = d;
      #3 q_b = ~d;
    end
endmodule
```

Figure 3.38 Procedural Latch

Latches. Figure 3.38 shows a D-latch described by an **always** statement. The outputs of the latch are declared as **reg** because they are being driven inside the **always** procedural block. Latch clock and data inputs (*c* and *d*) appear in the sensitivity list of the **always** block, making this procedural statement sensitive to *c* and *d*. This means that when an event occurs on *c* or *d*, the

always block wakes up and it executes all its statements in the sequential order from begin to end.

The **if**-statement enclosed in the **always** block puts *d* into *q* when *c* is active. This means that if *c* is **1** and *d* changes, the change on *d* propagates to the *q* output. This behavior is referred to as transparency, which is how latches work. While clock is active, a latch structure is transparent, and input changes affect its output.

Any time the **always** statement wakes up, if *c* is **1**, it waits 4 nanoseconds and then puts *d* into *q*. It then waits another 3 nanoseconds and then puts the complement of *d* into *q_b*. This makes the delay of the *q_b* output 7 ns.

D Flip-Flop. While a latch is transparent, a change on the D-input of a D flip-flops does not directly pass on to its output. The Verilog code of Figure 3.39 describes a positive-edge trigger D-type flip-flop.

The sensitivity list of the procedural statement shown includes **posedge** of c*lk*. This **always** statement only wakes up when c*lk* makes a **0** to **1** transition. When this statement does wake up, the value of *d* is put into *q*. Obviously this behavior implements a rising-edge D flip-flop.

```
`timescale 1ns/100ps

module d_ff ( d, clk, q, q_b );
   input d, clk;
   output q, q_b;
   reg q, q_b;

   always @ ( posedge clk )
   begin
      #4 q = d;
      #3 q_b = ~d;
   end
endmodule
```

Figure 3.39 A Positive-Edge D Flip-Flop

Instead of **posedge,** use of **negedge** would implement a falling-edge D flip-flop. After the specified edge, the flow into the **always** block begins. In our description, this flow is halted by 4 nanoseconds by the #4 delay-control statement. After this delay, the value of *d* is read and put into *q*. Following this transaction, the flow into the **always** block is again halted by 3 nanoseconds, after which *~d* is put into *qb*. This makes the delay of *q* after the edge of the clock equal to 4 nanoseconds. The delay for *q_b* becomes the accumulation of the delay values shown, and it is 7 nanoseconds. Delay values are ignored in synthesis.

Synchronous Control. The coding style presented for the above simple D flip-flop is a general one and can be expanded to cover many features found in flip-flops and even memory structures. The description shown in Figure 3.40 is a D-type flip-flop with synchronous set and reset (*s* and *r*) inputs.

The description uses an **always** block that is sensitive to the positive-edge of c*lk*. When c*lk* makes a **0** to **1** transition, the flow into the **always** block begins. Immediately after the positive-edge, *s* is inspected and if it is active (**1**), after 4 ns *q* is set to **1** and 3 ns after that *q_b* is set to **0**. Following the positive-edge of *clk*, if *s* is not **1**, *r* is inspected and if it is active, *q* is set to **0**. If neither *s* nor *r* are **1**, the flow of the program reaches the last **else** part of the **if**-statement and assigns *d* to *q*.

The behavior discussed here only looks at *s* and *r* on the positive-edge of *clk,* which corresponds to a rising-edge trigger D-type flip-flop with synchronous active high set and reset inputs. Furthermore, the set input is given a higher priority over the reset input. The flip-flop structure that corresponds to this description is shown in Figure 3.41.

Other synchronous control inputs can be added to this flip-flop in a similar fashion. A clock enable *(en)* input would only require inclusion of an **if**-statement in the last **else** part of the **if**-statement in the code of Figure 3.40.

```
module d_ff ( d, s, r, clk, q, q_b );
  input d, clk, s, r;
  output q, q_b;
  reg q, q_b;

  always @ ( posedge clk ) begin
    if ( s ) begin
      #4q = 1'b1;
      #3q_b=1'b0;
    end else if ( r ) begin
      #4q = 1'b0;
      #3q_b=1'b1;
    end else begin
      #4 q = d;
      #3 q_b = ~d;
    end
  end

endmodule
```

Figure 3.40 D Flip-Flop with Synchronous Control

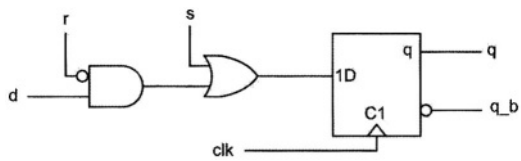

Figure 3.41 D Flip-Flop with Synchronous Control

Asynchronous Control. The control inputs of the flip-flop of Figure 3.40 are synchronous because the flow into the **always** statement is only allowed to start when the **posedge** of *clk* is observed. To change this to a flip-flop with asynchronous control, it is only required to include asynchronous control inputs in the sensitivity list of its procedural statement.

Figure 3.42 shows a D flip-flop with active high asynchronous set and reset control inputs. Note that the only difference between this description and the code of Figure 3.40 (synchronous control) is the inclusion of **posedge** *s* and **posedge** *r* in the sensitivity list of the **always** block. This inclusion allows the flow into the procedural block to begin when *clk* becomes **1** or *s* becomes **1** or *r* becomes **1**. The **if**-statement in this block checks for *s* and *r* being **1**, and if none are active (activity levels are high) then clocking *d* into *q* occurs.

An active high (low) asynchronous input requires inclusion of **posedge** (**negedge**) of the input in the sensitivity list, and checking its **1** (**0**) value in the **if**-statement in the **always** statement. Furthermore, clocking activity in the flip-flop (assignment of *d* into *q*) must always be the last choice in the **if**-statement the procedural block.

The graphic symbol corresponding to the flip-flop of Figure 3.42 is shown in Figure 3.43.

```
module d_ff ( d, s, r, clk, q, q_b );
   input d, clk, s, r;
   output q, q_b;
   reg q, q_b;
   always @ (posedge clk or posedge s or posedge r )
   begin
     if ( s ) begin
       #4 q = 1'b1;
       #3 q_b = 1'b0;
     end else if ( r ) begin
       #4 q = 1'b0;
       #3 q_b = 1'b1;
     end  else  begin
       #4 q = d;
       #3 q_b = ~ d;
     end
   end
endmodule
```

Figure 3.42 D Flip-Flop with Asynchronous Control

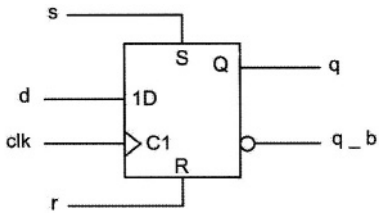

Figure 3.43 Flip-Flop with Asynchronous Control Inputs

3.3.3 Registers, Shifters and Counters

Registers, shifter-registers, counters and even sequential circuits with more complex functionalities can be described by simple extensions of the coding styles presented for the flip-flops. In most cases, the functionality of the circuit only affects the last **else** of the **if**-statement in the procedural statement of codes shown for the flip-flops.

Registers. Figure 3.44 shows an 8-bit register with synchronous set and reset Inputs. The *set* input puts all **1**s in the register and the *reset* input resets it to all **0**s. The main difference between this and the flip-flop with synchronous control is the vector declaration of inputs and outputs.

```
module register (d, clk, set, reset, q);
  input [7:0] d;
  input clk, set, reset;
  output [7:0] q;
  reg [7:0] q;

  always @ ( posedge clk )
  begin
    if ( set )
      #5 q = 8'b1;
    else if ( reset )
      #5 q = 8'b0;
    else
      #5 q = d;
  end

endmodule
```

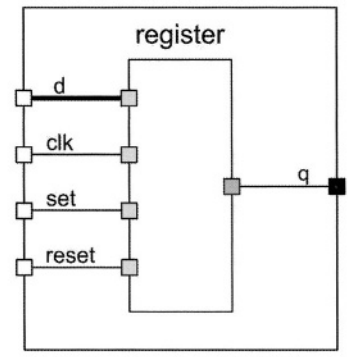

Figure 3.44 An 8-bit Register

Shift-Registers. A 4-bit shift-register with right- and left-shift capabilities, a serial-input, synchronous reset input, and parallel loading capability is shown in Figure 3.45. As shown, only the positive-edge of *clk* is included in the sensitivity list of the **always** block of this code, which makes all activities of the shift-register synchronous with the clock input. If *rst* is **1**, the register is reset, if *ld* is **1** parallel *d* inputs are loaded into the register, and if none are **1** shifting left or right takes place depending on the value of the *l_r* input (**1** for left, **0** for right). Shifting in this code is done by use of the concatenation operator { }. For left-shift, *s_in* is concatenated to the right of *q[2:0]* to form a 4-bit vector that is put into *q*. For right-shift, *s_in* is concatenated to the left of *q[3:1]* to form a 4-bit vector that is clocked into *q[3:0]*.

The style used for coding this register is the same as that used for flip-flops and registers presented earlier. In all these examples, a single procedural block handles function selection (e.g., zeroing, shifting, or parallel loading) as well as clocking data into the register output.

```
module shift_reg (d, clk, ld, rst, l_r, s_in, q);
   input [3:0] d;
   input clk, ld, rst, l_r, s_in;
   output [3:0] q;
   reg [3:0]q;

   always @( posedge clk ) begin
      if (rst)
         #5 q = 4'b0000;
      else if ( ld )
         #5 q = d;
      else if ( l_r )
         #5 q = {q[2:0], s_in};
      else
         #5 q = {s_in, q[3:1]};
   end

endmodule
```

Figure 3.45 A 4-bit Shift Register

Another style of coding registers, shift-registers and counters is to use a combinational procedural block for function selection and another for clocking.

As an example, consider a shift-register that shifts *s_cnt* number of places to the right or left depending on its *sr* or *sl* control inputs (Figure 3.46). The shift-register also has an *ld* input that enables its clocked parallel loading. If no shifting is specified, i.e., *sr* and *sl* are both zero, then the shift register retains its old value.

The Verilog code of Figure 3.46 shows two procedural blocks that are identified by *combinational* and *register.* A block name appears after the **begin** keyword that begins a block and is separated from this keyword by use of a colon. Figure 3.47 shows a graphical representation of the coding style used for the description of our shifter.

The *combinational* block is sensitive to all inputs that can affect the shift register output. These include the parallel *d_in,* the *s_cnt* shift-count, *sr* and *sl* shift control inputs, and the *ld* load control input. In the body of this block an **if-else** statement decides on the value placed on the *int_q* internal variable. The value selection is based on values of *ld, sr,* and *sl.* If *ld* is **1**, *int_q* becomes *d_in* that is the parallel input of the shift register. If *sr* or *sl* is active, *int_q* receives the previous value of *int_q* shifted to right or left as many as *s_cnt* places. In this example, shifting is done by use of the >> and << operators. On the left, these operators take the vector to be shifted, and on the right they take the number of places to shift.

The *int_q* variable that is being assigned values in the *combinational* block is a 4-bit **reg** that connects the output of this block to the input of the register block.

The *register* block is a sequential block that handles clocking *int_q* into the shift register output. This block (as shown in Figure 3.46) is sensitive to the positive edge of *clk* and its body consists of a single **reg** assignment.

Note in this code that both *q* and *int_q* are declared as **reg** because they are both receiving values in procedural blocks.

```
module shift_reg ( d_in, clk, s_cnt, sr, sl, ld, q );
   input [3:0] d_in;
   input clk, sr, sl, ld;
   input [1:0] s_cnt;
   output [3:0] q;
   reg [3:0] q, int_q;

   always @ ( d_in or s_cnt or sr or sl or ld ) begin: combinational
      if ( ld ) int_q = d_in;
      else if ( sr ) int_q = int_q >> s_cnt;
      else if ( sl ) int_q = int_q << s_cnt;
      else int_q = int_q;
   end

   always @ ( posedge clk ) begin: register
      q = int_q;
   end

endmodule
```

Figure 3.46 Shift-Register Using Two Procedural Blocks

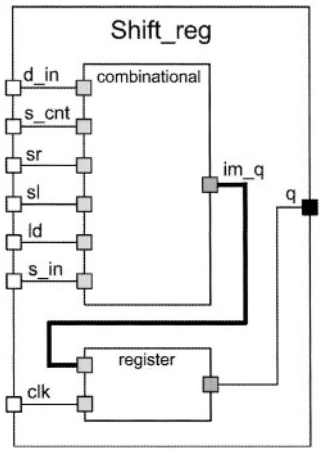

Figure 3.47 Shifter Block Diagram

Counters. Any of the styles described for the shift-registers in the previous discussion can be used for describing counters. A counter counts up or down, while a shift-register shifts right or left. We use arithmetic operations in counting as opposed to shift or concatenation operators in shift-registers.

Figure 3.48 shows a 4-bit up-down counter with a synchronous *rst* reset input. The counter has an *ld* input for doing the parallel loading of *d_in* into the counter. The counter output is *q* and it is declared as **reg** since it is receiving values within a procedural statement.

Discussion about synchronous and asynchronous control of flip-flops and registers also apply to the counter circuits. For example, inclusion of ***posedge rst*** in the sensitivity list of the counter of Figure 3.48 would make its resetting asynchronous.

```
module counter (d_in, clk, rst, ld, u_d, q );
   input [3:0] d_in;
   input clk, rst, ld, u_d;
   output [3:0] q;
   reg [3:0] q;

   always @ ( posedge clk ) begin
      if ( rst)
         q = 4'b0000;
      else if ( ld )
         q = d_in;
      else if ( u_d )
         q = q + 1;
      else
         q = q - 1;
   end

endmodule
```

Figure 3.48 An Up-Down Counter

3.3.4 State Machine Coding

Coding styles presented so far can be further generalized to cover finite state machines of any type. This section shows coding for Moore and Mealy state machines. The examples we will use are simple sequence detectors. These circuits represent the controller part of a digital system that has been partitioned into a data path and a controller. The coding styles used here apply to such controllers, and will be used in later chapters of this book to describe CPU and multiplier controllers.

Moore Detector. State diagram for a Moore sequence detector detecting **101** on its *x* input is shown in Figure 3.49. The machine has four states that are labeled, *reset, got1, got10,* and *got101.* Starting in *reset,* if the **101** sequence is detected, the machine goes into the *got101* state in which the output becomes **1.** In addition to the *x* input, the machine has a *rst* input that forces the machine into its *reset* state. The resetting of the machine is synchronized with the clock.

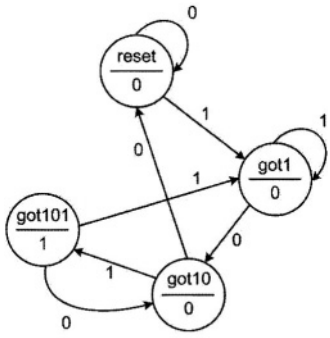

Figure 3.49 A Moore Sequence Detector

```
module moore_detector (x, rst, clk, z );
 input x, rst, clk;
 output z; reg z;
 parameter [1:0]  reset = 0, got1 = 1, got10 = 2, got101 = 3;
 reg [1:0] current;
 always @ ( posedge clk ) begin
  if (rst) begin
   current = reset;  z = 1'b0;
  end
  else case ( current )
   reset:  begin
        if ( x==1 'b1 ) current = got1 ;
        else current = reset;  z = 1'b0;
       end
   got1: begin
        if ( x==1'b0 ) current = got10;
        else current = got1; z = 1'b0;
       end
   got10:  begin
        if ( x==1'b1 ) begin
         current = got101; z=1'b1;
        end else begin
         current = reset;  z = 1'b0;
        end
       end
   got101: begin
        if ( x==1'b1 ) current = got1;
        else current = got10;
        z = 1'b0;
       end
  endcase
 end
endmodule
```

Figure 3.50 Moore Machine Verilog Code

The Verilog code of the Moore machine of Figure 3.49 is shown in Figure 3.50. After the declaration of inputs and outputs of this module, **parameter** declaration declares four states of the machine as two-bit parameters. The square-brackets following the **parameter** keyword specify the size of parameters being declared. Following parameter declarations in the code of Figure 3.50, the two-bit *current* **reg** type variable is declared. This variable holds the current state of the state machine.

The **always** block used in the module of Figure 3.50 describes state transitions and output assignments of the state diagram of Figure 3.49. The main task of this procedural block is to inspect input conditions (values on *rst* and *x*) during the present state of the machine defined by *current* and set values into *current* for the next state of the machine.

The flow into the **always** block begins with the positive edge of *clk*. Since all activities in this machine are synchronized with the clock, only *clk* appears on the sensitivity list of the **always** block. Upon entry into this block, the *rst* input is checked and if it is active, *current* is set to *reset* (*reset* is a declared parameter and its value is **0**). The value put into *current* in this pass through the **always** block gets checked in the next pass with the next edge of the clock. Therefore this assignment is regarded as the next-state assignment. When this assignment is made, the **if-else** statements skip the rest of the code of the **always** block, and this **always** block will next be entered with the next positive edge of *clk*.

Upon entry into the **always** block, if *rst* is not **1**, program flow reaches the **case** statement that checks the value of *current* against the four states of the machine. Figure 3.51 shows an outline of this **case**-statement.

```
case ( current )
    reset:  begin . . . end
    got1:   begin . . . end
    got10:  begin . . . end
   got101:  begin . . . end
endcase
```

Figure 3.51 case-Statement Outline

The **case**-statement shown has four **case**-alternatives. A **case**-alternative is followed by a block of statements bracketed by the **begin** and **end** keywords. In each such block, actions corresponding to the active state of the machine are taken.

Figure 3.52 shows the Verilog code of the *got10* state and its diagram from the state diagram of Figure 3.49. As shown here, the **case**-alternative that corresponds to the *got10* state only specifies the next values for the state and output of the circuit.

Note, for example, that the Verilog code segment of state *got10* does not specify the output of this state. Instead, the next value of *current* and the next value of *z* are specified based on the value of *x*. If *x* is **1**, the next state becomes *got101* in which *z* is **1**, and if *x* is **0**, the next state becomes *reset*.

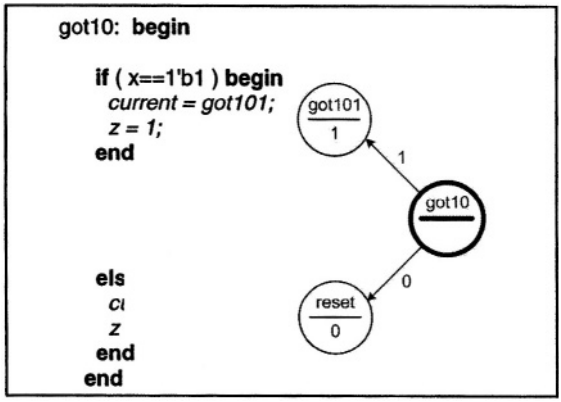

Figure 3.52 Next Values from *got10*

In this coding style, for every state of the machine there is a **case**-alternative that specifies the next state values. For larger machines, there will be more **case**-alternatives, and more conditions within an alternative. Otherwise, this style can be applied to state machines of any size and complexity.

This same machine can be described in Verilog in many other ways. We will show alternative styles of coding state machines by use of examples that follow.

A Mealy Machine Example. Unlike a Moore machine that has outputs that are only determined by the current state of the machine, in a Mealy machine, the outputs are determined by the state the machine is in as well as the inputs of the circuit. This makes Mealy outputs not fully synchronized with the circuit clock. In the state diagram of a Mealy machine the outputs are specified along the edges that branch out of the states of the machine.

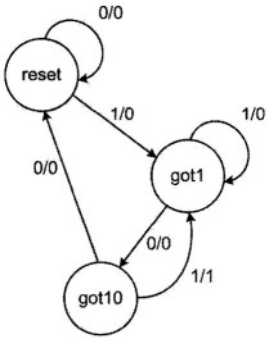

Figure 3.53 A 101 Mealy Detector

Figure 3.53 shows a **101** Mealy detector. The machine has three states, *reset, got1* and *got10*. While in *got10,* if the *x* input becomes **1** the machine prepares to go to its next state with the next clock. While waiting for the clock, its output becomes **1**. While on the edge that takes the machine out of *got10*, if the clock arrives the machine goes into the *got1* state. This machine allows overlapping sequences. The machine has no external resetting mechanism. A sequence of two zeros on input *x* puts the machine into the *reset* state in a maximum of two clocks.

The Verilog code of the **101** Mealy detector is shown in Figure 3.54. After input and output declarations, a **parameter** declaration defines bit patterns (state assignments) for the states of the machine. Note here that state value 3 or **11** is unused. As in the previous example, we use the *current* two-bit **reg** to hold the current state of the machine.

After the declarations, an **initial** block sets the initial state of the machine to *reset*. This procedure for initializing the machine is only good for simulation and is not synthesizable.

This example uses an **always** block for specifying state transitions and a separate statement for setting values to the *z* output. The **always** statement responsible for state transitions is sensitive to the circuit clock and has a **case** statement that has **case** alternatives for every state of the machine. Consider for example, the *got10* state and its corresponding Verilog code segment, as shown in Figure 3.55.

```
module mealy_detector ( x, clk, z );
 input x, clk;
 output z;
 parameter [1:0]
  reset = 0, // 0 = 0 0
  got1  = 1, // 1 = 0 1
  got10 = 2; // 2 = 1 0

 reg [1:0] current;

 initial current = reset;
 always @ ( posedge clk )
 begin
  case ( current )
   reset:  if ( x==1 'b1 ) current = got1;
      else current = reset;
   got1:  if ( x==1'b0 ) current = got10;
      else current = got1;
   got10:  if ( x==1 'b1 ) current = got1;
       else current = reset;
    default: current = reset;
   endcase
 end
 assign z= ( current==got10 && x==1'b1 ) ? 1'b1 : 1'b0;

endmodule
```

Figure 3.54 Verilog Code of 101 Mealy Detector

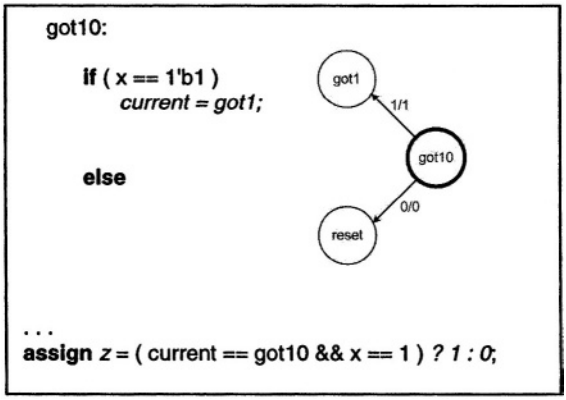

Figure 3.55 Coding a Mealy State

As shown, the Verilog code of this state only specifies its next states and does not specify the output values. Notice also in this code segment that the **case** alternative shown does not have **begin** and **end** bracketing. Actually, **begin** and **end** keywords do not appear in blocks following **if** and **else** keywords either.

Verilog only requires **begin** and **end** bracketing if there is more than one statement in a block. The use of this bracketing around one statement is optional. Since the **if** part and the **else** part each only contain one statement, **begin** and **end** keywords are not used. Furthermore, since the entire **if-else** statement reduces to only one statement, the **begin** and **end** keywords for the **case**-alternative are also eliminated.

The last **case**-alternative shown in Figure 3.54 is the **default** alternative. When checking *current* against all alternatives that appear before the **default** statement fail, this alternative is taken. There are several reasons that we use this default alternative. One is that, our machine only uses three of the possible four 2-bit assignments and **11** is unused. If the machine ever begins in this state, the default case makes *reset* the next state of the machine. The second reason why we use **default** is that Verilog assumes a four-value logic system that includes **Z** and **X**. If *current* ever contains a **Z** or **X**, it does not match any of the defined case alternatives, and the default case is taken. Another reason for use of **default** is that our machine does not have a hard reset and we are making provisions for it to go to the *reset* state. The last reason for **default** is that it is just a good idea to have it.

The last statement in the code fragment of Figure 3.55 is an **assign** statement that sets the *z* output of the circuit. This statement is a concurrent statement and is independent of the **always** statement above it. When *current* or *x* changes, the right hand side of this assignment is evaluated and a value of **0** or **1** is assigned to *z*. Conditions on the right hand side of this assignment are according to values put in *z* in the state diagram of Figure 3.54. Specifically, the output is **1** when *current* is *got10* and *x* is **1**, otherwise it is **0**. This statement implements a combinational logic structure with *current* and *x* inputs and *z* output.

Huffman Coding Style. The Huffman model for a digital system characterizes it as a combinational block with feedbacks through an array of registers. Verilog coding of digital systems according to the Huffman model uses an **always** statement for describing the register part and another concurrent statement for describing the combinational part.

We will describe the state machine of Figure 3.49 to illustrate this style of coding. Figure 3.56 shows the combinational and register part partitioning that we will use for describing this machine. The *combinational* block uses *x* and *p_state* as input and generates *z* and *n_state.* The *register* block clocks *n_state* into *p_state,* and reset *p_state* when *rst* is active.

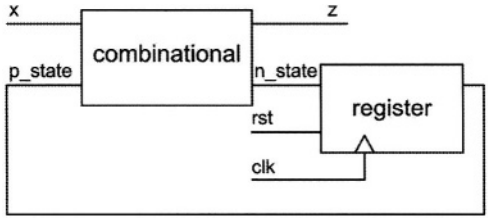

Figure 3.56 Huffman Partitioning of 101 Moore Detector

Figure 3.57 shows the Verilog code of Figure 3.49 according to the partitioning of Figure 3.56. As shown, parameter declaration declares the states of the machine. Following this declaration, *n_state* and *p_state* variables are declared as two-bit **regs** that hold values corresponding to the states of the **101** Moore detector. The *combinational* **always** block follows this **reg** declaration. Since this a purely combinational block, it is sensitive to all its inputs, namely *x* and *p_state.* Immediately following the block heading, *n_state* and *z* are set to their inactive or reset values. This is done so that these variables are always reset with the clock to make sure they do not retain their old values. As discussed before, retaining old values implies latches, which is not what we want in our combinational block.

The body of the combinational **always** block of Figure 3.57 contains a **case**-statement that uses the *p_state* input of the **always** block for its **case**-expression. This expression is checked against the states of the Moore machine. As in the other styles discussed before, this **case**-statement has **case**-alternatives for *reset, got1, got10,* and *got101* states.

In a block corresponding to a **case**-alternative, based on input values, *n_state* and *z* output are assigned values. Unlike the other styles where *current* is used both for the present and next states, here we use two different variables, *p_state* and *n_state.*

The next procedural block shown in Figure 3.57 handles the register part of the Huffman model of Figure 3.56. In this part, *n_state* is treated as the register input and *p_state* as its output. On the positive edge of the clock, *p_state* is either set to the *reset* state (**00**) or is loaded with contents of *n_state.*

Together, *combinational* and *register* blocks describe our state machine in a very modular fashion.

```
module moore_detector ( x, rst, clk, z );
  input x, rst, clk;
  output z;
  reg z;
  parameter [1:0]
    reset = 2'b00, got1 = 2'b01, got10 = 2'b10, got101 = 2'b11;

  reg [1:0] p_state, n_state;

  always @ ( p_state or x ) begin : combinational
    n_state = 0; z = 0;
    case ( p_state )
      reset:  begin
          if( x==1'b1 ) n_state = got1;
          else n_state = reset; z = 1'b0;
        end
      got1:  begin
          if( x==1'b0 ) n_state = got10;
          else n_state = got1; z = 1'b0;
        end
      got10:  begin
          if( x==1'b1 ) n_state = got101;
          else n_state = reset; z = 1 'b0;
      end
      got101:  begin
          if( x==1'b1 ) n_state = got1;
          else n_state = got10; z = 1 'b1;
      end
      default: n_state = reset;
    endcase
  end

  always @ ( posedge clk ) begin : register
    if ( rst ) p_state = reset;
    else p_state = n_state;
  end

endmodule
```

Figure 3.57 Verilog Huffman Coding Style

The advantage of this style of coding is in its modularity and defined tasks of each block. State transitions are handled by the *combinational* block and clocking is done by the *register* block. Changes in clocking, resetting, enabling or presetting the machine only affect the coding of the *register* block. If we were to change the synchronous resetting to asynchronous, the only change we had to make was adding ***posedge*** *rst* to the sensitivity list of the register block.

A More Modular Style. For a design with more input and output lines and more complex output logic, the *combinational* block may further be partitioned into a

block for handling transitions and another for assigning values to the outputs of the circuit. For coding both of these blocks, it is necessary to follow the rules discussed for combinational blocks in Section 3.2.4.

```
module mealy_detector ( x, en, clk, rst, z );
 input x, en, clk, rst; output z; reg z;
 parameter [1:0]  reset = 0, got1 = 1, got10 = 2, got11 = 3;

 reg [1:0] p_state, n_state;

 always  @( p_state or x ) begin : Transitions
  n_state = reset;
  case ( p_state )
   reset:  if ( x == 1'b1  ) n_state = got1;
      else n_state = reset;
   got1: if ( x == 1'b0 ) n_state = got10;
      else n_state = got11;
   got10: if ( x == 1'b1  ) n_state = got1;
      else n_state = reset;
   got11: if ( x == 1'b1 ) n_state = got11;
      else n_state = got10;
   default:  n_state = reset;
  endcase
 end

 always  @(p_state or x)  begin: Outputting
  z = 0;
  case ( p_state )
   reset:  z = 1'b0;
   got1: z = 1'b0;
   got10:  if (x == 1'b1 ) z = 1'b1;
      else z = 1'b0;
   got11:  if ( x==1'b1 ) z = 1'b0;
      else z = 1'b1;
   default: z = 1'b0;
  endcase
 end

 always  @  ( posedge clk )  begin: Registering
  if ( rst ) p_state = reset;
  else if( en ) p_state = n_state;
 end

endmodule
```

Figure 3.58 Separate Transition and Output Blocks

Figure 3.58 shows the coding of the **110-101** Moore detector using two separate blocks for assigning values to *n_state* and the *z* output. In a situation like what we have in which the output logic is fairly simple, a simple **assign** statement could replace the *outputting* procedural block. In this case, *z* must be a **net** and not a **reg**.

The examples discussed above, in particular, the last two styles, show how combinational and sequential coding styles can be combined to describe very complex digital systems.

3.3.5 Memories

Verilog allows description and use of memories. Memories are two-dimensional variables that are declared as **reg.** Verilog only allows **reg** data types for memories. Figure 3.59 shows a **reg** declaration declaring *mem* and its corresponding block diagram. This figure also shows several valid memory operations.

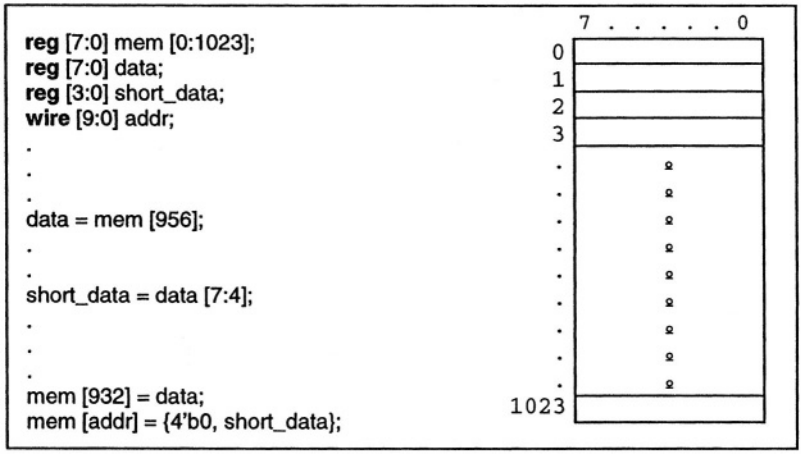

Figure 3.59 Memory Representation

Square brackets that follow the **reg** keyword specify the word-length of the memory. The square brackets that follow the name of the memory (*mem*), specify its address space. A memory can be read by addressing it within its address range, e.g., *mem[956].* Part of a word in a memory cannot be read directly, i.e., slicing a memory word is not possible. To read part of a word, the whole word must first be read in a variable and then slicing done on this variable. For example, *data[7:4]* can be used after a memory word has been placed into *data.*

With proper indexing, a memory word can be written into by placing the memory name and its index on the left hand side of an assignment, e.g., *mem[932] = data;* , memories can also be indexed by **reg** or **net** type variables, e.g., *mem[addr],* when *addr* is a 10-bit address bus. Writing into a part of the memory is not possible. In all cases data directly written into a memory word affects all bits of the word being written into. For example to write the four-bit *short_data* into a location of *mem*, we have to decide what goes into the other four bits of the memory word.

Figure 3.60 shows a memory block with separate input and output busses. Writing into the memory is clocked, while reading from it only requires *rw* to be **1.** An **assign** statement handles reading and an **always** block performs writing into this memory.

```
module memory (inbus, outbus, addr, clk, rw);
   input [7:0] inbus;
   input [9:0] addr;
   output [7:0] outbus;
   input clk, rw;

   reg [7:0] mem [0:1023];

   assign outbus = rw ? mem [addr] : 8'bz;

   always @ (posedge clk)
      if (rw == 0) mem [addr] = inbus;

endmodule
```

Figure 3.60 Memory Description

3.4 Writing Testbenches

Verilog coding styles discussed so far were for coding hardware structures, and in all cases synthesizability and direct correspondence to hardware were our main concerns. On the other hand, testbenches do not have to have hardware correspondence and they usually do not follow any synthesizability rules. We will see that delay specifications, and **initial** statements that do not have a one-to-one hardware correspondence are used generously in testbenches.

For demonstration of testbench coding styles, we use the Verilog code of Figure 3.61 that is a **101** Moore detector, as the circuit to be tested.

This description is functionally equivalent to that of Figure 3.50. The difference is in the use of condition expressions (**?:**) instead of **if-else** statements, and separating the output assignment from the main **always** block. This code will be instantiated in the testbenches that follow.

3.4.1 Generating Periodic Data

Figure 3.62 shows a testbench module that instantiates *moore_detector* and applies test data to its inputs. The first statement in this code is the **'timescale** directive that defines the time unit of this description. The testbench itself has no ports, which is typical of all testbenches. All data inputs to a circuit-under-test are locally generated in its testbench.

```
module moore_detector ( x, rst, clk, z );
   input x, rst, clk;
   output z;
   parameter [1:0] a=0, b=1, c=2, d=3;
   reg [1:0] current;

   always @( posedge clk )
      if ( rst )    current = a;
      else case ( current )
         a : current = x ? b : a ;
         b : current = x ? b : c ;
         c : current = x ? d : a ;
         d : current = x ? b : c ;
         default : current = a;
      endcase
   assign z = (current==d) ? 1'b1 : 1'b0;
endmodule
```

Figure 3.61 Circuit Under Test

Because we are using procedural statements for assigning values to ports of the circuit-under-test, all variables mapped with the input ports of this circuit are declared as **reg.** The testbench uses two **initial** blocks and two **always** blocks. The first initial block initializes *clock, x,* and *reset* to **0, 0,** and **1** respectively. The next **initial** block waits for 24 time units (ns in this code), and then sets *reset* back to **0** to allow the state machine to operate.

The **always** blocks shown produce periodic signals with different frequencies on *clock* and *x.* Each block waits for a certain amount of time and then it complements its variable. Complementing begins with the initial values of *clock* and *x* as set in the first **initial** block. We are using different periods for *clock* and *x,* so that a combination of patterns on these circuit inputs is seen. A more deterministic set of values could be set by specifying exact values at specific times.

```
`timescale 1 ns / 100 ps

module test_moore_detector;
   reg x, reset, clock;
   wire z;
   moore_detector uut ( x, reset, clock, z );
   initial begin
      clock=1'b0; x=1'b0; reset=1'b1;
   end
   initial #24 reset=1'b0;
   always #5 clock=~clock;
   always #7 x=~x;
endmodule
```

Figure 3.62 Generating Periodic Data

3.4.2 Random Input Data

Instead of the periodic data on *x* we can use the **$random** predefined system function to generate random data for the *x* input. Figure 3.63 shows such a testbench.

This testbench also combines the two **initial** blocks for initially activating and deactivating *reset* into one. In addition, this testbench has an **initial** block that finishes the simulation after 165 ns.

When the flow into a procedural block reaches the **$finish** system task, the simulation terminates and exits. Another simulation control task that is often used is the **$stop** task that only stops the simulation and allows resumption of the stopped simulation run.

```
`timescale 1 ns / 100 ps

module test_moore_detector;
  reg x, reset, clock;
  wire z;
  moore_detector uut( x, reset, clock, z );
  initial  begin
     clock=1'b0; x=1'b0; reset=1'b1;
     #24 reset=1'b0;
  end
  initial #165 $finish;
  always #5 clock=~clock;
  always #7 x=~x;
endmodule
```

Figure 3.63 Random Data Generation

3.4.3 Synchronized Data

Independent data put into various inputs of a circuit may not be random enough to be able to catch many design errors. Figure 3.64 shows another testbench for our Moore detector that only reads random data into the *x* input after the positive edge of the *clock*.

The third **initial** statement shown in this code uses the **forever** construct to loop forever. Every time when the positive edge of *clock* is detected, after 3 nanoseconds a new random value is put into *x*. The **initial** statement in charge of clock generation uses a **repeat** loop to toggle the *clock* 13 times every 5 nanoseconds and stop. This way, after *clock* stops, all activities cease, and the simulation run terminates. For this testbench we do not need a simulation control task.

The testbench of Figure 3.64 uses an invocation of **$monitor** task to display the contents of the *current* state of the sequence detector every time it changes. The **initial** statement that invokes this task puts it in the background and every time *uut.current* changes, **$monitor** reports its new value. The *uut.current* name is a hierarchical name that uses the instance name of the circuit-under-test to look at its internal variable, *current*.

```
`timescale 1ns/100ps

module test_moore_detector;
  reg x, reset, clock;
  wire z;

  moore_detector uut( x, reset, clock, z );

  initial begin
    clock=1'b0; x=1'b0; reset=1'b1;
    #24 reset=1'b0;
  end
  initial repeat(13) #5 clock=~clock;
  initial forever @ (posedge clock) #3 x=$random;
  initial $monitor("New state is %d and occurs at %t", uut.current, $time);
  always @ (z) $display("Output changes at %t to %b", $time, z);

endmodule
```

Figure 3.64 Synchronized Test Data

The **$monitor** task shown also reports the time that *current* takes a new value. This time is reported by the **$time** task. The *uut.current* variable uses the decimal format (**%d**) and **$time** is reported using the time format (**%t**). Binary, Octal and Hexadecimal output can be obtained by using **%b**, **%o**, and **%h** format specifications.

The last statement in this testbench is an **always** statement that is sensitive to *z*. This statement uses the **$display** task to report values put on *z*. The **$display** task is like the **$monitor,** except that it only becomes active when flow into a procedural block reaches it. When *z* changes, flow into the **always** statement begins and the **$display** task is invoked to display the new value of *z* and its time of change. This output is displayed in binary format. Using **$monitor** inside an **initial** statement, for displaying *z* (similar to that for *uut.current*) would result in exactly the same thing as the **$display** inside an **always** block that is sensitive to *z*.

3.4.4 Applying Buffered Data

Examples discussed above use random or semi-random data on the *x* input of the circuit being tested. It is possible that we never succeed in giving *x* appropriate data to generate a **1** on the *z* output of our sequence detector. To correct this situation, we define a buffer, put the data we want in it and continuously apply this data to the *x* input.

Figure 3.65 shows another testbench for our sequence detector of Figure 3.61. In this testbench the 5-bit *buff* variable is initialized to contain **10110.** The **initial** block that follows the clock generation block, rotates concatenation of *x* and *buff* one place to the right 3 nanoseconds after every time the clock ticks. This process repeats for as long as the circuit clock ticks.

```
`timescale 1ns/100ps

module test_moore_detector;
  reg x, reset, clock;
  wire z;

  reg [4:0] buff;
  initial buff = 5'b10110;

  moore_detector uut( x, reset, clock, z );

  initial begin
    clock=1'b0; x=1'b0; reset=1'b1;
    #24 reset=1'b0;
  end

  initial repeat(18) #5 clock=~clock;
  initial forever @(posedge clock) #3 {buff,x}={x,buff};
  initial forever @(posedge clock) #1 $display(z, uut.current);

endmodule
```

Figure 3.65 Buffered Test Data

The last **initial** statement in this description outputs *z* and *uut.current* 1 nanosecond after every time the clock ticks. The **$display** task used for this purpose is unformatted which defaults to the decimal data output.

3.4.5 Timed Data

A very simple testbench for our sequence detector can be done by applying test data to *x* and timing them appropriately to generate the sequence we want, very similar to the way values were applied to *reset* in the previous examples. Figure 3.66 shows this simple testbench.

Techniques discussed in the above examples are just some of what one can do for test data generation. These techniques can be combined for more complex examples. After using Verilog for some time, users form their own test generation techniques. For small designs, simulation environments generally provide waveform editors and other tool-dependent test generation schemes. Some tools come with code fragments that can be used as templates for testbenches.

An important issue is developing testbenches is external file IO. Verilog allows the use of **$readmemh** and **$readmemb** system tasks for reading hex and binary test data into a declared memory. Moreover, for writing responses from a circuit-under-test to an external file, **$fdisplay** can be used. Examples for these features of the language will be shown in Chapters 11 and 14.

```
`timescale 1ns/100ps

module test_moore_detector;
  reg x, reset, clock;
  wire z;

  moore_detector uut( x, reset, clock, z);

  initial begin
    clock=1'b0; x=1'b0; reset=1'b1;
    #24 reset=1'b0;
  end

  always #5 clock=~clock;

  initial begin
    #7  x=1;
    #5  x=0;
    #18 x=1;
    #21 x=0;
    #11 x=1;
    #13 x=0;
    #33 $stop;
  end

endmodule
```

Figure 3.66 Timed Test Data Generation

3.5 Synthesis Issues

Verilog constructs described in this chapter included those for cell modeling as well as those for designs to be synthesized. In describing an existing cell, timing issues are important and must be included in the Verilog code of the cell. At the same time, description of an existing cell may require parts of this cell to be described by interconnection of gates and transistors. On the other hand, a design to be synthesized does not include any timing information because this information is not available until the design is synthesized, and designers usually do not use gates and transistors for high level descriptions for synthesis.

Considering the above, taking timing out of the descriptions, and only using gates when we really have to, the codes presented in this chapter all have one-to-one hardware correspondence and are synthesizable. For synthesis, a designer must consider his or her target library to see what and how certain parts can be synthesized. For example, most FPGAs do not have internal three-state structures and three-state bussings are converted to AND-OR busses.

3.6 Summary

This chapter presented the Verilog HDL language from a hardware design point of view. The chapter used complete design examples at various levels of abstraction for showing ways in which Verilog could be used in a design. We showed how timing details could be incorporated in cell descriptions. Aside from this discussion of timing, all examples that were presented had one-to-one hardware correspondence and were synthesizable. We have shown how combinational and sequential components can be described for synthesis and how a complete system can be put together using combinational and sequential blocks for it to be tested and synthesized.

This chapter did not cover all of Verilog, but only the most often used parts of the language.

This page intentionally left blank