6

MODELING AT DATA FLOW LEVEL

6.1 INTRODUCTION

Gate level design description makes use of the gate primitives available in Verilog. These are repeatedly and judiciously instantiated to achieve the full design description. Digital designers familiar with the basic logic gates and SSI / MSI circuits can describe the desired target circuit in terms of them on paper and proceed with the design description based on them. This was the approach followed in the last two chapters; it is practical for comparatively smaller designs – say those involving tens of gates. One can define modules in terms of primitives involving tens of gates and instantiate them in macro-modules. This increases the complexity of designs that can be handled by one order. Beyond that the gate level design description becomes too complicated to be practical.

Data flow level description of a digital circuit is at a higher level. It makes the circuit description more compact as compared to design through gate primitives. We have a number of operands and operations representing the simulations directly or indirectly. The operations are carried out on the operand(s) in singles or in combinations [IEEE]. The results are assigned to nets. The operand-operation-assignments representing data flow are carried out repeatedly to complete the design description [Thomas & Morby]. Further, these can be combined judiciously with the gate instantiations wherever necessary. With such combinations, design description of a comprehensive nature can be accommodated.

6.2 CONTINUOUS ASSIGNMENT STRUCTURES

A simple two input AND gate in data flow format has the form

```
assign c = a && b;
```

Here

- "**assign**" is the keyword carrying out the assignment operation. This type of assignment is called a continuous assignment.

Design Through Verilog HDL. T. R. Padmanabhan and B. Bala Tripura Sundari

ISBN: 0-471-44148-1

- a and b are operands – typically single-bit logic variables.
- "&&" is a logic operator. It does the bit-wise AND operation on the two operands a and b.
- "=" is an assignment activity carried out.
- c is a net representing the signal which is the result of the assignment.

In general, an operand can be of any one of the following types:

- A constant number [including real].
- Net of a scalar or vector type including part of a vector.
- Register variable of a scalar or vector type including part of a vector.
- Memory element.
- A call to a function that returns any of the above. The function itself can be a user-defined or of a system type [see Chapter 9].

There are other types of operators as well [see Section 6.5]. All types of combinational circuits can be modeled using continuous assignments. One need not necessarily resort to instantiation of gate primitives.

An AND gate module which uses the above assignment is shown in Figure 6.1. The test bench for the same is shown in Figure 6.2, and the waveforms of nets a, b, and c obtained with the simulation are shown in Figure 6.3. [The simulation software used has the facility to capture the waveforms of selected signals in the "run" phase; this has been invoked to get the waveforms in Figure 6.3. No separate **$monitor** command is included in the test bench of Figure 6.2. The same approach has been adopted with many of the test benches elsewhere in the book.]

Multiple assignments can be carried out through a direct extension of the structure adopted in the above case. Consider the AOI gate in Figure 6.4. A few patterns of the assignments for the circuit are given in Figure 6.5 to Figure 6.7.

```
module andgdf(c,a,b);
output c;
input a,b;
wire c;
assign c = a&&b;
endmodule
```

Figure 6.1 A module with an AND gate at the data flow level.

```
module tst_andgdf; //TESTBENCH
reg a,b;
wire c;
initial
begin
```

continued

continued

```
      a = 1'b0;
      b = 1'b0;
 #4   a = 1'b1;
 #4   b = 1'b1;
 #4   a = 1'b0;
 #4   b = 1'b0;
 #4   a = 1'b1;
end
andgdf g1(c,a,b);
initial #20 $stop;
endmodule
```

Figure 6.2 A test bench for the module in Figure 6.1.

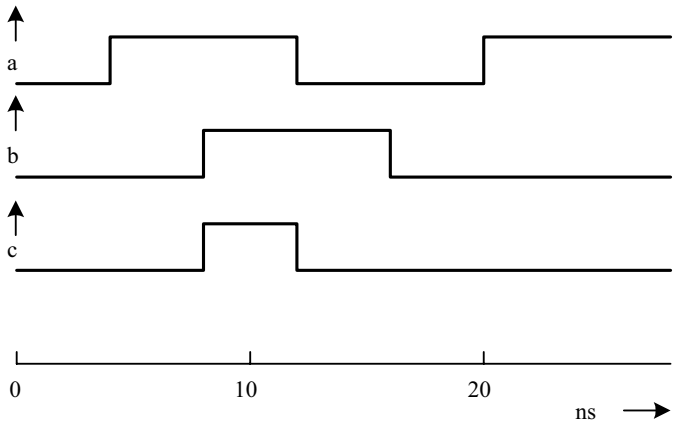

Figure 6.3 Waveforms of nets a, b, and c obtained with the simulation of the module in Figure 6.1 with the test bench in Figure 6.2.

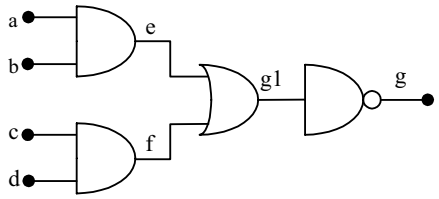

Figure 6.4 An A-O-I gate circuit.

```
assign e = a&&b, f = c&&d, g1 = e|f, g = ~g1;
```

Figure 6.5 A data flow level assignment statement to realize the A-O-I gate in Figure 6.4.

```
assign e = a & b, f = c & d;
assign g1 = e|f, g = ~g1;
```

Figure 6.6 Another set of data flow level assignment statements to realize the A-O-I gate in Figure 6.4.

```
assign e  = a & b;
assign f  = c & d;
assign g1 = e ! f;
assign g  = ~g1;
```

Figure 6.7 Yet another set of data flow level assignment statements to realize the A-O-I gate in Figure 6.4.

Observations:

- The semicolon terminates an assignment statement. Commas separate different assignments in an assignment statement.
- "|" is the bit-wise OR operator and "~" the bit-wise negation operator in Verilog.
- All the quantities in the left-hand side of a continuous assignment have to be of net type. Thus e, f, g, and g1 have to be declared as nets.
- All the operations in an assignment are evaluated whenever any of the operands in the assignment changes value. Further, all the assignments are carried out concurrently. Hence the order of the assignments or the statements is immaterial.
- The right-hand sides of assignment statements can be nets, regs, or function calls. Here a, b, c, and d can be nets or regs. All other variables have to be nets.

The module for the A-O-I gate of Figure 6.4 is given in Figure 6.8 – it is formed around the assignment statement of Figure 6.5. The same can be tested through a test bench.

6.2.1 Combining Assignment and Net Declarations

The assignment statement can be combined with the net declaration itself making the assignment implicit in the net declaration itself. Thus the two statements

```
wire c;
assign c = a & b;
```

can be combined as

```
wire c = a & b;
```

The above simplification cannot be carried over to multiple declarations. With this proviso, the module of Figure 6.8 can be modified as shown in Figure 6.9. In the modules of Figures 6.8 and 6.9, a, b, c, and d are declared as **input** and g as **output**. As was explained in Section 4.2, these would be taken as nets if there are no separate declarations concerning their types. However, the intermediate quantities – e, f, and g1– should be declared as **wire**. Synthesized version of the A-O-I circuit is shown in Figure 6.10.

```
module aoi2(g,a,b,c,d);
output g;
input a,b,c,d;
wire e,f,g1,g;
assign e = a && b,f = c && d, g1 = e||f, g=~g1;
endmodule
```

Figure 6.8 A compact description of the AOI module at the data flow level.

```
module aoi3(g,a,b,c,d);
output g;
input a,b,c,d;
wire g;
wire  e  = a && b;
wire  f  = c && d;
wire  g1 = e||f;
assign g = ~g1;
endmodule
```

Figure 6.9 Alternate design module to realize the A-O-I gate in Figure 6.4.

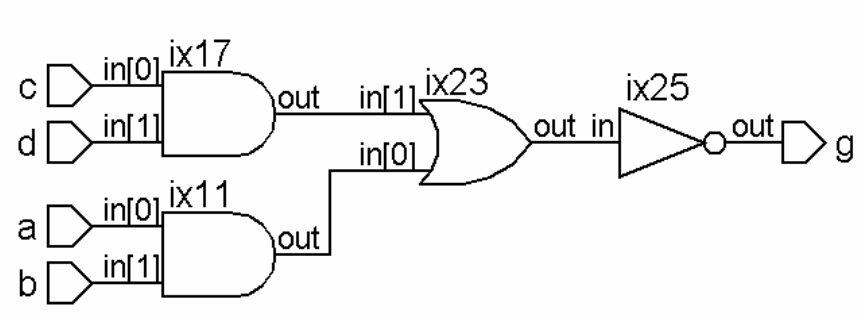

Figure 6.10 Synthesized circuit of the A-O-I gate module of Figure 6.9.

6.2.2 Continuous Assignments and Strengths

A net to which a continuous assignment is being made can be assigned strengths for its logic levels. The procedure is akin to the strength allocation to the outputs of primitives. The AOI gate of Figure 6.9 is modified with strength allocations to the output and is shown in Figure 6.11. The assignment to g can be combined with the wire declaration into a single statement as

```
wire (pull1, strong0) g = ~g1;
```

As mentioned earlier, one can have only one assignment in the statement here. In a bigger design, g in Figure 6.11 can be assigned to other expressions or primitives also. Any resulting contention in the output values will be resolved on the lines discussed in Chapter 4.

```
module aoi4 (g, a, b, c, d);
output g;
input a, b, c, d;
wire g;
wire e  = a &&b;
wire f  = c & &d;
wire g1 = e || f;
assign (pull1, strong0) g = ~g1;
endmodule
```

Figure 6.11 The module of Figure 6.9 modified with strength allocation to the output.

6.3 DELAYS AND CONTINUOUS ASSIGNMENTS

Delays can be incorporated at the data flow level in different ways [Ciletti]. Consider the combination of statements in Figure 6.12. The assignment takes effect with a time delay of 2 time steps. If a or b changes in value, the program waits for 2 time steps, computes the value of c based on the values of a and b at the time of computation, and assigns it to c. In the interim period, a or b may change further, but c changes and takes the new value only 2 time steps after the change in a or b initiates it. Typical waveforms for a, b, and c are shown in Figure 6.13. Note that the changes in a and b of duration less than 2 time steps are ignored *vis-à-vis* assignment to the net c. The following may be noted with respect to the waveforms:

- a changes at 0 ns, 2 ns, 5 ns, 8 ns, 9 ns, 12 ns and 13 ns; b changes at 0 ns, 2 ns, 6 ns, 8 ns and 13 ns. All these trigger changes to c.
- In every case change to c comes into effect with a time delay of 2 time steps – that is, at the 2nd, 4th, 7th, 8th, 10th, 11th, 14th and 15th ns, respectively.
- Whenever c changes, its new value is decided by the values of a and b at that instant of time. In effect, c changes at 2nd, 4th and 7th ns only.

```
wire c, a, b;
assign #2     c = a & b;
```

Figure 6.12 Illustration of combining delays with assignments.

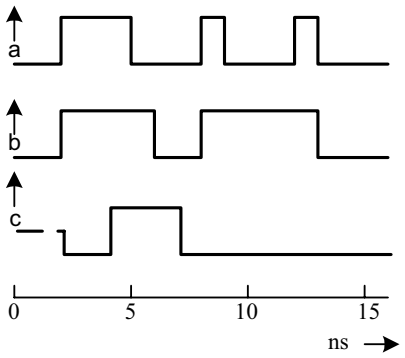

Figure 6.13 Waveforms of signals a, b, and c for the design segment of Figure 6.12.

The program segment in Figure 6.14 also gives the same output as shown in Figure 6.13. If the time delay is in the net and not in the assignment proper, its effect is not any different. Consider the program segment in Figure 6.15. Here the changes in the values of d are computed immediately following those in a and b. The assignment takes effect immediately. The delay in the net c causes a delay of 2 time steps in the assignment to c. Such a delay is not present for d. Typical waveforms for the program segment are shown in Figure 6.16. Note the following:

- a changes at 2 ns, 5 ns, 8 ns, 9 ns, 12 ns and 13 ns; b changes at 2 ns, 6 ns, 8 ns and 13 ns. All these trigger changes to c and d also.
- In every case, change to c comes into effect with a time delay of 2 time steps – that is, in effect, c changes at 2nd, 4th and 7th ns only.
- Whenever c changes, its new value is decided by the values of a and b at that instant of time.
- In every case, changes to d come into effect immediately.

```
wire a, b;
wire  #2 c = a & b;
```

Figure 6.14 Alternate description for the program segment of Figure 6.10.

```
wire a, b, d;
wire #2      c;
assign c = a & b;
assign d = a & b;
```

Figure 6.15 Illustration of combining delays with assignments.

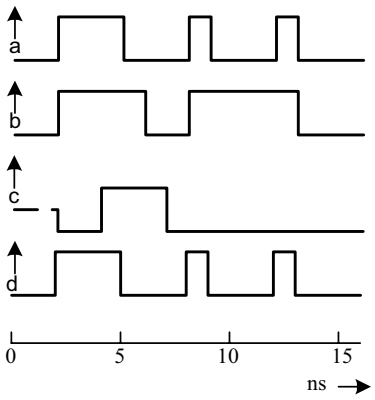

Figure 6.16 Waveforms of Signals a, b, c, and d for the design segment of Figure 6.15.

6.4 ASSIGNMENT TO VECTORS

The continuous assignments are equally applicable to vectors. A single statement can describe operations involving vectors wherever possible. This is illustrated in the adder module in Figure 6.17, which adds two 8-bit numbers. Here it is assumed that the sum is also of 8 bits. However to account for the possibility of a carry bit being generated in the course of the addition process, it is desirable to increase the vector size of c by one bit.

6.4.1 Concatenation of Vectors

One can concatenate vectors, scalars, and part vectors to form other vectors. The concatenated vector is enclosed within braces. Commas separate the components –scalars, vectors, and part vectors. If a and b are 8- and 4-bit wide vectors, respectively and c is a scalar

{a, b, c}

stands for a concatenated vector of 13 bits width. The vector components are formed in the order shown – c is the least significant bit and a[7] the most significant bit and the other bits are in between in the order specified. The concatenation can be with selected segments of vectors also. For example,

{a(7:4), b(2:0)}

represents a 7-bit vector formed by combining the 4 most significant bits of vector a with the 3 least significant bits of vector b. The size of each operand within the braces has to be specified fully to form the concatenated vector. Hence unsized constant numbers cannot be used as operands here.

Example 6.1 Eight-Bit Adder

Figure 6.18 shows the design description of an 8-bit adder, where the output vector is formed directly by concatenation. The adder takes a carry input and gives out a carry output. The adder module here can form the "seed" adder block in a multi-byte adder chain.

```
module add_8(a,b,c);
input[7:0]a,b;
output[7:0]c;
assign c = a + b ;
endmodule
```

Figure 6.17 An adder module at data flow level where the nets are vectors.

```
module add_8_c(c,cco,a,b,cci);
input[7:0]a,b;
output[7:0]c;
input cci;
output cco;
assign {cco,c} = (a + b + cci);
endmodule
```

Figure 6.18 A complete 8-bit adder module at data flow level.

When it is necessary to replicate vectors, scalars, *etc.,* to form other vectors, the same can be arrived at in a compact manner using the repetition multiplier again through concatenation. Thus,

{2{p}}

represents the concatenated vector

{p, p}

and

{2{p}, q}

represents the concatenated vector

{p, p, q}.

The two statements

```
assign GND=supply0;
p={8{GND}};
```

together ground the 8 bits of the vector p.

Concatenation operation can be nested to form bigger vectors when component combinations are repeated. For example,

{a, 3 {2{b , c}, d}}

is equivalent to the vector

{a, b, c, b, c, d, b, c, b, c, d, b, c, b, c, d }

6.5 OPERATORS

A set of operators is available in Verilog. The operator symbols are similar to those in C language [Gottfried]. With these operators we can carry out specified operations on the operands and assign the results to a net or a vector set of nets as the case may be. A few such operands have already been used in the examples so far. We discuss here the different operators, their types, and the operations carried out by each. Subsequently the use of operators is illustrated through a set of examples.

6.5.1 Unary Operators

Unary operators do an operation on a single operand and assign the result to the specified net. The unary operators in Verilog are given in Table 6.1. All unary operators get precedence over binary and ternary operators. The operators "+" and "–" preceding an integer or a real number change its sign. These are also unary operators, though not separately listed in Table 6.1.

6.5.2 Binary Operators

Most operators available are of the binary type. A binary operator takes on two operands; the operator comes in between the two operands in the assignment. The binary operators are grouped into type categories and discussed separately. The following are to be noted:

- The arithmetic operators treat both the operands as numbers and return the result as a number.
- All net and **`reg`** operand values are treated as unsigned numbers.
- Real and integer operands may be signed quantities.
- If either of the operand values has a zero value, the entire result has a zero value (?).

The result of any arithmetic operation — with the "+" or "–" or with any of the other arithmetic operators discussed later — will have an **`x`** value if any of the operand bits has an **`x`** or a **`z`** value.

6.5.2.1 Arithmetic Operators

The arithmetic operators of the binary type are given in Table 6.2. The modulus operand is similar to that in C language – It provides the remainder of the division

Table 6.1 Unary operators and their symbols

Operator type	Symbol	Remarks
Logical negation	!	Only for scalars
Bit-wise negation	~	For scalars and vectors
Reduction AND	&	For vectors – yields a single bit output
Reduction NAND	~&	
Reduction OR	|	
Reduction NOR	~|	
Reduction XOR	^	
Reduction XNOR	~^ or ^~	

Table 6.2 Arithmetic operators and their symbols

Operand type	Symbol	Remarks
Multiplication	*	
Division	/	The result is `x` if the denominator is zero
Modulus	%	
Addition	+	
Subtraction	–	

of two numbers. The module in Figure 6.17 is an example of the illustration of the use of the arithmetic binary operator "+" (for addition). Other arithmetic operators are also used in a similar manner.

Observations:

- In integer division the fractional part of the result is truncated and ignored.
- If any bit of an operand is **`x`** or **`z`** in an arithmetic operation, the result takes the **`x`** value.
- If the first operand of a modulus operator is negative, the result is also a negative number.

Depending on the type of definition of a number, a modulus operation can lead to different results. Typical examples are given in Table 6.3.

6.5.2.2 Logical Operators

There are two logical operators involving two operands. The operands concerned can be variables or expressions involving variables. In both cases the result of the operation is a single bit of value 1 (true) or 0 (false). If a bit in one of the operands is x or z, the result of evaluation of the expression has an **`x`** value. The operator details are shown in Table 6.4. The modules in Figure 6.8 and Figure 6.9 are examples of the illustration of the use of logical binary operators.

6.5.2.3 Relational Operators

There are four relational operators; their details are shown in Table 6.5. A relational operator treats both the operands as binary numbers and compares them. The result is a 1 (true) bit or a 0 (false) bit. If a bit in either operand is **`x`** or **`z`**, the result has **`x`** (unknown) value. The operands can be variables or expressions involving variables. Operands of net or **`reg`** type are treated as unsigned numbers. Real and integers can be positive or negative (*i.e.*, signed) numbers.

Table 6.3 Typical modulus operations and their results

Expressions involving modulus operator	Result of the operation	Remarks
15 % 5	0	Results are obvious
14 % 5	4	
4'hf % 5	0	The numbers 4'hf and 4'he are in hex format with decimal values of 15 and 14, respectively. But the denominator 5 is in decimal form.
4'he % 5	4	
6'o15 % 5	3	6'o15 is an octal number with a decimal value of 13.
–4 % 3	–1	
4 % –3		Illegal form

Table 6.4 Binary logical operators and their symbols

Operator type	Symbol	Possible output value
AND	&&	Single-bit output
OR	||	

Table 6.5 Relational operators and their symbols

Operator type	Symbol	Possible output value
Greater than	>	Single-bit output
Less than	<	
Greater than or equal to	>=	
Less than or equal to	<=	

6.5.2.4 Equality Operators

The equality operator makes a bit-by-bit comparison of the two operands and produces a result bit. The result bit is a 1 (true) if the operand condition is satisfied; otherwise it is 0 (false). The operands can be variables or expressions involving variables. If the operands are of unequal length, the shorter one is zero filled to match the larger operand. The operators in this category are only of two types – those to test the equality and those to test inequality. The four operators in this category are given in Table 6.6.

6.5.2.5 Bit-wise Logical Operators

The operator does a specified bit-by-bit operation on the two operands and produces a set of result bits. The result is (bit-wise) as wide as the wider operand.

Table 6.6 Equality operators and their symbols

Operand symbol	Description of operand	Possible logical value of result
==	(The symbol comprises two consecutive equal signs.) If the two operands are equal bit by bit, the result is 1 (true); else the result is 0 (false). If either operand has a **x** or **z** bit, the result is **x**.	0, 1, or **x**
!=	(The symbol comprises of an exclamation mark followed by an equal sign.) A bit-by-bit comparison of the two operands is made. The result is a 1 if there is a mismatch for at least one bit position.	0, 1, or **x**
===	(The symbol comprises of three consecutive equal signs.) The operand bits can be 0, 1, **x**, or **z**. If the two operands match on a bit by bit basis, the result is a 1 (true) bit; else it is 0 (false). Note that the result is never **x** here.	0 or 1
!==	(The symbol comprises an exclamation mark followed by 2 consecutive equal signs). The operand bits can be 0, 1, **x**, or **z**. If the two operands do not match on a bit by bit basis, the result is a 1 (true) bit; else it is 0 (false). Note that the result is never **x** here.	0 or 1

If the width of one of the operands is less than that of the other, it is bit-extended by filling zero bits and the widths are matched. Subsequently, the specified operation is carried out. If one of the operands has an **x** or **z** bit in it, the corresponding result bit is **x**. Either operand can be a single variable or an expression involving variables. Table 6.7 gives the four operators of this category.

6.5.2.6 Operator Truth Table

The truth tables for different types of bit-wise operators are given in Table 6.8. Note that an **z** input is treated as an **x** value (Compare these with their counterparts for respective gate primitives in Chapter 4.)

Table 6.7 Bit-wise logical operators and their symbols

Operator type	Symbol	Possible result
AND	&	0, 1, or **x**
OR	|	
XOR	^	
XNOR	~^ or ^~	

Table 6.8 Truth tables for bit-wise operators

AND

Input 1 \ Input 2	0	1	x
0	0	0	0
1	1	0	x
x	0	x	x

OR

Input 1 \ Input 2	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

XOR

Input 1 \ Input 2	0	1	x
1	1	0	x
x	x	x	x

XNOR

Input 1 \ Input 2	0	1	x
0	1	1	x
x	x	x	x

Negation

Input	0	1	x
Output	1	0	x

6.5.2.7 Shift Operators

Table 6.9 shows the two operators of this category. The << operator executes left shift operation, while the >> operator executes the right shift operation. In either case the operand specified on the left is shifted by the number of bits specified on the right. The shifting is done irrespective of whether the bits are 0, 1, **x**, or **z**. The bits shifted out are lost. The vacated positions created as a result of the shifting are filled with zeroes. If the right operand is **x** or **z**, the result has an x value. If the right operand is negative, the left operand remains unchanged.

6.5.3 Ternary Operator

Verilog has only one ternary operator – the conditional operator. It checks a condition and does a branching. It is a versatile and powerful operator. It enhances the potential of design description substantially (as can be seen through the examples below). The general form is

A?b:c

The conditional operation is made up of two operators – "?" and ":" – and three operands. The two operands separate the three operators in the order shown. The operational sequence of the operation is as follows:

Table 6.9 Shift-type operators and their symbols

Operand	Typical usage	Operation
>>	A >> b	The set of bits representing A are shifted right repeatedly b times.
<<	A<< b	The set of bits representing A are shifted left repeatedly b times.

- "A" is evaluated first.
- If A is true, b is evaluated.
- If A is false, c is evaluated.

If A evaluates to an ambiguous result, both b and c are evaluated. Then they are combined on a bit-by-bit basis to form the resultant bit stream. The result bit can have the following three possible values:

- 0 if the corresponding bits of b and c are 0.
- 1 if the corresponding bits of b and c are 1.
- **x** otherwise.

As an example, consider the assignment statement

```
assign y = w ? x : z;
```

where w, x, y and z are binary bits. If the bit w is true (1), y is assigned the value of x: otherwise – that is, if w is false (0) – y is assigned the value of z. The assignment statement here multiplexes x and z onto y; w is the control signal here. Consider the assignment

```
assign flag = (adr1 == adr2)?1'b1 : 1'b0;
```

Here adr1 and adr2 are two multibit vectors representing two addresses. If the two are identical, the flag bit is set to zero; else it is reset.

```
assgn zero_flag = (|byte)? 0:1;
```

All the bits of the byte are ORed together here. The zero_flag is set if the result is zero.

```
assign c = s ? a: b; //The net c is connected to a if s=1; else it is connected to b
```

The statement realizes a 2 to 1 mux. b and c have to be scalars or vectors of the same width. The assignment can be expanded to realize larger muxes.

The conditional operator can be nested [see Figure 6.19]. Nesting gives rise to a variety of uses of the operator. As an example, consider the formation of an ALU. ALU can be defined in a compact manner using the ternary operator.

```
assign d =  (f==add)?(a+b): ((f==subtract)?(a-b): ((f==compl)?~a: ~b));
```

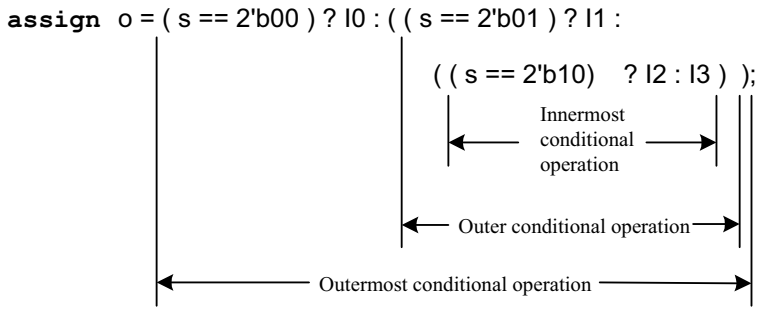

Figure 6.19 Illustration of nested conditional operations.

In the example here, f is taken as a control word. If it is equal to the number add, d is to be equal to the sum of a and b. If f is equal to the number subtract, d is to be equal to the difference between a and b. If it is equal to the number compl, d is to be the complement of a. Otherwise (*i.e.*, f = 3) d is taken as the complement of b. As another example consider a mux; the assignment statement in Figure 6.18 represents a 4-to-1 mux formed with a nested set of ternary operators. The construct in the figure can be judiciously used to form muxes of larger sizes.

Example 6.2 ALU

Figure 6.20 shows an ALU module. It is built around a single executable statement present as a continuous assignment. A test bench for the ALU is also shown in the figure. The synthesized circuit is shown in Figure 6.21. Results of running the test bench are shown in Figure 6.22. Some of the combinational circuit operations required are realized inside the "modgen" blocks of the FPGA used. The nature of the ALU description in the module decides the translation into circuit. Contrast this with the ALU considered at the gate level of design in Section 5.7 where each functional block is instantiated separately and the selected set of outputs steered to the final output. Each such instantiated module translates into a separate circuit block. Their outputs are mux'ed into the final output vector. There is a one-to-one correspondence between the elements of the design description and their respective realizations.

```
module alu_df1 (d, co, a, b, f,cci);
//a SIMPLE ALU FOR ILLUSTRATION PURPOSES
output [3:0] d;
output co;
wire[3:0]d;
```

continued

continued

```
wire co;
input cci;
input [3 : 0 ] a, b;
input [1 : 0] f;//f is a two-bit function select input;
assign {co,d}=(f==2'b00)?(a+b+cci):((f==2'b01)?(a-b)
          :((f==2'b10)? {1'bz,a^b}:{1'bz,~a}));
/*co is the carry bit in case of addition;it is the
borrow bit in case of subtraction. In the other two
cases, co is not required. Hence it is assigned z
value.*/
endmodule

module tst_aludf1; //test-bench
reg [3:0]a,b;
reg[1:0] f;
reg cci;
wire[3:0]d;
wire co;
alu_df1 aa(d,co,a,b,f,cci);
initial
begin
    cci= 1'b0;
    f  = 2'b00;
    a  = 4'b0;
    b  = 4'h0;
end
always
    begin
    #2 cci = 1'b0;f = 2'b00;a = 4'h1;b = 4'h0;
    #2 cci = 1'b1;f = 2'b00;a = 4'h8;b = 4'hf;
    #2 cci = 1'b1;f = 2'b01;a = 4'h2;b = 4'h1;
    #2 cci = 1'b0;f = 2'b01;a = 4'h3;b = 4'h7;
    #2 cci = 1'b1;f = 2'b10;a = 4'h3;b = 4'h3;
    #2 cci = 1'b1;f = 2'b11;a = 4'hf;b = 4'hc;
    end
initial $monitor($time, " cci = %b , a= %b ,b = %b ,
f = %b ,d =%b ,co= %b ",cci ,a,b,f,d,co);
initial #30 $stop;
endmodule
```

Figure 6.20 A 4-bit 4-function ALU and a test bench for the same.

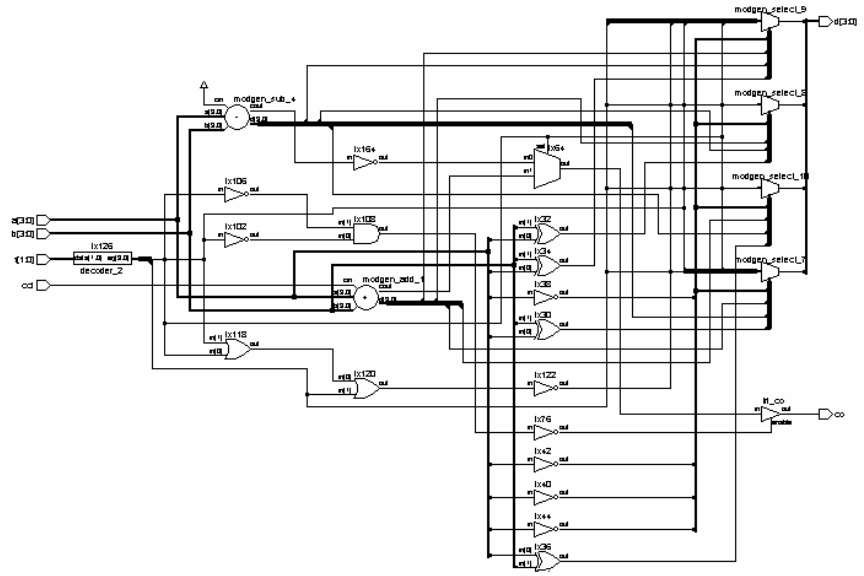

Figure 6.21 Synthesized circuit of the ALU in Example 6.18.

```
output listing
# 0 cci = 0 , a= 0000 ,b = 0000 ,f = 00 ,d =0000 ,co= 0
# 2 cci = 0 , a= 0001 ,b = 0000 ,f = 00 ,d =0001 ,co= 0
# 4 cci = 1 , a= 1000 ,b = 1111 ,f = 00 ,d =1000 ,co= 1
# 6 cci = 1 , a= 0010 ,b = 0001 ,f = 01 ,d =0001 ,co= 0
# 8 cci = 0 , a= 0011 ,b = 0111 ,f = 01 ,d =1100 ,co= 1
#10 cci = 1 , a= 0011 ,b = 0011 ,f = 10 ,d =0000 ,co= z
#12 cci = 1 , a= 1111 ,b = 1100 ,f = 11 ,d =0000 ,co= z
#14 cci = 0 , a= 0001 ,b = 0000 ,f = 00 ,d =0001 ,co= 0
#16 cci = 1 , a= 1000 ,b = 1111 ,f = 00 ,d =1000 ,co= 1
#18 cci = 1 , a= 0010 ,b = 0001 ,f = 01 ,d =0001 ,co= 0
#20 cci = 0 , a= 0011 ,b = 0111 ,f = 01 ,d =1100 ,co= 1
#22 cci = 1 , a= 0011 ,b = 0011 ,f = 10 ,d =0000 ,co= z
#24 cci = 1 , a= 1111 ,b = 1100 ,f = 11 ,d =0000 ,co= z
#26 cci = 0 , a= 0001 ,b = 0000 ,f = 00 ,d =0001 ,co= 0
#28 cci = 1 , a= 1000 ,b = 1111 ,f = 00 ,d =1000 ,co= 1
```

Figure 6.22 Results of running the test bench for the ALU module in Figure 6.20.

Example 6.3 Four*-to-One *Mux

Figure 6.23 shows a 4-to-1 mux module realized through repeated similar assignments. It multiplexes one out of four 4-bit-wide buses to the output side. The assignments are done through 4-bit-wide switches. (The mux can be built up in other ways too; for example, it can be built around the compact assignment statement in Figure 6.20.) The synthesized version of the mux is shown in Figure 6.24; it is essentially the vector counterpart of the 4-to-1 mux of Figure 4.40.

```
module mux_df1(ao, a1, a2, a3, a4, f, en);
//f is a 2 bit selector input & en is an enable input
output [3:0] ao;
input[3:0] a1, a2, a3, a4;
input en;
input [1:0]f;
trireg [3:0]aa0;
parameter d=4'hz;
assign aa0=(f==2'b00)?a1:d;
assign aa0=(f==2'b01)?a2:d;
assign aa0=(f==2'b10)?a3:d;
assign aa0=(f==2'b11)?a4:d;
assign ao =(en)?aa0:d;
endmodule
```

Figure 6.23 A 4 to 1 vector multiplexor module at the data flow level.

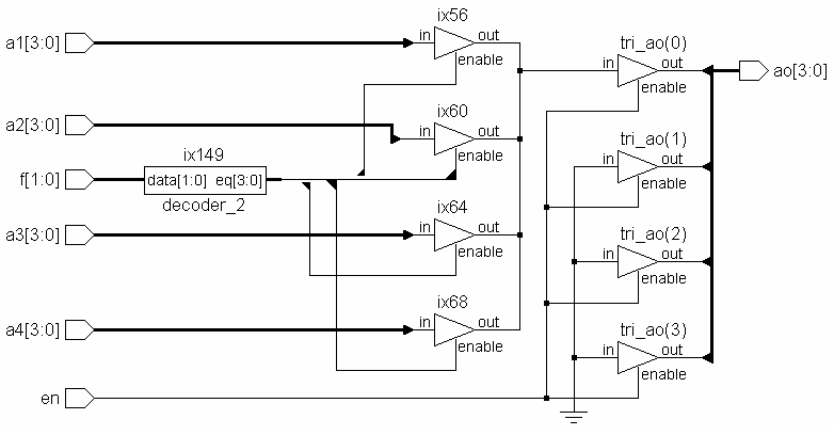

Figure 6.24 Synthesized circuit of the mux in Figure 6.21.

Example 6.4 BCD Adder

A BCD adder can be formed through a compact assignment using a ternary operator. The assignment statement has the form

assign {co, sumd} = (sumb<=4'b1001)?{1'b0,sumb} : (sumb + 4'b0110;

The adder module using the above assignment and a test-bench for the same are shown in Figure 6.25. The synthesized version of the circuit is shown in Figure 6.26. The results of running the test bench are given in Figure 6.27.

```
module bcd(co,sumd,a,b);
input [3:0]a,b;
output [3:0]sumd;
output co;
wire [3:0]sumb;
assign sumb = a + b;
assign{co,sumd}=(sumb<=4'b1001)?{1'b0,sumb}:(sumb+4'b01
10);
endmodule

module tst_bcd;//Test bench
reg [3:0]a,b;
wire co;
wire [3:0]sumd;
bcd bcc(co,sumd,a,b);
initial
    begin
        a = 4'h0 ; b = 4'h0;
    #2  a = 4'h1 ; b = 4'h0;
    #2  a = 4'h2 ; b = 4'h1;
    #2  a = 4'h4 ; b = 4'h5;
    #2  a = 4'h6 ; b = 4'h6;
    #2  a = 4'hd ; b = 4'h1;
    #2  a = 4'hf ; b = 4'h0;
    end
initial $monitor($time,"a = %b, b = %b, co = %b, sumd =
%b",a,b,co,sumd);
initial #16 $stop;
endmodule
```

Figure 6.25 A BCD adder module at the data flow level.

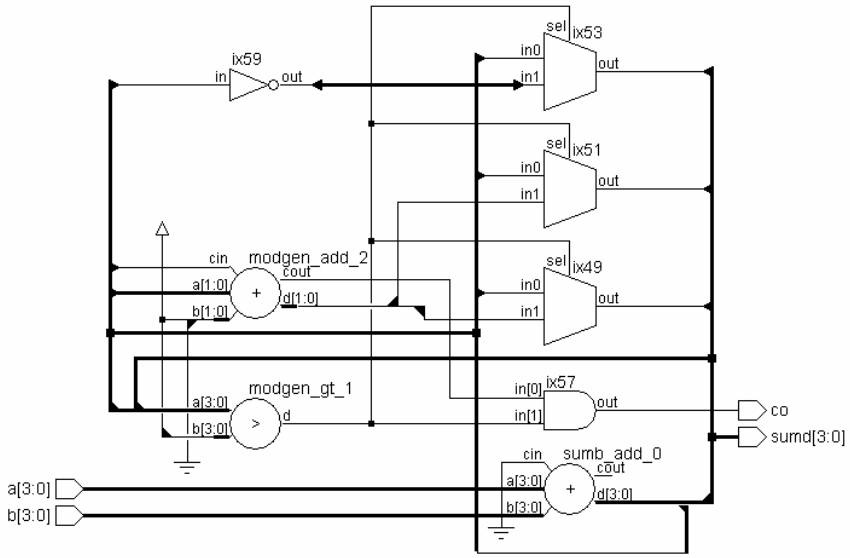

Figure 6.26 Synthesized circuit of the BCD adder.

```
# 0 a = 0000 , b = 0000 ,  co = 0 , sumd = 0000
# 2 a = 0001 , b = 0000 ,  co = 0 , sumd = 0001
# 4 a = 0010 , b = 0001 ,  co = 0 , sumd = 0011
# 6 a = 0100 , b = 0101 ,  co = 0 , sumd = 1001
# 8 a = 0110 , b = 0110 ,  co = 1 , sumd = 0010
#10 a = 1101 , b = 0001 ,  co = 1 , sumd = 0100
#12 a = 1111 , b = 0000 ,  co = 1 , sumd = 0101
```

Figure 6.27 Results of running the test bench for the BCD adder in Figure 6.24.

6.5.4 Operator Priority

A clear understanding of the operator precedence makes room for a compact design description. But it may lead to ambiguity and to inadvertent errors. Whenever one is not sure of the operator priorities, it is better to resort to the use of parentheses and ensure clarity and accuracy of expressions. Further, some synthesizers may not interpret the operator precedence properly. These too call for the apt use of parentheses.

The operators are arranged in tabular form and shown in Table 6.10. The table brings out the order of precedence. The order of precedence decides the priority for sequence of execution and circuit realization in any assignment statement. The following form the basic rules for the same:

Table 6.10 Operator precedence details

Unary operators	! & ~& | ~| ^ ~^ + –	Highest precedence
Binary operator	* ? /	
	+ -	
	<< >>	
	< <= > >=	
	== != === !==	
	& ^ ~^	
	|	
	&&	
	||	
Ternary operators	? :	Lowest precedence

- Unary operators have the highest priority and execute first.
- Subsequently the binary operators execute. Amongst these the algebraic operators have the highest precedence. Amongst the algebraic operators *, / and % have precedence over + and – operators.
- Subsequent precedence amongst the binary operators is as shown in the table.
- Conditional operator has the lowest precedence and hence is executed last.
- In any expression, operators associate from left to right. Ternary operator is the only exception to this; it associates from right to left.

6.5.4.1 Examples

```
P = Q – R + S;
```

Here R is subtracted from Q and then S is added to the result. However, operator precedence does not cause any ambiguity or change the result here.

```
P = Q – R / S;
```

In the above case the “divide” operator “/” has precedence over the “subtract” operator “–”. Hence R will be divided by S, and the result will be subtracted from Q. If division of (Q – R) is desired, the expression has to be recast as

```
P = (Q – R)  / S;
```

In a lengthier expression such as

```
P = a1 – a2 / a3 + a4 * a5;
```

the operation is equivalent to

```
P = a1 – ( a2 / a3 ) + ( a4 * a5 );
```

Use of parentheses adds to clarity especially in operations involving more than two operators. The operation

```
P > Q – R
```

is the same as

```
P > (Q – R)
```

since the relational operator ">" has a lower precedence than the algebraic operator "–". Similarly, the expression

```
P + Q <= R
```

is the same as

```
( P + Q ) <= R.
```

6.5.5 Bit Widths of Expressions

When expressions are evaluated or continuous assignments are made, the bit width of the result is decided by different factors. Three cases arise here:

- The operators decide the bit width of the result; logical operators like '&&' and "||" are examples.
- Widths of all operands are specified and they are consistent in all the expressions used. Bit-wise logic with all the operands having the same width are examples of this.
- Widths of all operands are not specified or do not match. The result of expression evaluation and assignments can lead to ambiguity here. However, the rules to resolve these lead mostly to a natural solution.

Bit widths of results of evaluating expressions are given in Table 6.11 for various cases.

6.6 ADDITIONAL EXAMPLES

The use of operands and their combinations are illustrated through a set of two examples here. They also illustrate how data flow level statements can be combined with instantiation of primitives in defining the modules. The results of running the test-benches are shown as waveforms of selected signals. **`$monitor`** or **`$display`** commands are not inserted in the test benches.

Table 6.11 Bit widths of expressions: A, B and C represent operands in the table; *opr* represents an operator

Expression	Bit width
Integer, unsized constant number	Compiler-specific
Sized constant number	Decided by the specified size
Opr A where opr is an unary operator out of +, - or ~	Same as that of A
Opr A where opr is an unary operator of Table 6.1	1
A opr B where opr is a logical operator of Table 6.4, a relational operator of Table 6.5 or an equality operator of Table 6.6	
A opr B where opr is an algebraic operator from Table 6.2 or a bit-wise logical operator from Table 6.7.	Width of A or B, whichever is higher
A opr B where opr is a shift operator from Table 6.8	Same as that of A
C ? A : B	Width of A or B, whichever is higher
{A, . . , B}	The sum of the bit widths of all the operands
{N*{A, . . . , B}}	N times the sum of the bit widths of all the operands

Example 6.5 Bus Switcher

Figure 6.28 shows the module of a 4-bit bus switcher. A is a 4-bit input bus that is switched on to a selected 4-bit bus. The selection is done by a 2-bit select vector and carried out through a set of simple ternary operator-based assignments. The synthesized circuit of the switcher is shown in Figure 6.29. It decodes the 2-bit select vector into 4 lines that form the control lines for switching. The switching is done through a 4 × 4 tri-state buffer bank. The bus switcher can be easily scaled up to form switches of 8- or 16-bit widths.

```
module demux_df1(a1, a2, a3, a4, a, f);
//A 1 to 4 demux module at data flow level:
output[3:0]a1, a2, a3, a4; // output vectors
input[3:0]a; //a 4 bit input vector
input[1:0]f; //f is the select vector
parameter d = 4'hz;
```

continued

continued

```
assign a1=(f==2'b00)?a:d;
assign a2=(f==2'b01)?a:d;
assign a3=(f==2'b10)?a:d;
assign a4=(f==2'b11)?a:d;
endmodule
```

Figure 6.28 A 4-bit switcher module at the data flow level.

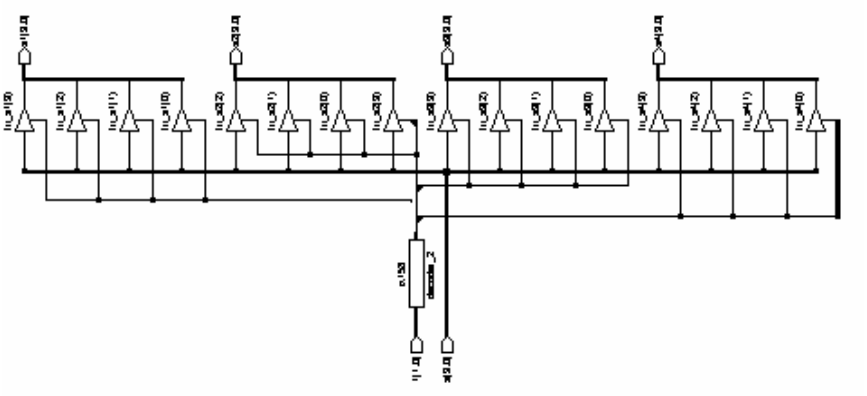

Figure 6.29 Synthesized circuit of the 4-bit switcher.

Example 6.6 Ring Counter

A ring counter is built here in a step-by-step manner. Firstly the simple latch of Example 5.1 has been modified to form another latch shown in Figure 6.30. It has two sets of inputs – sb, rb; and d, db – in place of the single set of sb and rb in Example 5.1. The synthesized circuit is shown in Figure 6.31. The basic cell in the design library being a 2-input AND gate, the NAND function is realized with 2 AND gates followed by a NOT gate. With the additional set of inputs here – d and db – set and reset operations can be carried out independently of the data input.

```
module srff7474(sb, d, rb, db, q, qb);
input sb, rb, d, db;
output q, qb;
nand(q,  sb, d ,qb);
nand(qb, rb, db, q);
endmodule
```

Figure 6.30 A basic latch module.

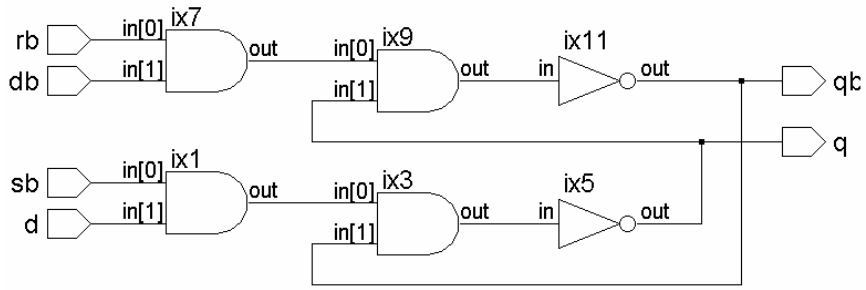

Figure 6.31 Synthesized circuit of the basic latch in Figure 6.30.

A positive edge-triggered flip-flop of the 7474 type is formed by repeated instantiation of the latch in the module of Figure 6.30. Such a flip-flop module is shown in Figure 6.32; it is an enhanced version of the edge-triggered flip-flop in Example 5.5 and in Figure 5.20. The synthesized circuit is shown in Figure 6.33. The srff7474 instantiations are represented there as black boxes.

Figure 6.34 shows a module, which has 4 instantiations of the above edge-triggered flip-flop. This cluster of 4 flip-flops can form the "seed module" of a wide variety of sequential circuits. Figure 6.35 shows the corresponding synthesized circuit.

```
module dff7474new(cp,d,sd,rd,q,qb);
input d,cp,sd,rd;
output q,qb;
wire sdd,rdd;
not(sdd,sd);
not (rdd,rd);
wire n1,n2,n1b,n2b;
 srff7474 ff1(sdd,n2b,rdd,cp,n1,n1b);
 srff7474 ff2(n1b,cp,rdd,d,n2,n2b);
 srff7474 ff3(sdd,n1b,rdd,n2,q,qb);
endmodule
```

Figure 6.32 An edge-triggered flip-flop built with the latch in Figure 6.30 and a test bench for the same.

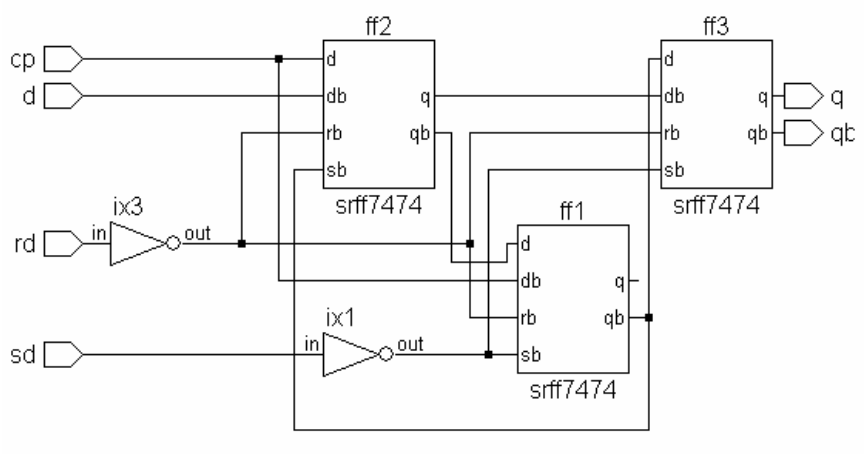

Figure 6.33 Synthesized circuit of the edge-triggered flip-flop in Figure 6.32.

The 4 flip-flops in Figure 6.34 and Figure 6.35 have been connected to form a simple 4-bit ring counter in Figure 6.36. Cen is the overall enabling signal for the ring counter connection. The connection is defined through a set of direct continuous assignments. A test-bench for the ring counter is also included in Figure 6.36. Initially the binary number 1000 is loaded into the set of the 4 flip-flops. Subsequently the flip-flops are connected in a ring counter fashion by enabling Cen. At every positive edge of the clock the data in the ring counter is shifted right by one bit and it circulates. Waveforms of the 4 flip-flops of the ring counter obtained when running the test bench are shown in Figure 6.37. The synthesized circuit of the ring counter is shown in Figure 6.38.

```
module unishrg(clk,d,sd,rd,q,qb);
input clk;
input[3:0]d,sd,rd;
output[3:0]q,qb;
dff7474new ff1(clk,d[0],sd[0],rd[0],q[0],qb[0]);
dff7474new ff2(clk,d[1],sd[1],rd[1],q[1],qb[1]);
dff7474new ff3(clk,d[2],sd[2],rd[2],q[2],qb[2]);
dff7474new ff4(clk,d[3],sd[3],rd[3],q[3],qb[3]);
endmodule
```

Figure 6.34 A module for a general set of 4 edge-triggered flip-flops.

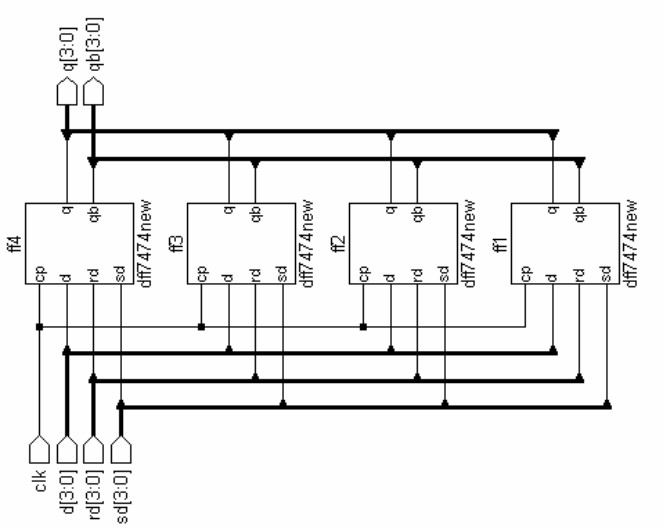

Figure 6.35 Synthesized circuit of the module for a general set of 4 edge-triggered flip-flops in Figure 6.34.

```
module rng_ctr(cen,clk,sd,rd,q,qb);
input clk,cen;
input[3:0]sd,rd;
output [3:0]q,qb;
wire [3:0]d;
unishrg uu(clk,d,sd,rd,q,qb);
assign d[1]=(cen)? q[0]:1'b0;
assign d[2]=(cen)? q[1]:1'b0;
assign d[3]=(cen)? q[2]:1'b0;
assign d[0]=(cen)? q[3]:1'b0;
endmodule

module tst_rng_ctr;//test-bench
reg clk,cen;
reg[3:0]sd,rd;
wire [3:0]q,qb;
rng_ctr rsh(cen,clk,sd,rd,q,qb);
initial
begin
      clk=0;sd=4'b1000;rd=4'b0111;
```

continued

continued

```
    #3sd=4'b0000;rd=4'b0000;
    #2cen=1'b1;
end
always
begin
#2clk =~clk;
end
initial #50 $stop;
endmodule
```

Figure 6.36 A module for a ring counter and a test bench for the same.

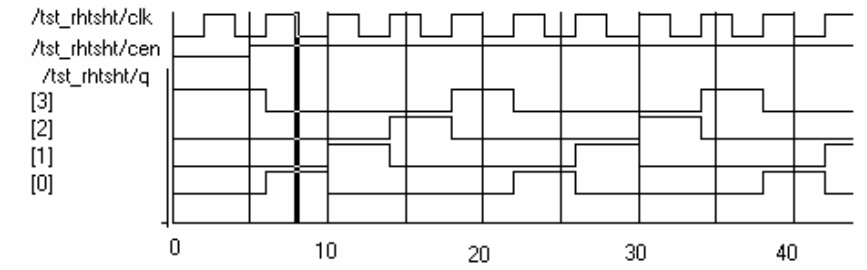

Figure 6.37 Waveforms of a selected set of signals obtained with the test bench in Figure 6.36. The numbers below indicate the time steps

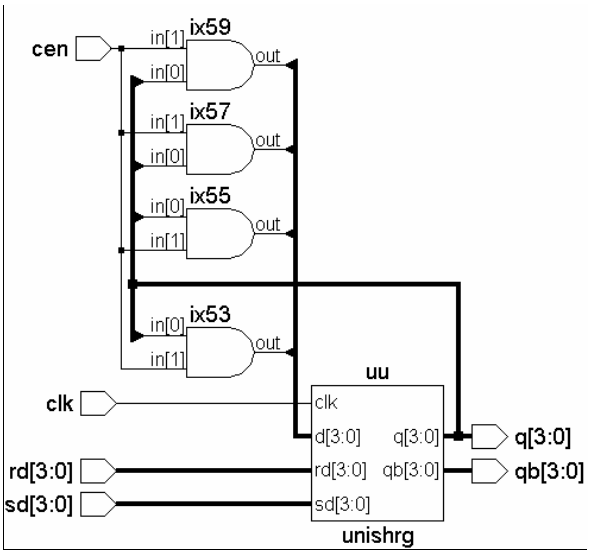

Figure 6.38 Synthesized circuit of the ring counter in Figure 6.36.

6.7 EXERCISES

1. Use continuous assignment statements to design circuits for the following: Byte comparator, Parity generator for one data byte, Binary byte to BCD code, a pair of BCD digits to binary, BCD to Ex-3 code, Ex-3 to BCD, byte multiplier, BCD nibble to 7-segment decoder [Bignel, Sedra, Tocci].
2. What is the result vector in each of the following concatenation operations?
   {3{a},b,c}; {3{a},2{b},c};{3{{a},2{b ,c}}};{3{3{a},2{b}},c};
   {{3{a},b},c};{3{a},b,2{c,1'b0}};};{3{{a, 2'b10, b},2{c,1'b0}}}.
3. Consider the program segment in Figure 6.39; test the segment through a test-bench with values of p and q ranging from 0 to 10. Explain why only r3 is correct. Declare r1, r2, and r3 to be 5 bits wide: Repeat the test run and comment on the results.

```
…..
reg[3:0] p, q, r1, r2, r3;
….
….
….
assign  r1 = p + q;
assign r1 = p + q + 3'b0;
assign r1 = p + q + 0;
…..
….
```

Figure 6.39 Segment of a module for Exercise No. 3 above.

4. Realize the edge-triggered flip-flop of Figure 5.14 through continuous assignments for the gates. Test it through a test bench.
5. Form the NOR gate counterpart of the edge-triggered flip-flop of Figure 5.14; realize it through continuous assignments. Test it through a test bench.
6. Use the set of 4 edge-triggered flip-flops of Figure 6.34 as the basis and form the following. In each case, form a test-bench and test the design.
   - A left-shift-type shift register.
   - An 8-bit shift register of the left shift type.
   - A 4-bit Johnson counter.
   - Have a select line sl. If sl = 1, q[0], q[1], and q[2] are to be connected to the data inputs d[1], d[2], and d[3], respectively and the set of flip-flops should function as a right-shift-type shift register. If sl = 0, q[3], q[2], and q[1] are to be connected to d[2], d[1], and d[0], respectively, to form a left-shift-type shift register.

7. Use the edge-triggered flip-flop of Figure 6.32 as the basis and form (a) a ripple counter of the count up type, (b) a ripple counter of the count-down type, (c) an up down counter. In the case of the up down counter, U_Db is the mode signal. If it is high, the counter will count up. If it is low, the counter will count down.
8. Maximum length sequences (Pseudo-random sequences)[Proakis]: Consider a set of r flip-flops connected in a shift register fashion. D[k] and q[k] represent the data input and output of the kth flip-flop, respectively. The flip-flops are clocked at regular intervals by the clock signal clk. D[1], the data input to the first flip-flop, is formed as the XOR function of a select set of flip-flop outputs; if these are selected suitably, the binary vector representing the outputs of all the flip-flops together takes all possible states in a "pseudo-random" fashion and repeats the sequence cyclically. Specifically, N – the total number of states the sequence passes through – is given by

$$N = 2^r - 1$$

Table 6.12 gives the flip-flop numbers whose outputs are to be XOR'ed to form $d[1]$ to yield the maximum length sequence. Design the Maximum length sequence generator for different values of r. Give the clock input and obtain the output waveform.

Table 6.12 Details for Exercise 8 above

r	2	3	4	5	6	7	8	9	10	11	12
N	3	7	15	31	63	127	255	511	1023	2047	4095
FF numbers to be XOR'ed	2,1	3,1	4,1	5,2	6,2	7,1	8,5,3,1	9,4	10,3	11,1	12,6,4,1

7

BEHAVIORAL MODELING — 1

7.1 INTRODUCTION

Design descriptions at data flow level and gate level are close to the circuit. At every stage of the design description process, one can relate the modules and the instantiations with the corresponding logic or sequential blocks and their interconnections. The approach is practical and effective as long as the gate count remains within a few hundred. An increase in gate count may still be accommodated, if it is due to an increase in vector size –for example, when a system designed and tested at the 8-bit level is being scaled up to a 16- or 32-bit level. But with many of the VLSI's of today, one has to work at a different dimension – the circuit can have a million gates. The increase in vector size may still be accommodated at the data flow level (*e.g.*, 32- or 64-bit systems), since it calls only for scaling of a smaller design. But increase in terms of functional complexity makes the approach almost intractable for many designs.

Behavioral level modeling constitutes design description at an abstract level. One can visualize the circuit in terms of its key modular functions and their behavior; it can be described at a functional level itself instead of getting bogged down with implementation details. The description is carried out essentially with constructs similar to those in "C" language; the design itself is similar to programming in "C" [Gottfried]. For example, one can describe an FFT or a digital filter routine in terms of these constructs. The design can be simulated, debugged, and finalized. This completes the system level structure for the design. Subsequently, one can expand the design by describing the modules in terms of components closer to the data flow and gate level models. One can simulate and debug each such component module, check it for its functionality, integrate it with the main design and test conformity. Constructs for such layered expansion of design are available in behavioral modeling. Proceeding with the layered expansion of design, one can have the final design description at the RTL level itself. However, we may add here that such a top-down activity is more in the realm of design.

The constructs available in behavioral modeling aim at the system level description. Here direct description of the design is not a primary consideration in

Design Through Verilog HDL. T. R. Padmanabhan and B. Bala Tripura Sundari

ISBN: 0-471-44148-1

the Verilog standard. Rather, flexibility and versatility in describing the design are in focus [IEEE]. One should be able to describe the design and simulate it for its functionality. Hence the constructs aim essentially at these two aspects of the design. Synthesis tools available from different vendors can synthesize most of the constructs at the data flow as well as the gate levels, but not all constructs or combinations possible at the behavioral level can be synthesized. The extent to which the constructs at the behavioral level are accommodated in synthesis varies with vendors. The synthesized circuit need not guarantee optimum or near-optimum realization either. These limitations are in line with the basic purpose of behavioral level modeling mentioned above – that is, to complete an error or bug-free description and identify the functional modules required. Their synthesis is more often done following a more detailed design description at the RTL level.

7.2 OPERATIONS AND ASSIGNMENTS

The design description at the behavioral level is done through a sequence of assignments. These are called 'procedural assignments' – in contrast to the continuous assignments at the data flow level. Though it appears similar to the assignments at the data flow level discussed in the last chapter, the two are different. The procedure assignment is characterized by the following:

- The assignment is done through the "=" symbol (or the "<=" symbol) as was the case with the continuous assignment earlier.
- An operation is carried out and the result assigned through the "=" operator to an operand specified on the left side of the "=" sign – for example,
  $N = \sim N$;
  Here the content of **`reg`** *N* is complemented and assigned to the reg *N* itself. The assignment is essentially an updating activity.
- The operation on the right can involve operands and operators. The operands can be of different types – logical variables, numbers – real or integer and so on.
- All the operands are given in Tables 6.1 to 6.9. The format of using them and the rules of precedence remain the same.
- The operands on the right side can be of the net or variable type. They can be scalars or vectors.
- It is necessary to maintain consistency of the operands in the operation expression – *e.g.*,
  $N = m / l$;
  Here *m* and *l* have to be same types of quantities – specifically a **`reg`**, **`integer`**, **`time`**, **`real`**, **`realtime`**, or memory type of data – declared in advance.
- The operand to the left of the "=" operator has to be of the variable (*e.g.*, **`reg`**) type. It has to be specifically declared accordingly. It can be a scalar, a vector, a part vector, or a concatenated vector.

- Procedural assignments are very much like sequential statements in C. Normally they are carried out one at a time sequentially. As soon as a specified operation on the right is carried out, the result is assigned to the quantity on the left – for example
  $N = m + l$;
  $N1 = N * N$;
  The above form a set of two procedures placed within an **always** block. Generally they are carried out sequentially in the order specified; that is, first m and l are added and the result assigned to N. Then the square of N is assigned to $N1$. Subsequently the following assignment, if any, is carried out. However, there can be exceptions to this which will be discussed later. The sequential nature of the assignments requires the operands on the left of the assignment to be of **reg** (variable) type. The basic sequential nature of assignments here is in direct contrast to the concurrent nature of assignments at the data flow level.

Procedural assignments within a process are of a variety of types. These are discussed later.

7.3 FUNCTIONAL BIFURCATION

Design description at the behavioral level is done in terms of procedures of two types; one involves functional description and interlinks of functional units. It is carried out through a series of blocks under an "**always**" banner – discussed later. The second concerns simulation – its starting point, steering the simulation flow, observing the process variables, and stopping of the simulation process; all these can be carried out under the "**always**" banner, an "**initial**" banner, or their combinations. However, each **always** and each **initial** block initiates an activity flow during simulation. In general the activity with all such blocks starts at the simulation time and flows concurrently during the whole simulation process. The concurrent flow of activity with all processes is characteristic of any behavioral level module. A procedure-block of either type – **initial** or **always** – can have a structure shown in Figure 7.1. A block starts with the declaration of the type of block – that is, **initial** or **always**. It may be followed by the definition of a triggering activity and then the body of the block. The body may be a single procedural assignment or a group of procedural assignments. In the latter case the block appears within a "**begin – end**" or similar blocks. The **initial** and **always** blocks have distinct characteristics. The two are discussed separately.

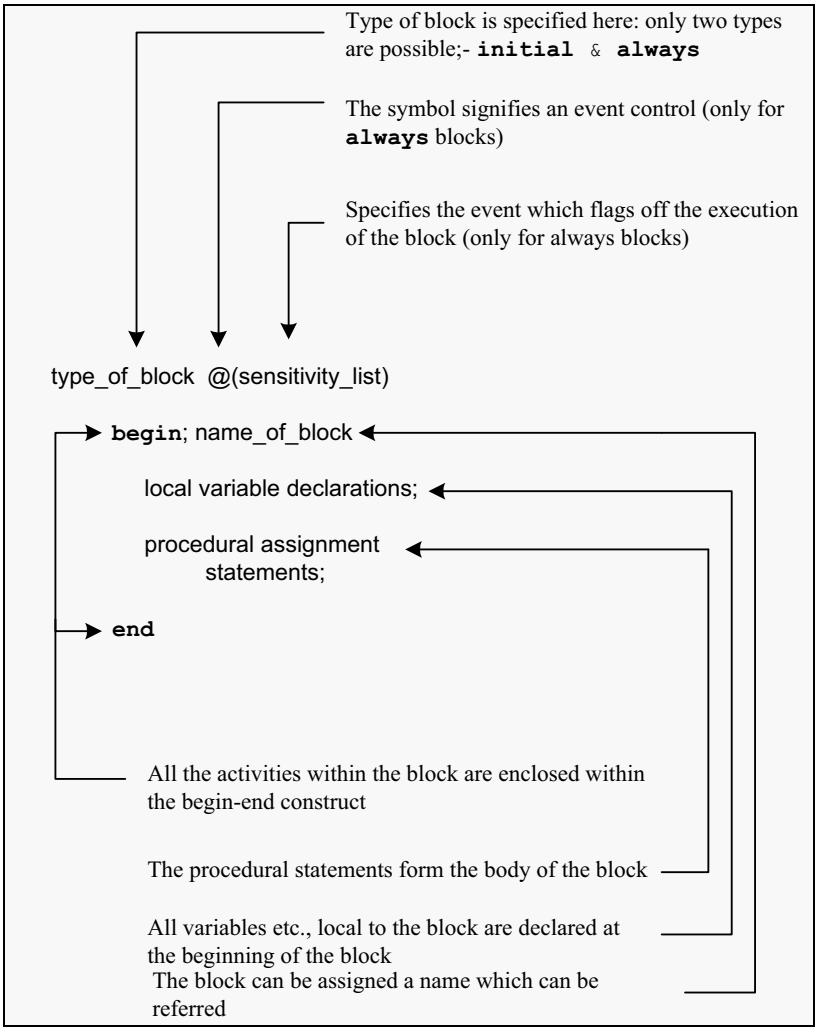

Figure 7.1 Structure of a typical procedural block.

7.3.1 `begin` – `end` Construct

If a procedural block has only one assignment to be carried out, it can be specified as below:

```
initial #2 a=0;
```

The above statement assigns the value 0 to variable a at the simulation time of 2 ns. It is possibly the simplest initial block. More often more than one procedural assignment is to be carried out in an **`initial`** block. All such assignments are grouped together between "**`begin`**" and "**`end`**" declarations. Functionally, the construct is similar to the **`begin`**–**`end`** construct in Pascal or the { } construct in C language. The following are to be noted here:

- Every **`begin`** declaration must have its associated **`end`** declaration.
- **`begin`** – **`end`** constructs can be nested as many times as desired.
- For clarity in description and to avoid mistakes, nested **`begin`** – **`end`** blocks are separated suitably (see Figure 7.2).

7.3.2 Name of the Block

Any block can be assigned a name, but it is not mandatory. Only the blocks which are to be identified and referred by the simulator need be named. Needless to say the names assigned to different blocks have to be different. Names chosen should conform to the rules for the selection of names to variables [see Section 3.4]. Assigning names to blocks serves different purposes:

- Registers declared within a block are local to it and are not available outside. However, during simulation they can be accessed for simulation, *etc*., by proper dereferencing [see Section 11.4].
- Named blocks can be disabled selectively when desired [see Section 8.6].

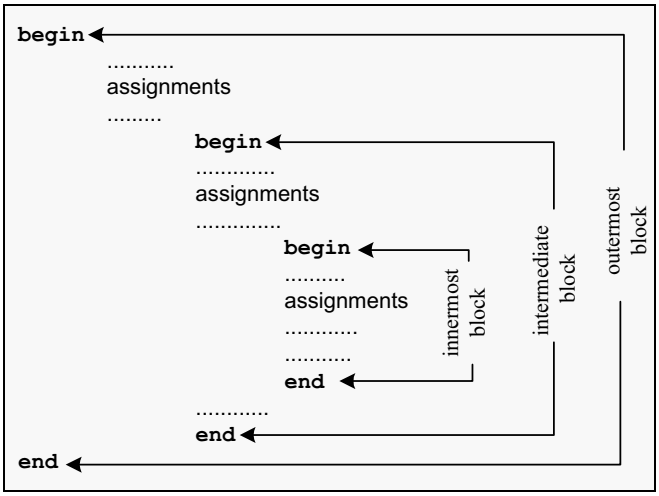

Figure 7.2 Nesting of **`begin`**-**`end`** blocks.

7.3.3 Local Variables

Variables used exclusively within a block can be declared within it. Such a variable need not be declared outside, in the module encompassing the block. Such local declarations conserve memory and offer other benefits too. Regs declared and used within a block are static by nature. They retain their values at the time of leaving the block. The values are modified only at the next entry to the block.

7.4 INITIAL CONSTRUCT

A set of procedural assignments within an **initial** construct are executed only once – and, that too, at the times specified for the respective assignments. Consider the **initial** process shown in Figure 7.3. It is characterized by the following:

- In any assignment statement the left-hand side has to be a storage type of element (and not a net). It can be a **reg**, **integer**, or **real** type of variable. The right-hand side can be a storage type of variable (**reg**, **integer**, or **real** type of variable) or a net.
- As already mentioned in Section 7.2, all the operations described in Tables 6.1 to 6.9 for continuous assignment can be used for procedural assignments as well. The context decides whether the assignment is of a continuous type or procedural type. In the latter case it is present within an **always** or an **initial** construct.
- All the procedural assignments appear within a **begin**–**end** block explained earlier.
- All the procedural assignments are executed sequentially – in the same order as they appear in the design description. The waveforms of a and b conforming to the assignments in the block are shown in Figure 7.4.
- Initially (at time t = 0 ns), a and b are set equal to zero.

```
reg a,b;
initial
      begin
            a = 1'b0;
            b = 1'b0;
       #2   a = 1'b1;
       #3   b = 1'b1;
       #1   a = 1'b0;
       #100$stop;
       end
```

Figure 7.3 A typical initial block.

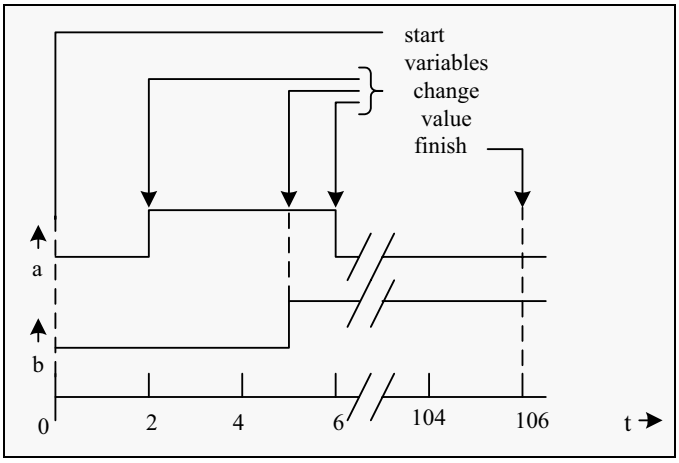

Figure 7.4 Nature of variation of a and b with time in the module of Figure 7.3.

- At time 2 ns a is made equal to 1. After 3 more nanoseconds – that is, at the 5th ns – b is made equal to 1.
- After one more ns – that is, at the 6th ns – a is made equal to 0.
- $**stop** is a system task. 100 ns later – that is, at the 106th ns – the simulation comes to an end (see Figure 7.4).

Integer values have been used here to decide time delay values. In a more general case the delay value can be a constant expression. It is evaluated and decided dynamically as the simulation proceeds.

The **initial** block above does three controlling activities during the simulation run.

- Initialize the selected set of **reg**'s at the start.
- Change values of **reg**'s at predetermined instances of time. These form the inputs to the module(s) under test and test it for a desired test sequence.
- Stop simulation at the specified time.

Figure 7.4 depicts the events for the above case; *t* is the time axis here.

Specific system tasks available in Verilog can be used to tabulate the values of selected variables. Providing such output display in a desired or preferred format is the activity of the simulation run. Two system tasks are useful here – $**display** & $**monitor** [see Section 3.15 and Chapter 11]. By way of illustration consider the simulation routine in Figure 7.5. It incorporates the block

```
module nil;
reg a, b;
initial
begin
        a = 1'b0;
        b = 1'b0;
        $display ("display:  a = %b,  b = %b", a, b);
  #2    a = 1'b1;
  #3    b = 1'b1;
  #1    a = 1'b0;
  #100  $stop;
end
initial
$monitor("monitor:  a = %b,  b = %b", a, b);
endmodule
```

Figure 7.5 A typical module with an **`initial`** block.

Figure 7.3 and two system tasks. The result of the simulation is shown in Figure 7.6. The $**`display`** task is a one-time activity. It is executed when encountered. At that instant in simulation the values of *a* and *b* are zero and the same are displayed. In contrast, $**`monitor`** is a repeated activity. It need be present only once in a simulation routine – all the specified variables will be monitored. If multiple $**`monitor`** tasks are present in the routine, only the last one will be active. All others will be ignored. In contrast, the $**`display`** task may appear any number of times in a module. It is executed every time it is encountered.

Simulators have the facility to observe the waveforms and changes in the magnitudes of different variables with simulation time. The necessary facility is provided with the help of user-friendly menus and icons. Waveforms of *a* and *b* obtained with the test bench of Figure 7.5 are shown in Figure 7.7; they can be seen to be consistent with their values shown in Figure 7.6.

```
output
      # display : a = 0 ,b = 0
      # monitor : a = 0 ,b = 0
      # monitor : a = 1 ,b = 0
      # monitor : a = 1 ,b = 1
      # monitor : a = 0 ,b = 1
```

Figure 7.6 Results of running the test bench in Figure 7.5.

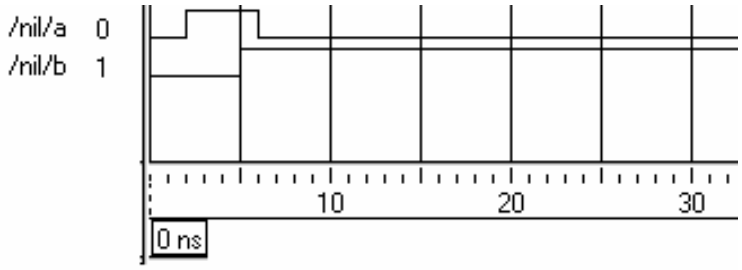

Figure 7.7 Results of running the test bench in Figure 7.5 shown as waveforms.

7.4.1 Multiple `Initial` Blocks

A module can have as many **`initial`** blocks as desired. All of them are activated at the start of simulation. The time delays specified in one **`initial`** block are exclusive of those in any other block. Consider the module in Figure 7.8 which is a modified version of that in Figure 7.5. It has four **`initial`** blocks. The $**`monitor`** task is declared separately (a healthy practice). The simulated results are shown in Figure 7.9. The following observations are in order here:

```
module nil1;
initial
reg a, b;
begin
            a = 1'b0;
            b = 1'b0;
            $display ($time,"display:  a = %b,  b = %b", a, b);
  #2        a = 1'b1;
  #3        b = 1'b1;
  #1        a = 1'b0;
end
initial  #100$stop;
initial $monitor ($time, “monitor:  a = %b,  b = %b", a, b);
initial
            begin
   #2       b = 1'b1;
            end
endmodule
```

Figure 7.8 A typical module with multiple initial blocks.

```
output
      # display : a = 0 , b = 0
      # monitor : a = 0 , b = 0
      # monitor : a = 0 , b = 1
      # monitor : a = 1 , b = 1
      # monitor : a = 1 , b = 0
      # monitor : a = 1 , b = 1
      # monitor : a = 0 , b = 1
```

Figure 7.9 Results of running the test bench in Figure 7.8.

- All changes in *a* are brought about in one initial block.
- Changes to *b* are specified in two blocks, and both these blocks are executed concurrently.
- The progress of simulation time in different blocks is concurrent. However, those in one block are sequential. Changes in *b* are consistent with this.
- The $**stop** task is in an independent **initial** block. Hence simulation is terminated at 100 ns. Contrast this with the previous case (Figure 7.4), where sequential execution results in finish of simulation after 106 ns (even though in both the cases the statement "#100 $**stop**" remains the same).
- More than one activity may be scheduled for execution at one time instant. Those in one **initial** block are executed in the same order as they appear – that is, sequentially.
  Thus, the two events
  a = 1'b0;
  b = 1'b0;
  are executed in the same sequential order – that is, *b* is set to 0 after a is set to 0, although both the activities are scheduled for execution at the same time.
- At 2 ns *a* changes to 1 and *b* changes to 0. These two activities are to be done concurrently. They are in different **initial** blocks. The order of their execution depends upon the implementation. This does not cause any anomaly in the present case. But it can be a potential source of problem in more involved designs and their simulation.

7.5 ALWAYS CONSTRUCT

The **always** process signifies activities to be executed on an "always basis." Its essential characteristics are:

- Any behavioral level design description is done using an always block.
- The process has to be flagged off by an event or a change in a net or a reg. Otherwise it ends in a stalemate.

- The process can have one assignment statement or multiple assignment statements. In the latter case all the assignments are grouped together within a "**begin** – **end**" construct.
- Normally the statements are executed sequentially in the order they appear.

7.5.1 Event Control

The **always** block is executed repeatedly and endlessly. It is necessary to specify a condition or a set of conditions, which will steer the system to the execution of the block. Alternately such a flagging-off can be done by specifying an event preceded by the symbol "**@**". The event can be a change in the variable specified in either direction or a change in a specified direction. For example,

- @(**negedge** clk) :
  executes the following block at the negative edge of the **reg** (variable) clk.
- @(**posedge** clk) :
  executes the following block at the positive edge of the **reg** (variable) clk.
- @clk :
  executes the following block at both the edges of clk.

The event can be a combination as well.

- @(prt or clr) :
  With the above event the block is executed whenever either of the variables prt or clr undergoes a change.
- @(**posedge** clk1 or **negedge** clk2) :
  With the above event the block is executed in two cases – whenever the clock clk1 changes from 0 to 1 state or the clock clk2 changes from 1 to 0. One can specify more elaborate events by OR'ing individual ones. The following are to be noted:
- The events can be changes in **reg**, **integer**, **real** or a signal on a net. These should be declared beforehand.
- No algebra or logic operation is permitted as an event. The OR'ing signifies "execute the block if any one of the events takes place."
- The edge transition on each event is to be specified separately
- Note the difference between the following:
  - (**posedge** clk1 or clk2): means "execute the block following if clk1 goes to 1 state or clk2 changes state (whether 0 to 1 or 1 to 0)."
  - (**posedge** clk1 or **posedge** clk2): means "execute the block following if clk1 goes to 1 state or clk2 goes to 1 state."

- The positive transition for a reg type single bit variable is a change from 0 to1. For a logic variable it is a transition from false to true.
- The "**posedge**" transition for a signal on a net can be of three different types:
  - 0 to1
  - 0 to **x** or **z**
  - **x** or **z** to 1
- The "**negedge**" transition for a signal on a net can be of three different types:-
  - 1 to 0
  - 1 to **x** or **z**
  - **x** or **z** to 0
- If the event specified is in terms of a multibit **reg**, only its least significant bit is considered for the transition. Changes in the other bits are ignored.
- The event-based flagging-off of a block is applicable only to the **always** block.
- According to the recent version of the LRM, the comma operator (,) plays the same role as the keyword **or**. The two can be used interchangeably or in a mixed form. Thus the following are identical:

```
@ (a or b or c)
@ (a or b, c)
@ (a, b, c)
@ (a, b or c)
```

7.6 EXAMPLES

A few simple design examples are considered here [Arnold, Bogart, Navabi]]; they are aimed at bringing out the potential flexibility at the behavioral level, despite the compactness in the module descriptions. Some of these examples have already been discussed in earlier chapters at the data flow as well as the gate levels.

Example 7.1 A Versatile Counter

We consider a versatile up-down counter module with the following facilities:

- *Clear input*: If it goes high, the counter is cleared and reset to zero.
- *U/D input*: If it goes high, the counter counts up; if it goes down, the counter counts down.
- The counter counts at the negative edge of the clock.
- The counter counts up or down between 0 and *N* where *N* is any 4-bit hex number.

The above counter design specifications are implemented in stages. The module in Figure 7.10 is an up counter which counts up repeatedly from 0 to a preset number *N*. A test-bench for the counter is also shown in the figure. *N* is an input to the module. The count advances at every negative edge of the clock. When the count reaches the value *N*, the count value *a* is reset to 0. The simulation results are shown as waveforms in Figure 7.11 (only partially shown). The periodic clock waveform (with a period of 4 ns), the incrementing of *a* at every negative edge of the clock and counting of *a* from 0 to the set value of *N* (=1011 in this specific

```
module counterup(a,clk,N);
input clk;
input[3:0]N;
output[3:0]a;
reg[3:0]a;
initial a=4'b0000;
always@(negedge clk) a=(a==N)?4'b0000:a+1'b1;
endmodule

module tst_counterup;//TEST_BENCH
reg clk;
reg[3:0]N;
wire[3:0]a;
counterup c1(a,clk,N);
initial
begin
     clk = 0;
     N   = 4'b1011;
end
always #2 clk=~clk;
initial $monitor($time,"a=%b,clk=%b,N=%b",a,clk,N);
endmodule
```

Figure 7.10 An up counter module.

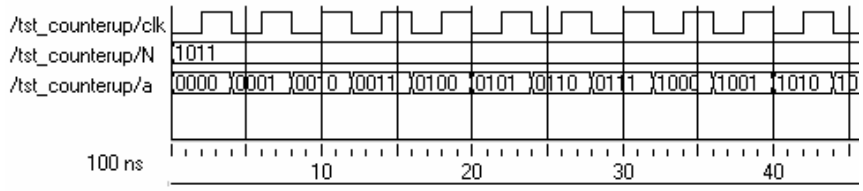

Figure 7.11 Partial results of running the test bench in Figure 7.10.

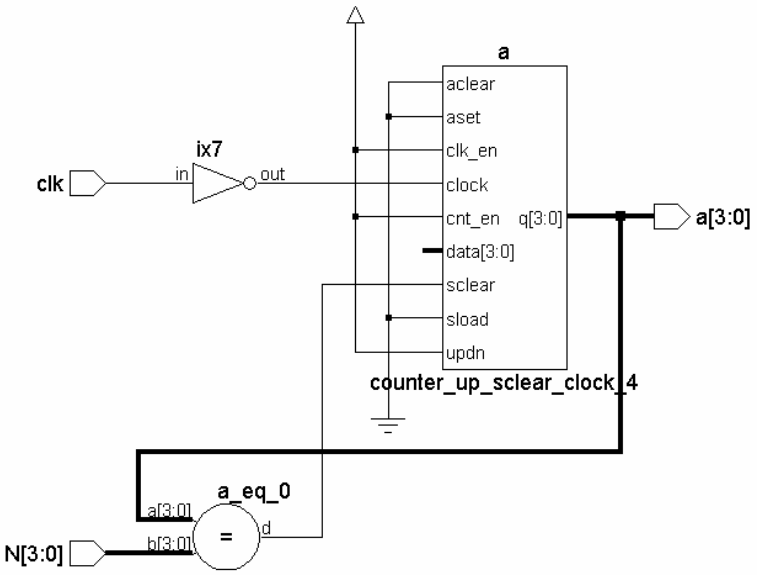

Figure 7.12 Synthesized circuit of the up counter in Figure 7.10.

case) can be seen from the figure. The synthesized circuit of the counter is shown in Figure 7.12. It has a versatile counter block and a comparator. The comparator compares the value of *a* with the set value of *N* and resets the counter when the two are equal – as specified in the design module.

The module of Figure 7.13 is a down counter. The count `a` decrements at the negative edge of the clock – clk. The counter counts down from *N* to zero. As soon as the count reaches the value 0, it is set back to *N*. The simulation results are shown tabulated in Figure 7.14 and as waveforms in Figure 7.15; these can be seen to be consistent with the design module. The synthesized circuit is shown in Figure 7.16. The basic blocks – namely versatile counter, comparator and buffer for the clock – are the same as those for the up counter of Figure 7.12. The comparator output loads the value of *N* back into the counter every time `a` reaches the set value of *N* (In contrast, in the case of the up counter above, the comparator resets the counter back to zero, whenever `a` reaches the set value of *N*.)

```
module counterdn(a,clk,N);
input clk;
input[3:0]N;
output[3:0]a;
reg[3:0]a;
initial a =4'b0000;
```

continued

continued

```
always@(negedge clk) a=(a==4'b0000)?N:a-1'b1;
endmodule

module tst_counterdn();//TEST_BENCH
reg clk;
reg[3:0]N;
wire[3:0]a;
counterdn cc(a,clk,N);
initial
begin
     N   = 4'b1010;
     Clk = 0;
end
always #2 clk=~clk;
initial $monitor($time,"a=%b,clk=%b,N=%b",a,clk,N);
initial #55 $stop;
endmodule
```

Figure 7.13 Design module of a down counter and a test bench for the same.

```
Output
     #  0a=1010,clk=0,N=1010
     #  2a=1010,clk=1,N=1010
     #  4a=1001,clk=0,N=1010
     #  6a=1001,clk=1,N=1010
     #  8a=1000,clk=0,N=1010
     # 10a=1000,clk=1,N=1010
     # 12a=0111,clk=0,N=1010
     # 14a=0111,clk=1,N=1010
     # 16a=0110,clk=0,N=1010
     # 18a=0110,clk=1,N=1010
     # 20a=0101,clk=0,N=1010
     # 22a=0101,clk=1,N=1010
     # 24a=0100,clk=0,N=1010
     # 26a=0100,clk=1,N=1010
     # 28a=0011,clk=0,N=1010
     # 30a=0011,clk=1,N=1010
     # 32a=0010,clk=0,N=1010
     # 34a=0010,clk=1,N=1010
     # 36a=0001,clk=0,N=1010
```

continued

continued

```
# 38a=0001,clk=1,N=1010
# 40a=0000,clk=0,N=1010
# 42a=0000,clk=1,N=1010
# 44a=1010,clk=0,N=1010
# 46a=1010,clk=1,N=1010
# 48a=1001,clk=0,N=1010
# 50a=1001,clk=1,N=1010
# 52a=1000,clk=0,N=1010
# 54a=1000,clk=1,N=1010
```

Figure 7.14 Results of running the test bench in Figure 7.13.

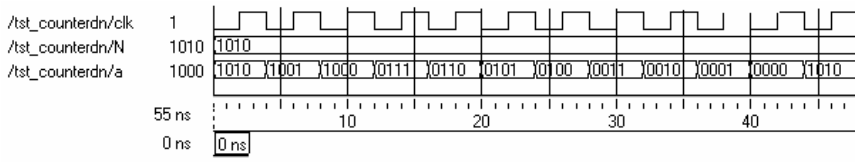

Figure 7.15 Results of running the test bench in Figure 7.13 – shown partly as waveform.

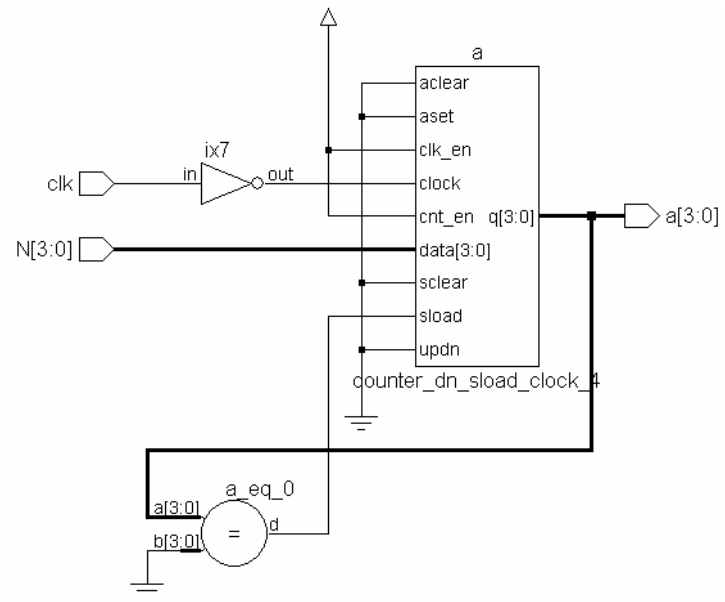

Figure 7.16 Synthesized circuit of the down counter in Figure 7.13.

The up and down modes of counting have been combined in the up down counter of Figure 7.17. A test bench is also shown in the figure. The test results are tabulated in Figure 7.18 and also shown as waveforms in Figure 7.19. Figure 7.20 shows the synthesized circuit; the counter block remains the same as in the last two cases; the mode control part of the circuit has been changed to meet the enhanced needs. The counting can be seen to be changing from "up" to the "down" type, when the mode control input `u_d` changes.

```
module updcounter(a,clk,N,u_d);
input clk,u_d;
input[3:0]N;
output[3:0]a;
reg[3:0]a;
initial a =4'b0000;
always@(negedge clk)
a=(u_d)?((a==N)?4'b0000:a+1'b1):((a==4'b0000)?N:a-
1'b1);
endmodule

module tst_updcounter();//TEST_BENCH
reg clk,u_d;
reg[3:0]N;
wire[3:0]a;
updcounter c2(a,clk,N,u_d);
initial
begin
    N   = 4'b0111;
    u_d = 1'b0;
    clk = 0;
end
always #2 clk=~clk;
always #34u_d=~u_d;
initial $monitor
($time,"clk=%b,N=%b,u_d=%b,a=%b",clk,N,u_d,a);
initial #64 $stop;
endmodule
```

Figure 7.17 Design module of an up down counter and a test bench for the same.

```
#                    0clk=0,N=0111,u_d=0,a=0111
#                    2clk=1,N=0111,u_d=0,a=0111
#                    4clk=0,N=0111,u_d=0,a=0110
#                    6clk=1,N=0111,u_d=0,a=0110
#                    8clk=0,N=0111,u_d=0,a=0101
#                   10clk=1,N=0111,u_d=0,a=0101
#                   12clk=0,N=0111,u_d=0,a=0100
#                   14clk=1,N=0111,u_d=0,a=0100
#                   16clk=0,N=0111,u_d=0,a=0011
#                   18clk=1,N=0111,u_d=0,a=0011
#                   20clk=0,N=0111,u_d=0,a=0010
#                   22clk=1,N=0111,u_d=0,a=0010
#                   24clk=0,N=0111,u_d=0,a=0001
#                   26clk=1,N=0111,u_d=0,a=0001
#                   28clk=0,N=0111,u_d=0,a=0000
#                   30clk=1,N=0111,u_d=0,a=0000
#                   32clk=0,N=0111,u_d=0,a=0111
#                   34clk=1,N=0111,u_d=1,a=0111
#                   36clk=0,N=0111,u_d=1,a=0000
#                   38clk=1,N=0111,u_d=1,a=0000
#                   40clk=0,N=0111,u_d=1,a=0001
#                   42clk=1,N=0111,u_d=1,a=0001
#                   44clk=0,N=0111,u_d=1,a=0010
#                   46clk=1,N=0111,u_d=1,a=0010
#                   48clk=0,N=0111,u_d=1,a=0011
#                   50clk=1,N=0111,u_d=1,a=0011
#                   52clk=0,N=0111,u_d=1,a=0100
#                   54clk=1,N=0111,u_d=1,a=0100
#                   56clk=0,N=0111,u_d=1,a=0101
#                   58clk=1,N=0111,u_d=1,a=0101
#                   60clk=0,N=0111,u_d=1,a=0110
#                   62clk=1,N=0111,u_d=1,a=0110
```

Figure 7.18 Results of running the test bench in Figure 7.17.

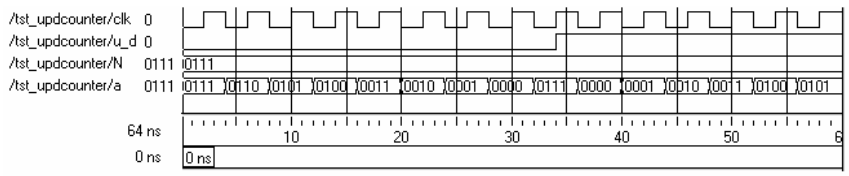

Figure 7.19 Results of running the test bench in Figure 7.17 – shown partly as waveforms.

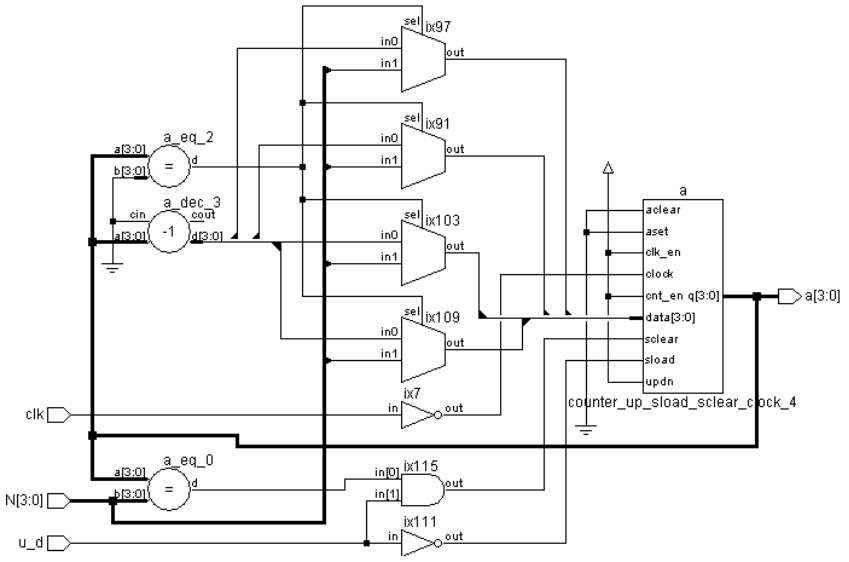

Figure 7.20 Synthesized circuit of the up down counter in Figure 7.17.

The counter as described in Figure 7.21 has an additional "clear" input. With this enhancement, it has become versatile (compare with 74196 or 74197). Note that despite the versatility offered by the design, the full counter has been described in the single line of executable statement reproduced below:

```
always@(negedge clk or posedge clr)
a=(clr)?4'h0:((u_d)?((a==N)?4'b0000:a+1'b1):((a==4'b0000)?N:a-1'b1));
```

```
module clrupdcou(a,clr,clk,N,u d);
input clr,clk,u_d;
input[3:0]N;
output[3:0]a;
reg[3:0]a;
initial a =4'b0000;
always@(negedge clk or posedge clr)
      a=(clr)?4'h0:((u_d)?((a==N)?4'b0000:a+1'b1):((a==
4'b0000)?N:a-1'b1));
 /*signals having priority over clk have to be included
in the sensitivity list*/
endmodule
```

continued

continued

```
module tst_clrupdcou;//TEST_BENCH
reg clr,clk,u_d;
reg[3:0]N;
wire [3:0]a;
clrupdcou cc11(a,clr,clk,N,u_d);
initial
begin
     N   = 4'b0111;
     Clr = 1'b1;u_d=1'b1;
     Clk = 0;
end
always
begin
     #2 clk = ~clk;
        clr = 1'b0;
end
always  #34 u_d<=~u_d;
initial $monitor($time
,"clk=%b,clr=%b,u_d=%b,N=%b,a=%b",clk,clr,u_d,N,a);
initial #60 $stop;
endmodule
```

Figure 7.21 Design module of an up down counter with clear facility and a test bench for the same.

The test bench for the counter is also shown in the figure. The test results are reproduced in Figure 7.22 as waveforms; the synthesized circuit is shown in Figure 7.23.

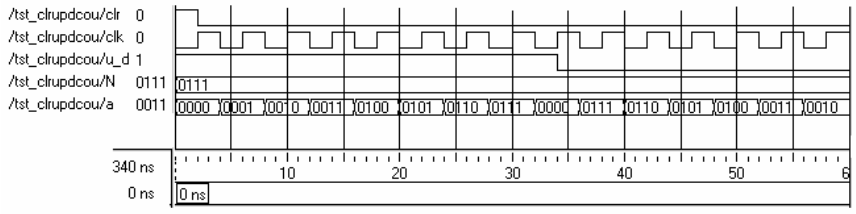

Figure 7.22 Results of running the test bench in Figure 7.21 – shown partly as waveforms.

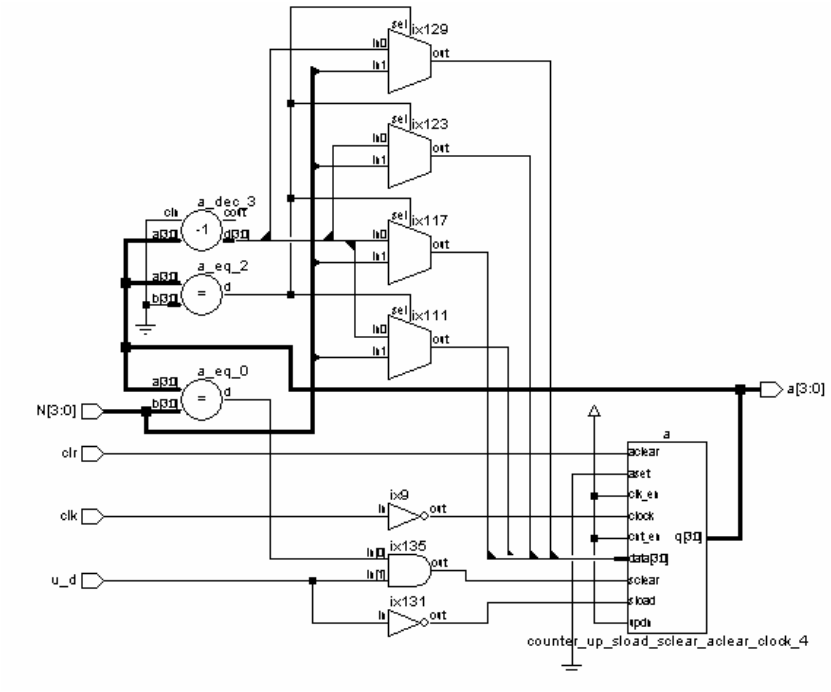

Figure 7.23 Synthesized circuit of the up counter in Figure 7.21.

Example 7.2 Shift Register

Figure 7.24 shows an 8-bit shift register module along with a test bench for the same. The register shifts by one bit to the right if r_l = 1 and to the left by one bit otherwise (*i.e.*, if r_l = 0). The whole shift register is described in a single line of procedural assignment, namely

```
always@(negedge clk) a=(r_l)?(a>>1'b1):(a<<1'b1);
```

The simulation results are given in tabular form in Figure 7.25 and as waveforms in Figure 7.26.

```
module shifrlter(a,clk,r_l);
input clk,r_l;
output [7:0]a;
reg[7:0]a;
initial a= 8'h01;
always@(negedge clk)
```

continued

continued

```
begin
     a=(r_l)?(a>>1'b1):(a<<1'b1);
end
endmodule

module tst_shifrlter;//test-bench
reg clk,r_l;
wire [7:0]a;
shifrlter shrr(a,clk,r_l);
initial
begin
     clk =1'b1;
     r_l = 0;
end
always #2 clk =~clk;
initial #16 r_l =~r_l;
initial
$monitor($time,"clk=%b,r_l = %b,a =%b ",clk,r_l,a);
initial #30 $stop;
endmodule
```

Figure 7.24 Design module of a shift register with facility for right or left shift and a test bench for the same.

```
Output
  #    0 clk=1, r_l = 0 , a = 00000001
  #    2 clk=0, r_l = 0 , a = 00000010
  #    4 clk=1, r_l = 0 , a = 00000010
  #    6 clk=0, r_l = 0 , a = 00000100
  #    8 clk=1, r_l = 0 , a = 00000100
  #   10 clk=0, r_l = 0 , a = 00001000
  #   12 clk=1, r_l = 0 , a = 00001000
  #   14 clk=0, r_l = 0 , a = 00010000
  #   16 clk=1, r_l = 1 , a = 00010000
  #   18 clk=0, r_l = 1 , a = 00001000
  #   20 clk=1, r_l = 1 , a = 00001000
  #   22 clk=0, r_l = 1 , a = 00000100
  #   24 clk=1, r_l = 1 , a = 00000100
  #   26 clk=0, r_l = 1 , a = 00000010
  #   28 clk=1, r_l = 1 , a = 00000010
```

Figure 7.25 Results of running the test bench in Figure 7.24.

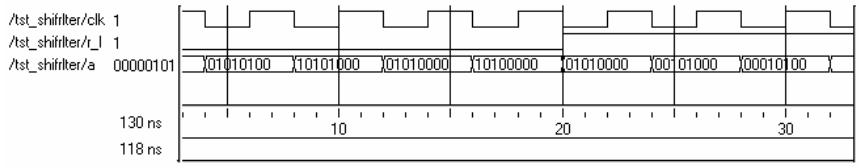

Figure 7.26 Results of running the test bench in Figure 7.24 – shown partly as waveforms.

Example 7.3 Clocked Flip-Flop

The module for a clocked flip-flop is shown in Figure 7.27. A test bench for the flip-flop is also included in the figure. The test results are shown in Figure 7.28 and Figure 7.29 in tabular form and as waveforms, respectively. The input can be seen to be sensed, latched, and presented as output at every negative edge of the clock. Otherwise the output remains frozen at the last latched value. The synthesized circuit of the flip-flop is shown in Figure 7.30.

```
module dff(do,di,clk);
output do;
input di,clk;
reg do;
initial
do=1'b0;
always@(negedge clk) do=di;
endmodule

module tst_dffbeh();//test-bench
reg di,clk;
wire do;
dff d1(do,di,clk);
initial
begin
     clk=0;
     di=1'b0;
end
always #3clk=~clk;
always #5 di=~di;
initial
$monitor($time,"clk=%b,di=%b,do=%b",clk,di,do);
initial #35 $stop;
endmodule
```

Figure 7.27 Design module of a D-flip-flop and a test bench for the same.

```
Output
  #                    0clk=0,di=0,do=0
  #                    3clk=1,di=0,do=0
  #                    5clk=1,di=1,do=0
  #                    6clk=0,di=1,do=1
  #                    9clk=1,di=1,do=1
  #                   10clk=1,di=0,do=1
  #                   12clk=0,di=0,do=0
  #                   15clk=1,di=1,do=0
  #                   18clk=0,di=1,do=1
  #                   20clk=0,di=0,do=1
  #                   21clk=1,di=0,do=1
  #                   24clk=0,di=0,do=0
  #                   25clk=0,di=1,do=0
  #                   27clk=1,di=1,do=0
  #                   30clk=0,di=0,do=0
  #                   33clk=1,di=0,do=0
```

Figure 7.28 Results of running the test bench in Figure 7.27.

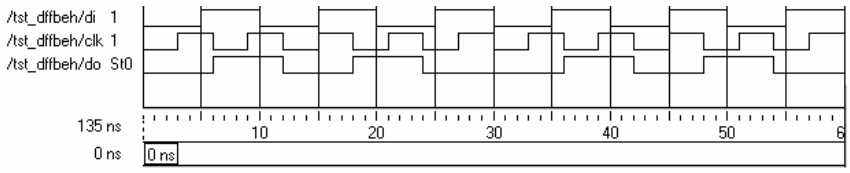

Figure 7.29 Results of running the test bench in Figure 7.27– shown partly as waveforms.

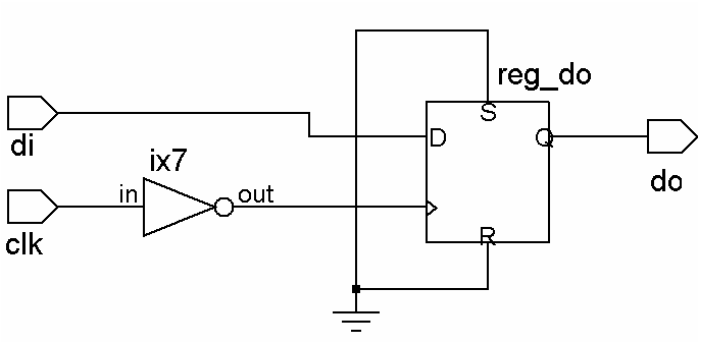

Figure 7.30 Synthesized circuit of the D-flip-flop in Figure 7.27.

Example 7.4 D Latch

Figure 7.31shows the module of a D latch along with its test bench. Whenever en is high, the output follows the input; the latch is transparent. When en goes low the output remains frozen at the last value. The simulation results are shown as waveforms in Figure 7.32.

```
module dffen(do,di,en); // d-latch
output do;
input di,en;
reg do;
initial
do=1'b0;
always@(di or en)
if(en)
do=di;
endmodule

module tst_dffbehen;//test-bench
reg di,en;
wire do;
dffen d1(do,di,en);
initial
begin
     en=0;
     di=1'b0;
end
always#7 en =~en;
always#4 di=~di;
initial
$monitor($time,"en=%b,di=%b,do=%b",en,di,do);
initial #50 $stop;
endmodule
```

Figure 7.31 Design module of a D-latch and a test bench for the same.

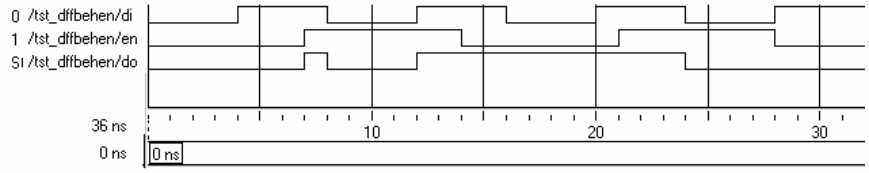

Figure 7.32 Results of running the test bench in Figure 7.31– shown partly as waveforms.

Example 7.5 Clock Waveform

Consider the design description line

```
always #3 clk = ~clk;
```

The sequence of operation taking place within this line segment is as follows:

- When the system comes across the statement, it schedules an activity 3 ns later.
- At the end of the 3 ns, the value of clk is sensed; the sensed value is complemented and then stored temporarily.
- Then the stored value is assigned to the clock, which completes the activity of the always block; once again, execution resumes at step 1.

The clock waveform is shown in Figure 7.33.

7.7 ASSIGNMENTS WITH DELAYS

Specific delays can be associated with procedural assignments. The delay refers to the specific activity it qualifies. A variety of possibilities of specifying delays to assignments exist. A clear understanding makes room for flexibility through their judicious use; the absence of a clear understanding can be disastrous! The variety and flexibility are brought here through simple illustrations.
Consider the assignment

```
always #3 b = a;
```

simulator encounters this at zero time and posts the entire activity to be done 3 ns later. Further, by virtue of the always nature of the activity, the assignment is scheduled to be repeated every 3 ns, irrespective of whether a changes in the meantime. Values of a at the 3rd, 6th, 9th, *etc*., ns are sampled and assigned to b. Figure 7.35 shows the waveforms of a and b with the above assignment and execution of the module in Figure 7.34. Changes in the values of a lasting less than 3 ns may be ignored. Specifically, in this case, a took the value of 1 during the interval 4th ns to the 5th ns which is not passed on to b.

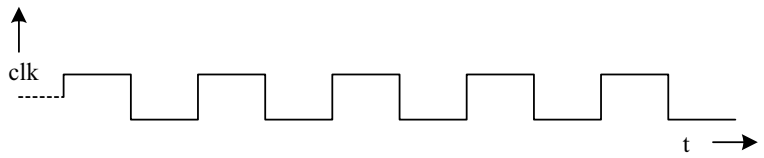

Figure 7.33 The clock waveform with an **always** block of one statement to generate a clock.

```
module del1;
reg a,b;
always  #3 b=a;
Initial

begin
                a = 1'b1;
                b = 1'b0;
          #1    a = 1'b0;
          #3    a = 1'b1;
          #1    a = 1'b0;
          #2    a = 1'b1;
          #3    a = 1'b0;
end
initial $monitor($time, "  a = %d,  b = %d", a, b);
initial #20 $finish;
endmodule
```

Figure 7.34 A module to illustrate delayed assignment.

The module of figure 7.36 is a modified version of that in Figure 7.34. The activities within the always block (of a single statement) are carried out whenever the value of a changes. The sole activity is that of assigning the value of a to b with a delay of 2 ns – that is, 2 ns after a changes sign. The waveform assigned to a as well as the resulting waveform of b is shown in Figure 7.37. If a were to remain invariant, b will have no assignment here. In contrast in the previous case (Figure 7.35), b is given an assignment (=a) at every 3rd ns.

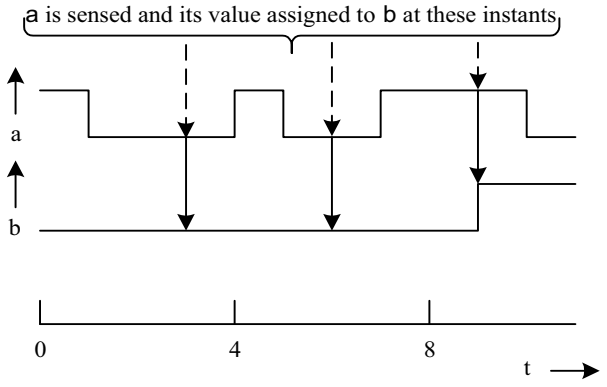

Figure 7.35 Waveforms of a and b with the simulation of the module in Figure 7.34.

```
module del2;
reg a,b;
always @(a) #2 b=a;

Initial
begin
                a = 1'b1;
                b = 1'b0;
        #1      a = 1'b0;
        #3      a = 1'b1;
        #1      a = 1'b0;
        #2      a = 1'b1;
        #3      a = 1'b0;
end
initial $monitor($time, "  a = %d,  b = %d", a, b);
initial #20 $finish;
endmodule
```

Figure 7.36 A modified version of the module in Figure 7.34.

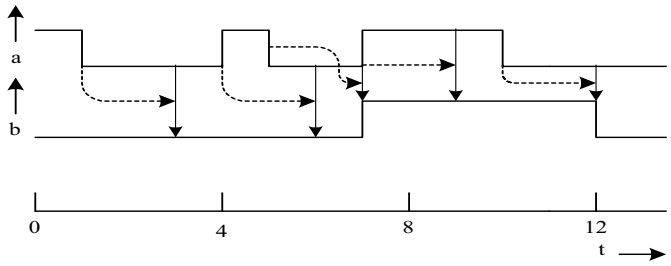

Figure 7.37 Waveforms of a and b obtained with the simulation of the module in Figure 7.36.

Consider a more detailed example – that of Figure 7.38. The **always** block has two assignments. These are carried out sequentially and repeatedly. At the 3rd ns the assignment b = a is executed. The assignment that follows is executed 1 ns later – that is, at the 4th ns. Again 3 ns later – that is, at the 7th ns – the first assignment is executed, and so on. The results obtained are shown in Table 7.1. Only the values of a, b, and c at the first few time step values are shown in the table.

```
module del3;
integer a,b, c;
always
begin
        # 3 b = a;
        # 1 c = a;
       end
initial
begin
                a = 0;
                b = 0;
                c = 0;
            #2  a = 1;
             #2  a = 2;
             #2  a = 3;
             #2  a = 4;
             #2  a = 5;
             #2  a = 6;
end
initial $monitor($time, "  a = %d,  b = %d", a, b);
initial #20 $finish;
endmodule
```

Figure 7. 38 A module where b and c are versions of a with different delays.

7.7.1 Intra-assignment Delays

An assignment delay of the type discussed above, delays execution of the whole assignment by the specified time duration. In contrast, the "intra-assignment" delay carries out the assignment in two parts. An assignment with an intra-assignment has the form

A = # dl expression;

Here the expression is scheduled to be evaluated as soon as it is encountered. However, the result of the evaluation is assigned to the right-hand side quantity a

Table 7.1 Values of variables in the module of Figure 7.38

t	0	1	2	3	4	5	6	7	8	9	10	11	12
a	0	0	1	1	2	2	3	3	4	4	5	5	6
b	0	0	0	1	1	1	1	3	3	3	3	5	5
c	0	0	0	0	2	2	2	2	4	4	4	4	6

after a delay specified by dl. dl can be an integer or a constant expression [see Section 7.7.2]. Consider the example in Figure 7.39. b is assigned the value of a with an intra-assignment delay of 2 ns. The value of a is sensed at zero ns and assigned to b after 2 ns. Until that time, b retains its old value. Again at the 2nd ns, a is sensed and b is assigned the new value of a at the 4th ns, and so on. Partial results of simulation are shown in Table 7.2. The following points are to be noted here:

- The value of a is sensed at time instants 2, 4, 6, *etc*.
- Values at other instants of time are not sensed.
- All assignments are carried out with a delay of 2 ns.
- Changes in a which do not last for 2 ns may be ignored.

```
Module del4;
Integer a, b;
Always b = #2 a;
Initial
begin
          a = 0;      b = 0;   #2    a =1;   #2    a =2;   #2 a =3;
  #2      a =4;   #2 a = 5;   #2    a =6;  #2    a =7;   #2  a =8;
end
initial $monitor($time, "  a = %d,  b = %d", a, b);
initial #20 $finish;
endmodule
```

Figure 7.39 A module to illustrate delayed assignment.

Delays tied to different segments of an assignment have different effects. The subtle differences are brought out through two more examples crafted specifically for the purpose. Consider the module in Figure 7.40. The integer a is assigned the value 0 at 0th ns and the value 1 at 1 ns. Subsequently, it is incremented every 2 ns until the end of simulation. Values are assigned to b, c, and d – declared as integers. These assignments are done with specific delays. The results of the simulation are given in Table 7.3. Changes to b, c, and d and the reasons for the same in each case are explained in the remarks columns of the table. A few observations are in order here:

Table 7.2 Partial output with the simulation of the module in Figure 7.39

t	a	b	Remarks
0	0	0	There are two assignment statements to a at 2 ns intervals – namely the one in the always block and the other one in the initial block; both are concurrent. The simulator decides the precedence. The output here shows that the assignment in the always block has the precedence.
2	1	x	
4	2	1	
6	3	2	
8	4	2	

```
Module del_dem4;
Integer a,b,c,d,n;
Always
begin
            #2      b = a;
                    c = #1 a;
                    d = a;
end
initial
begin
          a = 0; b = 0; c = 0; d = 0;
       #1 a = 1;  #2 a = 2;  #2 a = 3;  #2 a = 4;
       #2 a = 5;  #2 a = 6;  #2 a = 7;  #2 a = 8; #2 a = 9; #2 a = 10;
end
initial $monitor ($time, "  a = %d,  b = %d,  c = %d,  d = %d", a, b, c, d);
endmodule
```

Figure 7.40 A module to illustrate combinations of delays.

- The always block extends for three time steps. Thereafter it is repeated cyclically.
- The assignment statements in the always block are sequential assignment statements.
- Precedence of assignments slotted for a specific time instant, when they are in one block, is clear. However, when they are in different blocks, the compiler decides the precedence. But this does not cause any discrepancy in the present case.

Table 7.3 Output obtained with the simulation of the module in Figure 7.40 (shown rearranged)

Time	a	b	c	d	Remarks
0	0	0	0	0	…
1	1	0	0	0	…
2	1	1	0	0	The value of a at 2nd ns is assigned to b; the same is stored for assignment to c, 1 ns later
3	2	1	1	2	c is assigned the value of a 1 ns earlier; the present value of a is assigned to d.
5	3	3	1	2	All assignments within the always block are done; the assignment sequential is repeated; no change at the 4th ns; at the 5th ns b is assigned the value of a and so on.
6	3	3	3	3	…
7	4	3	3	3	…

Consider the module of Figure 7.41, which is a slight variant of the above in Figure 7.40. The assignments to b and c in the module of Figure 7.40 have been interchanged to form the module here. The simulated results are shown in Table 7.4. The following additional observations are in order here:

- The always block is repeated after every 3 ns – the total assignment time for the sequential.
- At t = 0, a is sampled and the sampled value is stored for assignment to c at t = 1; the sampling precedes the assignment a = 0 at zero time. Hence the value of c at zero time is not decided.
- The increment to a and (the samples of a for subsequent assignment to c) at 0th, 3rd, 6th *etc*., ns values are concurrent. The compiler decides their precedence. With the specific compiler used, the value of a is sampled and only then a is incremented. Hence the assignment to c at the 4th ns is the value of a sampled at the 3rd ns before its increment – that is, 1. Similar is the case with the subsequent assignment changes to c.
- At the 3rd, 6th, 9th, *etc*., ns values, a is sampled and assigned to b as well as d. Hence changes in b and d are identical. Contrast this with the previous example where the assignment sequence

  ```
  c = #1 a;
  d = a;
  ```

  results in different sampling instances and assignments to c and d.

```
module del_dem5;
integer a,b,c,d;
always
begin
                    c = #1 a;
          #2        b = a;
                    d = a;
end
initial
begin
          a = 0; b = 0; c = 0; d = 0;
      #1 a = 1; #2 a = 2; #2 a = 3; #2 a = 4; #2 a = 5; #2 a = 6; #2 a = 7; #2 a = 8;
      #2 a = 9; #2 a = 10;
end
initial $monitor ($time, "  a = %d,  b = %d,  c = %d,  d = %d", a, b, c, d);
endmodule
```

Figure 7.41 Another module to illustrate combinations of delays.

Table 7.4 Simulated results of the module of Figure 7.41

t	0	1	3	4	5	6	7	9	10	11
a	0	1	2	2	3	3	4	5	5	6
b	0	0	1	1	1	3	3	5	5	5
c	0	X	X	1	1	1	3	3	5	5
d	0	0	1	1	1	3	3	5	5	5

t	12	13	15	16	17	18	19	21	22
a	6	7	8	8	9	9	10	10	10
b	6	6	8	8	8	9	9	10	10
c	5	6	6	8	8	8	9	9	10
d	6	6	8	8	8	9	9	10	10

7.7.2 Delay Assignments

In all the illustrations above, delay was specified as a number. It may be a variable or a constant expression. In case it is an expression, it is evaluated and execution delayed by the number of time steps. If the number evaluates to a negative quantity, the same is interpreted as a 2's complement value. In the statement

```
always #b       a = a + 1;
```

a and b are variables. The execution incrementing a is scheduled at b ns. If b changes, the execution time also changes accordingly. As another example consider the procedural assignment

```
always #(b + c)  a = a + 1;
```

Here a, b, and c are variables. The algebraic addition of variables b and c is to be done. The scheduler schedules the incrementing of a and reassigning the incremented values back to a with a time delay of (b + c) ns. As an additional example consider the assignment below with an intra-assignment delay.

```
always  #(a + b) a = #(b + c) a +1;
```

Here the simulator evaluates (a + b) during simulation. After a lapse of (a + b) ns, execution of the statement is taken up; (a + 1) is evaluated and assigned as the new value of a – but the assignment is delayed by (b + c) ns.

7.7.3 Zero Delay

A delay of 0 ns does not really cause any delay. However, it ensures that the assignment following is executed last in the concerned time slot. Often it is used to avoid indecision in the precedence of execution of assignments.

7.8 `wait` CONSTRUCT

The **`wait`** construct makes the simulator wait for the specified expression to be true before proceeding with the following assignment or group of assignments. Its syntax has the form

`wait` (alpha) assignment1;

alpha can be a variable, the value on a net, or an expression involving them. If alpha is an expression, it is evaluated; if true, assignment1 is carried out. One can also have a group of assignments within a block in place of assignment1. The activity is level-sensitive in nature, in contrast to the edge-sensitive nature of event specified through @. Specifically the procedural assignment

@clk a = b;

assigns the value of b to a when clk changes; if the value of b changes when clk is steady, the value of a remains unaltered. In contrast, with

`wait`(clk) #2 a = b;

the simulator waits for the clock to be high and then assigns b to a with a delay of 2 ns. The assignment will be refreshed as long as the clk remains high. The use of wait construct is brought out here through two examples.

Example 7.6

Figure 7.42 shows one version of the up-down counter module along with a test bench. It is a modification of the up down counter of Figure 7.10 and uses a **`wait`** construct. It has an enable input En. The counter is active and counts only when En = 1, that is, from the 5th ns to the 25th ns. The simulation results reproduced in Figure 7.43 confirm this.

```
module ctr_wt(a,clk,N,En);
input clk,En;
input[3:0]N;
output[3:0]a;
reg[3:0]a;
initial a=4'b1111;
always
begin
        wait(En)
        @(negedge clk)
        a=(a==N)?4'b0000:a+1'b1;
end
endmodule
```

continued

continued

```
//TEST_BENCH
module tst_ctr_wt;
reg clk,En;
reg[3:0]N;
wire[3:0]a;
ctr_wt c1(a,clk,N,En);
initial
begin
clk=0;N=4'b1111;En=1'b0;#5 En=1'b1;#20 En=1'b0;
end
always
#2 clk=~clk;
initial #35 $stop;
initial $monitor($time,"clk=%h,En=%b,N=%b,a=%b",clk,En,N,a,);
endmodule
```

Figure 7.42 A counter module to illustrate the use of `wait` construct. The test bench is also shown in the figure.

```
//output
      #                0clk=0,En=0,N=1111,a=1111
      #                2clk=1,En=0,N=1111,a=1111
      #                4clk=0,En=0,N=1111,a=1111
      #                5clk=0,En=1,N=1111,a=1111
      #                6clk=1,En=1,N=1111,a=1111
      #                8clk=0,En=1,N=1111,a=0000
      #               10clk=1,En=1,N=1111,a=0000
      #               12clk=0,En=1,N=1111,a=0001
      #               14clk=1,En=1,N=1111,a=0001
      #               16clk=0,En=1,N=1111,a=0010
      #               18clk=1,En=1,N=1111,a=0010
      #               20clk=0,En=1,N=1111,a=0011
      #               22clk=1,En=1,N=1111,a=0011
      #               24clk=0,En=1,N=1111,a=0100
      #               25clk=0,En=0,N=1111,a=0100
      #               26clk=1,En=0,N=1111,a=0100
      #               28clk=0,En=0,N=1111,a=0101
      #               30clk=1,En=0,N=1111,a=0101
      #               32clk=0,En=0,N=1111,a=0101
      #               34clk=1,En=0,N=1111,a=0101
```

Figure 7.43 Simulation results of the module in Figure 7.42.p

Example 7.7

Figure 7.44 shows a rudimentary and crude version of a serial receiver module and its test bench. Simulation results are shown in Figure 7.45. The module receives serial data on the di line. The data are synchronized to the clock clk. The sequence of operations carried out by the module is as follows:

- Wait for recv input to go high.
- Once recv=1, latch the next 4 successive bits of incoming data into respective bit positions of the do register.

```
 //Example for 'wait'
module sr_rec(do, ack, clk, di, recv);
output [3:0] do; output ack;
input clk, recv, di;
reg [3:0] do; reg ack;
initial ack = 1'b0;
always   begin
                    wait(recv)
                    @(negedge clk) do[0]=di;
                    @(negedge clk) do[1]=di;
                    @(negedge clk) do[2]=di;
                    @(negedge clk) do[3]=di;
                    @(negedge clk) ack = 1'b1;
          end
endmodule

module tst_sr_rec;
reg clk, di, recv;
wire [3:0]do; wire ack;
initial    begin
          clk=1'b0; recv=1'b0; di=1'b0;  #5 recv=1'b1;
          end
always #2clk = ~clk;
initial    begin
          #7di=1'b1; #4di=1'b0; #8di=1'b1; #8di=1'b0;
          end
initial $monitor($time, "clk=%d, recv=%b, di=%b, do=%b, ack=%b",
          clk, recv, di, do, ack);
sr_rec rrcc(do, ack, clk, di, recv);

initial #25 $stop;
endmodule
```

Figure 7.44 A rudimentary serial transmitter module.

```
//output
#                0clk=0, recv=0, di=0, do=xxxx, ack=0
#                2clk=1, recv=0, di=0, do=xxxx, ack=0
#                4clk=0, recv=0, di=0, do=xxxx, ack=0
#                5clk=0, recv=1, di=0, do=xxxx, ack=0
#                6clk=1, recv=1, di=0, do=xxxx, ack=0
#                7clk=1, recv=1, di=1, do=xxxx, ack=0
#                8clk=0, recv=1, di=1, do=xxx1, ack=0
#               10clk=1, recv=1, di=1, do=xxx1, ack=0
#               11clk=1, recv=1, di=0, do=xxx1, ack=0
#               12clk=0, recv=1, di=0, do=xx01, ack=0
#               14clk=1, recv=1, di=0, do=xx01, ack=0
#               16clk=0, recv=1, di=0, do=x001, ack=0
#               18clk=1, recv=1, di=0, do=x001, ack=0
#               19clk=1, recv=1, di=1, do=x001, ack=0
#               20clk=0, recv=1, di=1, do=1001, ack=0
#               22clk=1, recv=1, di=1, do=1001, ack=0
#               24clk=0, recv=1, di=1, do=1001, ack=1
```

Figure 7.45 Simulation results of the module in Figure 7.44.

- Once the above nibble receipt is accomplished, set acknowledgment flag high.
- If recv continues to remain high, the subsequent serial bits will be loaded into the do nibble, again and again in groups of 4 bits.
- If at any time recv goes low, the receipt and the serial to parallel conversion will come to a stop.

7.9 MULTIPLE ALWAYS BLOCKS

All the activities within an always block are scheduled for sequential execution. The activities can be of a combinational nature, a clocked sequential nature, or a combination of these two. (A design description involving such combinations is conventionally called the 'Register Transfer Level' description.) Basically, any circuit block whose end-to-end operation can be described as a continuous sequence can be described within an **`always`** block. A typical circuit block conforming to the above description is shown in Figure 7.46. It has three activities termed A1, A2, and A3. These three are to be done in that order. Activity A1 accepts *x* as input, and it generates output *B* and *p*. *p* and *y* form inputs to activity A2. Similarly activity A2 generates outputs *c* and *q* after activity A1 is completed.

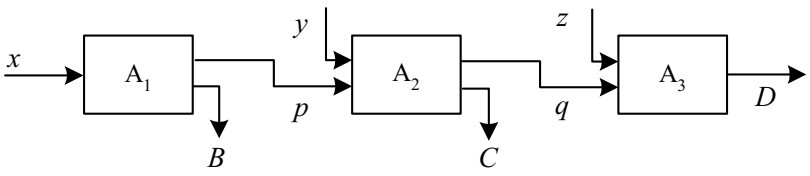

Figure 7.46 A module where execution proceeds through three blocks sequentially.

q and *z* form outputs of A2. After activity A2 is completed, activity A3 is scheduled. It accepts *z* and *q* as inputs and generates *D* as output. Here if A1, A2, and A3 are logical activities, the whole block can be synthesized as a combinational logic unit. If one or more of these are clocked events, execution may be sequential. The design examples considered so far are broadly of this category.

In a comparatively bigger IC, the activity flow can be more complex. One with an additional level of complexity is shown in Figure 7.47. The activities are marked A1-A2-A3 and B1-B2-B3, These are the two streams in the circuit. It is possible that the intermediate results of one may affect the flow of the other. Functioning of two timers – dependent on each other –is a typical example. A processor servicing serial reception and serial transmission simultaneously is another example. In all these cases, each sequential activity is described in a separate always block.

A design of the type in Figure 7.47 can be described with two always blocks. In some others, three or more always blocks may be called for. Examples of such designs are considered later.

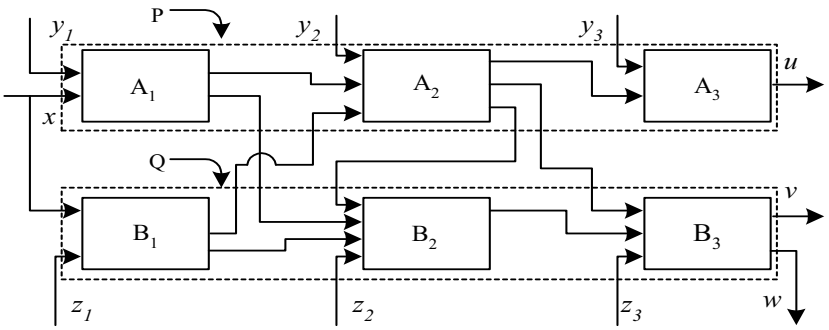

Figure 7.47 A module where execution proceeds concurrently through two groups of blocks.

Activities within one always block are normally sequential – as with the examples considered so far. If necessary, they can be made selectively concurrent. (see Section 7.11). But when designs are spread out in two or more always blocks (with design structures as in Figure 7.47), they are necessarily concurrent. Thus the blocks P and Q in Figure 7.47 are concurrent while the "sub-blocks" within each (namely A1, A2, and A3 within block P and B1, B2, and B3 within block Q) are sequential. In short, with behavioral level descriptions, one can organize the activities to be in concurrent form, in sequential form, or in combinations. In contrast, all design descriptions involving constructs at gate and data flow levels are necessarily concurrent.

7.10 DESIGNS AT BEHAVIORAL LEVEL

All simple algebraic as well as logical expressions can be described at the behavioral level. One can also mix them with blocks at the gate level as well as the data flow level to form composite as well as more involved modules. The simple A-O-I gate is taken as an example below to bring out the possibilities.

Example 7.8

Figure 7.48 shows a module of an AOI gate and its test bench; Figure 7.49 shows the simulation results, and the synthesized circuit is shown in Figure 7.50. The A-O-I gate module has two vector inputs – `a` and `b` – both being two bits wide. The bits of the two vectors are ANDed; the ANDed bits are subsequently used as the inputs to the following NOR gate to form the output. Note the following:

- All the input bits are to figure in the sensitivity list specified to trigger execution. If any one is left out, a change in that will not be reflected in the output immediately.
- The block becomes active, if any bit in the sensitivity list changes value.
- The assignments specified are executed out sequentially – but all at the same time step. Some elaboration is in order here. All the four assignments within the `aoibeh` module of Figure 7.48 are sequentially executed but at the same time step. The values of `a` and `b` displayed at the end of the respective time steps in Figure 7.49 confirm this. Concurrency of the assignments here also leads to a combinational circuit in synthesis.
- All quantities that appear to the left of the assignment statements have to be of the variable type; they have been declared as **`reg`** here.

```
module aoibeh(o,a,b);
output o;
input[1:0]a,b;
reg o,a1,b1,o1;
always@(a[1] or a[0]or b[1]or b[0])
begin
      a1=&a;
      b1=&b;
      o1=a1||b1;
      o=~o1;
end
endmodule

module tst_aoibeh;
reg [1:0]a,b; /*  specicific values will be assigned to
a1,a2,b1, and b2 and these connected
to input ports of the gate insatntiations;
hence these variables are declared as reg */
wire o;
initial
begin
     a[0]=1'b0;a[1] =1'b0;b[0]=1'b0;b[1] =1'b0;
     #3 a[0] =1'b1;
     #3 a[1] =1'b1;
     #3 b[0] =1'b1;
     #3 b[1] =1'b0;
     #3 a[0] =1'b1;
     #3 a[1] =1'b0;
     #3 b[0] =1'b0;
end
initial #100 $stop;//the simulation ends after running
for 100 tu's.
initial $monitor($time, "o =%b,a[0]=%b,a[1]=%b, b[0] =
%b ,b[1] = %b ",o,a[0],a[1],b[0],b[1]);
aoibeh gg(o,a,b);
endmodule
```

Figure 7.48 An A-O-I gate module at the behavioral level and its test bench.

```
 # 0  o = 1,a[0]=0,a[1]=0,b[0]=0,b[1]=0
 # 3  o = 1,a[0]=1,a[1]=0,b[0]=0,b[1]=0
 # 6  o = 0,a[0]=1,a[1]=1,b[0]=0,b[1]=0
 # 9  o = 0,a[0]=1,a[1]=1,b[0]=1,b[1]=0
 #18  o = 1,a[0]=1,a[1]=0,b[0]=1,b[1]=0
 #21  o = 1,a[0]=1,a[1]=0,b[0]=0,b[1]=0
```

Figure 7.49 Simulation results of the module in Figure 7.48.

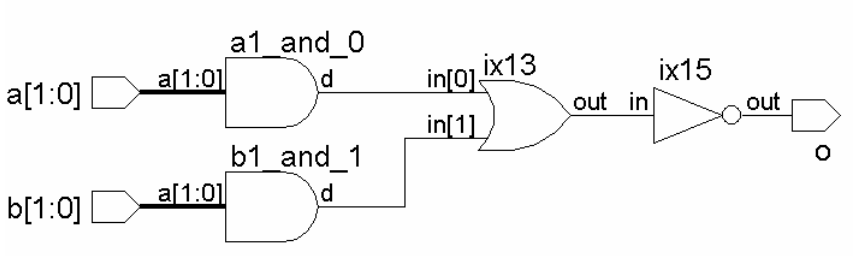

Figure 7.50 Synthesized circuit of the A-O-I module in Figure 7.48.

Example 7.9

Figure 7.51 shows an alternate but more compact description of the A-O-I gate again at the behavioral level. Since the full assignment is realized in one line, no **`begin-end`** type construct is called for. Simulation results are identical to those of Figure 7.49 and are not repeated.

Example 7.10

The AOI gate in Figure 7.51 has again been described as a module in Figure 7.52. Here the AND functions are realized as and-gate primitives. The NOR function alone is realized in behavioral mode. The sensitivity list includes the two outputs of the AND gates. The gate primitives describe a set of two continuous AND functions. In contrast, the NOR function is activated only when `a1` or `b1` changes. Though conceptually different, the latter also results in outputs identical to the continuous assignments. The test bench in Figure 7.51 can be used here by changing the instantiation statements suitably.

```
module aoibeh1(o,a,b);
output o;
input[1:0]a,b;
reg o;
always@(a[1]ora[0]or b[1]orb[0]) o=~((&a)||(&b));
endmodule
```

Figure 7.51 Another realization of the AOI gate at the behavioral level.

```
module aoibeh2(o,a,b);
output o;
input[1:0]a,b;
wire a1,b1;
reg o;
and g1(a1,a[1],a[0]),g2(b1,b[1],b[0]);
always@(a1 or b1)
o=~(a1||b1);
endmodule
```

Figure 7.52 AOI gate realization by the combined use of primitive instantiations and procedural assignments.

Example 7.11

Figure 7.53 shows another realization of the AOI gate. Here the AND functions are realized as continuous assignments. The NOR function is realized as an always block.

```
module aoibeh3(o,a,b);
output o;
input[1:0]a,b;
wire a1,b1;
reg o;
assign a1=&a,b1=&b;
always@(a1 or b1)o=~(a1||b1);
endmodule
```

Figure 7.53 The AOI gate realized by combining continuous assignments and procedural assignments.

Figure 7.54 shows another realization of the AOI gate where a gate primitive, a continuous assignment (at data flow level), and an always block are present.

The examples above bring out the variety of possibilities in design description. Designers' expertise as well as constraints and facilities in the simulation and synthesis tools often limit the choice. More often the same design may have to be described differently as one proceeds from a system level design and simulation to circuit synthesis [Navabi, Palnitkar].

```
module aoibeh4(o,a,b);
output o;
input[1:0]a,b;
wire a1,b1;
reg o;
assign a1=&a;
and g2(b1,b[1],b[0]);
always@(a1 or b1)
o=~(a1||b1);
endmodule
```

Figure 7.54 The AOI gate realized by combining primitive instantiation, continuous assignment, and procedural assignment.

7.11 BLOCKING AND NONBLOCKING ASSIGNMENTS

All assignment within an initial or an always block considered so far are done through an equality ("=") operator. These are executed sequentially – that is, one statement is executed, and only then the following one is executed. Such assignments block the execution of the following lot of assignments at any time step. Hence they are called "blocking assignments". Further, when such a blocking assignment has time delays associated with it, the delay is applicable to the following assignment or activity also. Different examples of groups of blocking assignments have been considered in the preceding sections.

One comes across situations where assignments are to be effected concurrently (as with the continuous assignments considered in the preceding chapter). A facility called the "nonblocking assignment" is available for such situations. The symbol "<=" signifies a non-blocking assignment. The same symbol signifies the "less than or equal to" operator in the context of an operation. The context decides the role of the symbol. The main characteristic of a non-blocking assignment is that its execution is concurrent with that of the following assignment or activity. A discussion of the features of nonblocking assignments and their comparison with blocking assignments are in order here.

Consider the set of nonblocking assignments in Figure 7.55. All three assignments are executed concurrently – that is, A, B, and C are assigned the values 00 01 and 11concurrently and not sequentially. Figure 7.56 shows the same non-blocking assignments with time delays. All three assignments are taken up for execution concurrently. If the block is entered at time step t1,

- A is assigned the value 00 at time step t1.
- B is assigned the value 01 with a time delay of 2 ns – that is, at time t1 + 2 ns.
- C is assigned the value 11 with a delay of 1 ns – that is, at time t1 + 1 ns (and not at time 3 ns as happens with blocking assignments).

```
A <= 2'b00;
B <= 2'b01;
C <= 2'b11;
```

Figure 7.55 A group of nonblocking assignments.

```
A <= 2'b00;
#2 B <= 2'b01;
#1 C <= 2'b11;
```

Figure 7.56 A group of nonblocking assignments with time delays.

Nonblocking assignments are essentially two-step affairs. For all the non-blocking assignments in a block, the right-hand sides are evaluated first. Subsequently the specified assignments are scheduled. Consider the block of assignments in Figure 7.57. First A is assigned the binary value 00, and then B is assigned the value 01. These two assignments are sequential. The subsequent two assignments are concurrent. The assignment

```
A <= b
```

"reads" the value of B, stores it separately, and then assigns it to A. The new value of a is 01. The assignment

```
B <= A ;
```

takes the value of A– i.e., 00 – stores it separately and assigns it to B. Thus the new value of B is 00. After the block is executed, A has the value 01 while B has the value 00. Contrast this with the set of blocking assignments in Figure 7.58. All four assignments here are sequential in nature. The third one, namely

```
A = B;
```

assigns the value 01 to a; subsequently the fourth and following assignment

```
B = A ;
```

assigns the present value of A (*i.e.*, 01) to b; The value of b remains at 01 itself. Consider the block of Figure 7.59. It has three nonblocking assignments. The sequence of execution of the three assignments is as follows:

1. At the positive edge of the clock, values of A, B, and C are read and stored and B &(~c) are computed.

```
A = 2'b00;
B = 2'b01;
A <= B;
B <= A;
```

Figure 7.57 Swapping variable values through nonblocking assignments.

```
A = 2'b00;
B = 2'b01;
A = B;
B = A;
```

Figure 7.58 Another group of blocking assignments.

```
initial
begin
    A=  1'b0;
    B=  1'b1;
    C = 1'b0;
end
always  @(posedge clk)
        begin
        A <= B;
        @(negedge clk) C <= B &(~c);
        #2 B< = C;
        end
```

Figure 7.59 Segment of a module involving blocking and nonblocking assignments.

2. A is assigned the stored value of B (=1); this and the activity in (1) above are carried out concurrently in the same time step.
3. At the next negative clk edge, C is assigned the value of B & (~C) evaluated and stored earlier (=1) – mentioned in (1) above.
4. Two nanoseconds after the positive edge of clk (*i.e.*, after the entry to the block), B is assigned the value of C stored earlier (=0).

In the segment in Figure 7.60, two always blocks do assignments concurrently; both of these are of the blocking variety. The values assigned to A and B are decided by the structure of the simulator. The block has the potential to create a race condition. In contrast, in the segment of Figure 7.61, the two assignments are of the nonblocking type; A is assigned the previous value of B, while B is assigned the previous value of A. The race condition is avoided here.

Observations :

- In a design whenever a number of concurrent data transfers take place after a common event, nonblocking assignments are preferred. The common event forms the sensitivity list followed by the nonblocking assignments.

```
always @(posedge clk)
A = B;
always @(posedge clk)
B = A;
```

Figure 7.60 A set of assignments with a potential race condition.

```
always @(posedge clk)
A <= B;
always @(posedge clk)
B <= A;
```

Figure 7.61 The assignments of Figure 7.60 modified to avoid race condition.

- All nonblocking assignments in a block are executed concurrently. However, the scheduling is done in the same order as the specified statements. If two assignments are done to a **reg** in a time step, the latter prevails. For example with the following sequence of statements in a block,

  ```
  A <= 1;
  A <= 0;
  ```

  A is assigned the value of zero.

- Although blocking and nonblocking assignment can be mixed in a block, many synthesis tools may not support such combinations.

7.11.1 Nonblocking Assignments and Delays

Delays – of the assignment type and the intra-assignment type – can be associated with nonblocking assignments also. The principle of their operation is similar to that with blocking assignments. As explained earlier, the delay values can be constant expressions. Blocking and nonblocking assignments, together with assignment and intra-assignment delays, open up a variety of possibilities. They can be used individually and in combinations to suit different situations. The subtle differences in their use are brought out here through a series of simple illustrations. Some further clarifications regarding assignments and time delays are in order here.

Example 7.12

Consider the module of Figure 7.62, which has a delay of 3 ns for the blocking assignment to c1. If a or b changes, the always block is activated. Three ns later, (a&b) is evaluated and assigned to c1. The event "(a or b)" will be checked for change or trigger again. If a or b changes, all the activities are frozen for 3 ns. If a or b changes in the interim period, the block is not activated. Hence the module does not depict the desired output.

```
module nil1 (c1, a, b);
output c1;
input a, b;
reg c1;
always @(a or b)
        #3        c1 = a&b;
endmodule
```

Figure 7.62 A time delay in an evaluation.

```
module nil2 (c2, a, b);
output c2;
input a, b;
reg c2;
always  @(a or b)
                  c2 = #3  a&b;
endmodule
```

Figure 7.63 An intra-assignment delay.

```
module nil3 (c3, a, b);
output c3;
input a, b;
reg c3;
always @(a or b)
        #3        c3 <= a&b;
endmodule
```

Figure 7.64 A time delay in a non-blocking assignment.

```
module nil4 (c4, a, b);
output c4;
input a, b;
reg c4;
always @(a or b)
        c4 <= #3 a&b;
endmodule
```

Figure 7.65 An intra-assignment delay in a nonblocking assignment.

Consider the module of Figure 7.63 with an intra-assignment delay of 3 ns to the assignment to c2. The always block is activated if a or b changes. (a & b) is evaluated immediately but assigned to c2 only after 3 ns. However, the behavior is not acceptable on two counts:

- The output assignment has to wait for 3 ns after the change.
- Only after the delayed assignment to c2, the event (a or b) checked for change. If a or b changes in the interim period, the block is not activated.

The module in Figure 7.64 has a blocking delay of 3 ns; but the assignment is of the nonblocking type. The block is entered if the value of a or b changes but the evaluation of a&b and the assignment to c3 take place with a time delay of 3 ns. If a or b changes in the interim period, the block is not activated. The module in Figure 7.65 possibly represents the best alternative with time delay. The always block is activated if a or b changes. (a&b) is evaluated immediately and scheduled for assignment to c4 with a delay of 3 ns. Without waiting for the assignment to take effect (*i.e.*, at the same time step as the entry to the block), control is returned to the event control operator. Further changes to a or b – if any – are again taken cognizance of. The assignment is essentially a delay operation.

Figure 7.66 shows the waveforms for c1, c2, c3, and c4 in the modules of Figures 7.62 to 7.65 for representative waveforms of a and b. One can clearly see that c4 has a representation of a & b, which is the most acceptable of the lot.

7.12 THE `case` STATEMENT

The **`case`** statement is an elegant and simple construct for multiple branching in a module. The keywords **`case`**, **`endcase`**, and **`default`** are associated with the **`case`** construct. Format of the **`case`** construct is shown in Figure 7.67. First expression is evaluated. If the evaluated value matches ref1, statement1 is executed; and the simulator exits the block; else expression is compared with

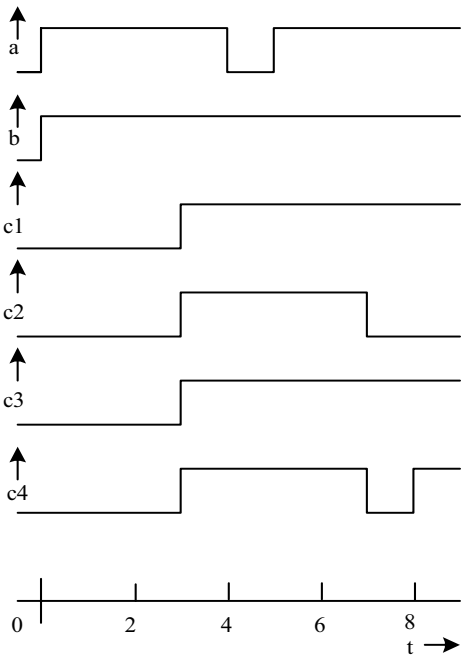

Figure 7.66 Waveforms of c1, c2, c3, and c4 of the modules in Figures 7.62 to 7.65 for representative values of a and b.

ref2 and in case of a match, statement2 is executed, and so on. If none of the ref1, ref2, *etc*., matches the value of expression, the **`default`** statement is executed.

```
Case (expression)
Ref1 : statement1;
Ref2 : statement2;
Ref3 : statement3;
.. .
. . .
default: statementd;
endcase
```

Figure 7.67 Structure of the case statement.

Observations:

- A statement or a group of statements is executed if and only if there is an exact – bit by bit – match between the evaluated expression and the specified ref1, ref2, *etc*.
- For any of the matches, one can have a block of statements defined for execution. The block should appear within the **`begin-end`** construct.
- There can be only one **`default`** statement or **`default`** block. It can appear anywhere in the case statement.
- One can have multiple signal combination values specified for the same statement for execution. Commas separate all of them.

Example 7.13

Consider the module in Figure 7.68 for a 2-to-4 decoder. The test bench is also included in the figure. One of the 4 output bits goes high, depending on the binary value of {i1, i2}. If i1, i2, or both take x or z values, there is no match and the **`default`** block is executed. The simulation results are shown in Figure 7.69.

```
module dec2_4beh(o,i);
output[3:0]o;
input[1:0]i;
reg[3:0]o;
always@(i)
begin
case(i)
      2'b00:o=4'h0;
      2'b01:o=4'h1;
      2'b10:o=4'h2;
      2'b11:o=4'h4;
default:
begin
      $display ("error");
      o=4'h0;
end
endcase
end
endmodule
```

continued

continued

```
//test bench
module tst_dec2_4beh();
wire [3:0]o;
reg[1:0] i;
//reg en;
dec2_4beh dec(o,i);
initial
begin
        i =2'b00;
      #2i =2'b01;
      #2i =2'b10;
      #2i =2'b11;
      #2i =2'b11;
      #2i =2'b0x;
end
initial $monitor ($time  ,  " output o =  %b  , input i
= %b " , o ,i);
endmodule
```

Figure 7.68 A 2-to-4 decoder using the **`case`** statement.

Example 7.14

Consider the module in Figure 7.70, which is a modified version of the decoder module in Figure 7.68. A test bench is also included in the figure. Here if either bit is at **`x`** state, all the output bits are in the **`x`** state. Default corresponds to one or both of the input bits being **`z`** or both the bits being at **`x`** state. In such a case an error message is also output by the simulator. The simulation results are shown in Figure 7.71.

```
output
# 0 output o =  0000  , input i  = 00
# 2 output o =  0001  , input i  = 01
# 4 output o =  0010  , input i  = 10
# 6 output o =  0100  , input i  = 11
# error
# 10 output o =  0000  , input i = 0x
```

Figure 7.69 Simulation results of the decoder module in Figure 7.69.

```
module dec2_4beh1(o,i);
output[3:0]o;
input[1:0]i;
reg[3:0]o;
always@(i)
begin
case(i)
      2'b00:o[0]=1'b1;
      2'b01:o[1]=1'b1;
      2'b10:o[2]=1'b1;
      2'b11:o[3]=1'b1;
      2'b0x,2'b1x,2'bx0,2'bx1:o=4'b0000;
default:    begin
                  $display ("error");
                  o=4'h0;
            end
endcase
end
endmodule

module tst_dec2_4beh1;//test bench
wire [3:0]o;
reg[1:0] i;
dec2_4beh1 dec(o,i);
initial
begin
       i =2'b00;
      #2i =2'b01;
      #2i =2'b10;
      #2i =2'b11;
      #2i =2'b11;
      #2i =2'b1x;
      #2i =2'b0x;
      #2i =2'bx0;
      #2i =2'bx1;
      #2i =2'bxx;
      #2i =2'b0z;
end
initial $monitor ($time  ,  " output o =  %b  , input i
= %b " , o ,i);
endmodule
```

Figure 7.70 A 2-to-4 decoder where all the outputs are forced to zero, if any of the inputs is at **x** state.

```
# 0 output o =  xxx1  , input i  = 00
# 2 output o =  xx11  , input i  = 01
# 4 output o =  x111  , input i  = 10
# 6 output o =  1111  , input i  = 11
# 10 output o = 0000  , input i  = 1x
# 12 output o = 0000  , input i  = 0x
# 14 output o = 0000  , input i  = x0
# 16 output o = 0000  , input i  = x1
# error
# 18 output o = 0000  , input i  = xx
# error
# 20 output o = 0000  , input i  = 0z
```

Figure 7.71 Results of the simulation run with the test bench in Figure 7.70.

Example 7.15 ALU

Figure 7.72 shows an ALU module along with a test bench. The ALU function has been realized through a block with a **case** construct. The ALU realization can be seen to be compact and elegant compared to the versions considered thus far. Additional functions can be added to the ALU by a direct expansion of the **case** block. The ALU size too can be altered to suit requirements. Results of the simulation run with the test bench in Figure 7.72 are shown in Figure 7.73 and Figure 7.74. The synthesized circuit is shown in Figure 7.75.

```
module  alubeh(c,s,a,b,f);
output[3:0]c;
output s;
input [3:0]a,b;
input[1:0]f;
reg s;
reg[3:0]c;
always@(a or b or f)
begin
      case(f)
      2'b00:      c=a+b;
      2'b01:      c=a-b;
      2'b10:      c=a&b;
      2'b11:      c=a|b;
      endcase
```

continued

continued

```
end
endmodule

module tst_alubeh;//test-bench
reg[3:0]a,b;
reg[1:0]f;
wire[3:0]c;
wire s;
alubeh aa(c,s,a,b,f);
initial
begin
f=2'b00;a=2'b00;b=2'b00;
end
always
begin
     #2 f=2'b00;a=4'b0011;b=4'b0000;
     #2 f=2'b01;a=4'b0001;b=4'b0011;
     #2 f=2'b10;a=4'b1100;b=4'b1101;
     #2 f=2'b11;a=4'b1100;b=4'b1101;
end
initial $monitor($time,"f=%b,a=%b,b=%b,c=%b",f,a,b,c);
initial #10 $stop;
endmodule
```

Figure 7.72 A simple ALU module along with its test bench.

```
0f=00,a=0000,b=0000,c=0000
#2f=00,a=0011,b=0000,c=0011
#4f=01,a=0001,b=0011,c=1110
#6f=10,a=1100,b=1101,c=1100
#8f=11,a=1100,b=1101,c=1101
#10f=00,a=0011,b=0000,c=0011
```

Figure 7.73 Results of the simulation run with the test bench in Figure 7.72.

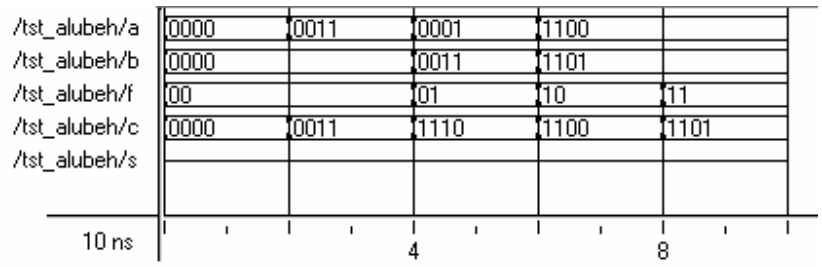

Figure 7.74 Results of the simulation run with the test bench in Figure 7.73 – another view.

7.12.1 `Casex` and `Casez`

The **`case`** statement executes a multiway branching where every bit of the **`case`** expression contributes to the branching decision. The statement has two variants where some of the bits of the **`case`** expression can be selectively treated as don't cares – that is, ignored. **`Casez`** allows **`z`** to be treated as a don't care. "?" character also can be used in place of **`z`**. **`casex`** treats **`x`** or **`z`** as a don't care. An illustrative example using **`casez`** construct follows.

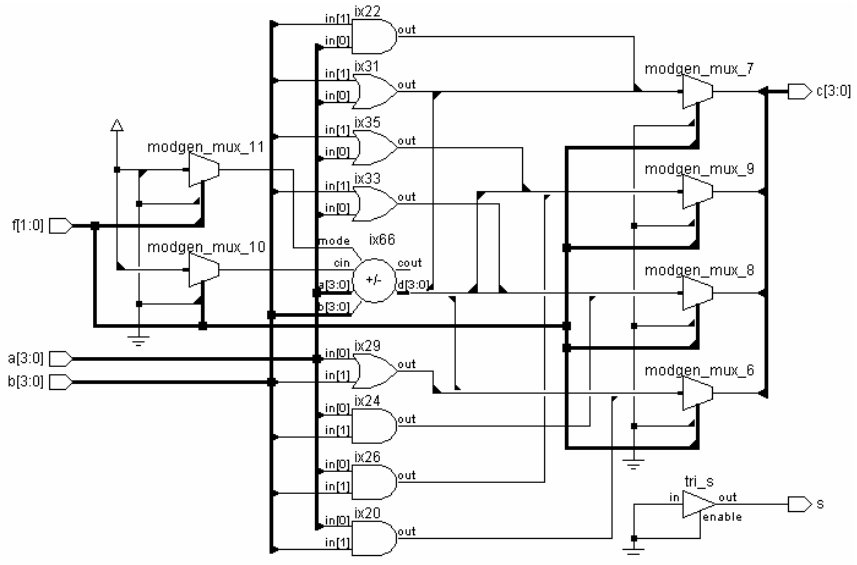

Figure 7.75 Synthesized circuit of the ALU module in Figure 7.73.

Example 7.16

A module for a priority encoder and a test bench for it are shown in Figure 7.76. The encoder gives a 2-bit output. The binary output represents the position of the first one bit in the 4-bit input combination. The simulation results are shown in Figure 7.77. The synthesized circuit is shown in Figure 7.78.

```
module pri_enc(a,b);
output[1:0]a;
input[3:0]b;
reg[1:0]a;
always@(b)
casez(b)
4'bzzz1:a=2'b00;
4'bzz10:a=2'b01;
4'bz100:a=2'b10;
4'b1000:a=2'b11;
endcase
endmodule

module pri_enc_tst;//test-bench
reg [3:0]b;
wire[1:0]a;
pri_enc pp(a,b);
initial b=4'bzzz0;
always
begin
     #2 b=4'bzzz1;
     #2 b=4'bzzz1;
     #2 b=4'bzz10;
     #2 b=4'bz100;
     #2 b=4'b1000;
end
initial $monitor($time, "input b =%b,a  =%b ",b,a);
initial #40 $stop;
endmodule
```

Figure 7.76 A design module for a 2-bit priority encoder using the **`casez`** statement; a test bench is also shown.

```
 0input  b  =  zzz0  ,a  =  01
 2input  b  =  zzz1  ,a  =  00
 6input  b  =  zz10  ,a  =  01
 8input  b  =  z100  ,a  =  10
10input  b  =  1000  ,a  =  11
12input  b  =  zzz1  ,a  =  00
16input  b  =  zz10  ,a  =  01
18input  b  =  z100  ,a  =  10
20input  b  =  1000  ,a  =  11
22input  b  =  zzz1  ,a  =  00
26input  b  =  zz10  ,a  =  01
28input  b  =  z100  ,a  =  10
30input  b  =  1000  ,a  =  11
32input  b  =  zzz1  ,a  =  00
36input  b  =  zz10  ,a  =  01
38input  b  =  z100  ,a  =  10
```

Figure 7.77 Results of simulating the test bench in Figure 7.76.

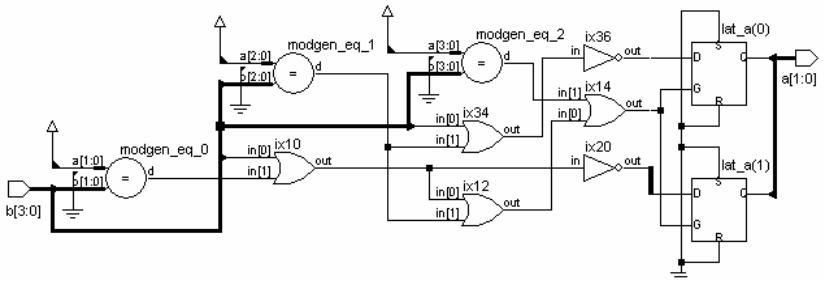

Figure 7.78 Synthesized circuit of the priority encoder in Figure 7.76.

7.13 SIMULATION FLOW

Different constructs for design description and simulation have been dealt with so far. These can be at different levels of abstraction – gate, data flow, or behavioral level. The constructs to be discussed in the following chapters add to the variety and flexibility. Such elements in different combinations make up the design and simulation modules in Verilog. Further, as an HDL, Verilog has to be an inherently parallel processing language. The fact that all the elements of a digital circuit (or any electronic circuit for that matter) function and interact continuously conforming to their interconnections demands parallel processing. In Verilog the parallel processing is structured through the following [IEEE]:

- Simulation time: Simulation is carried out in simulation time. The simulator functions with simulation time advancing in (equal) discrete steps.
- At every simulation step a number of active events are sequentially carried out.
- The simulator maintains an event queue – called the "Stratified Event Queue" – with an active segment at its top. The top most event in the active segment of the queue is taken up for execution next.
- The active event can be of an update type or evaluation type.
  The evaluation event can be for evaluation of variables, values on nets, expressions, *etc*.
  Refreshing the queue and rearranging it constitutes the update event.
- Any updating can call for a subsequent evaluation and *vice versa*.
- Only after all the active events in a time step are executed, the simulation advances to the next time step.

Completion of the sequence of operations above at any time step signifies the parallel nature of the HDL.

A number of active events can be present for execution at any simulation time step; all may vie for "attention." Amongst these, an event specified at #0 time is scheduled for execution at the end – that is, before simulation advances to the next time step. The order, in which the other events are executed, is essentially simulator-dependent.

7.13.1 Stratified Event Queue

The events being carried out at any instant give rise to other events – inherent in the execution process. All such events can be grouped into the following 5 types:

- Active events – explained above.
- Inactive events – The inactive events are the events lined up for execution immediately after the execution of the active events. Events specified with zero delay are all inactive events.
- Blocking Assignment Events – Operations and processes carried out at previous time steps with results to be updated at the current time step are of this category.
- Monitor Events – The Monitor events at the current time step – **`$monitor`** and **`$strobe`** – are to be processed after the processing of the active events, inactive events, and nonblocking assignment events.
- Future events – Events scheduled to occur at some future simulation time are the future events.

The simulation process conforming to the stratified event queue is shown in flowchart form in Figure 7.79.

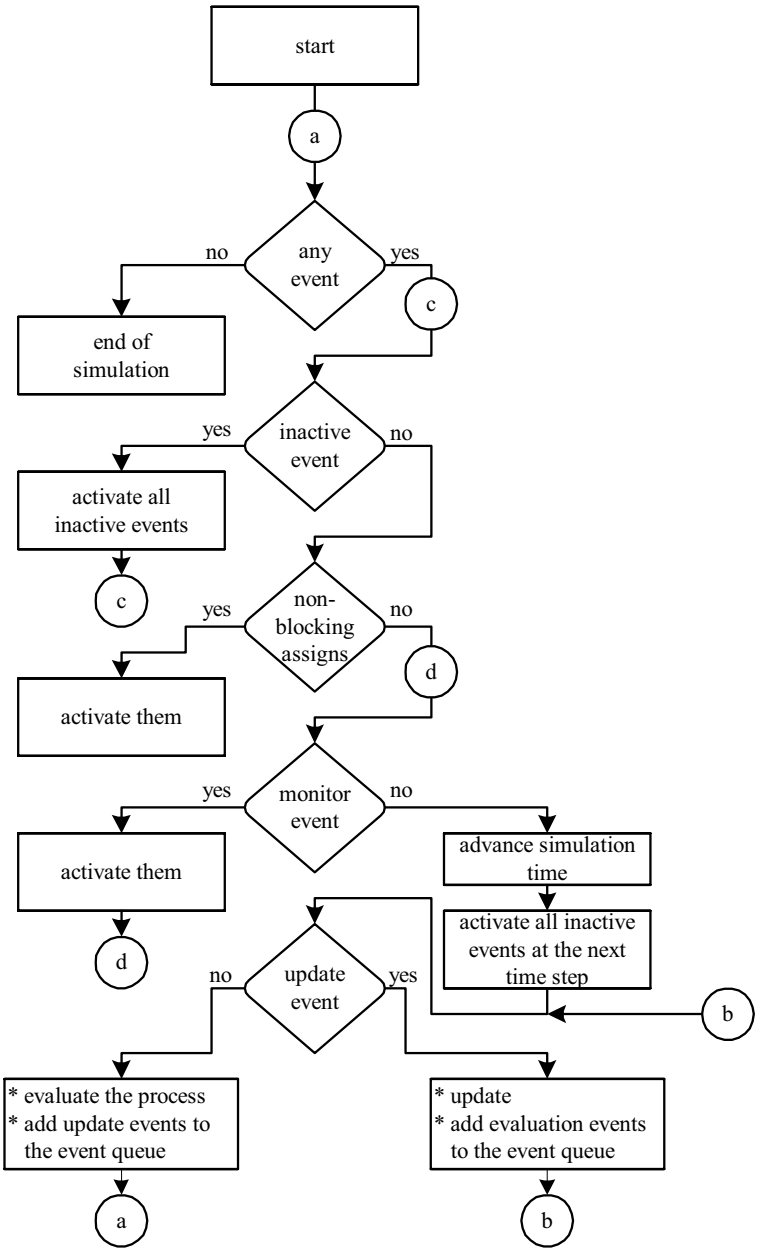

Figure 7.79 Flowchart for the simulation flow.

7.14 EXERCISES

Prepare design modules for the following operations [Sedra, Tocci, Wakerly]. In each case prepare a suitable test bench and test the design module.

1. Add two BCD nibbles.
2. Add two pairs of BCD nibbles – 2 decimal numbers each of two digits.
3. Interrupt Service Routine (ISR): An ISR receives an Interrupt Request (IRQ). The PC content is saved on a stack – 4 bytes deep. Then a specific byte 5a5ah is loaded into the PC. The ISR sets an INTA flag high and returns. Use the '**`wait`**' construct and design the module.
4. Form an ALU for two input bytes. All the operations are to be carried out using the "**`case`**" construct. Use the algebraic and logic instructions available with 8085, 6805, 6502 z80, and the PIC series of processors as the basis [in all 5 ALUs]. Designate the two input vectors as ba and bb. Output is on ba. All the flags are to be on bb.
5. Memory Block: Have a 1 kb size memory with a 10-bit Memory Address Register. Use clock beta for memory read and memory write. Use Wr and Rd as two separate control input lines. The operations to be realized are:
   - Wr=1: Write into the location specified by the MAR.
   - RD=1: Read from location specified by MAR.
   - Wr=0 & Rd=0: Condition to be satisfied to write into the MAR.

   Data input and data output are to be through an 8-bit-wide bus "ba."

6. Change the always block in Example 7.6 (Figure 7.42) to the following:

```
always
begin
        @(negedge clk)
        a=(En)?((a==N)?4'b0000:a+1'b1):a);
end
```

   How does the block here differ from that in the Example? Prepare a test bench, simulate, and explain.
7. A priority encoder is used to prioritize service to interrupt requests in a microcontroller. The priority encoder in Example 7.15 can be expanded to suit the desired role here. It receives a byte (IRQ byte) and outputs a byte (vector address). The vector address has to be 32 times the serial number of the leading one bit in the IRQ byte. Prepare the necessary design module and synthesize it.

8

BEHAVIORAL MODELING II

8.1 INTRODUCTION

Comparatively simple and direct behavioral level constructs were discussed in the last chapter. They are essentially centered on the algebraic or logic operators. Different combinational and sequential circuits can be realized using them. The **`case`** construct and its variants enhance the possibilities of design description considerably. A few constructs are available for looping and branching. Their usage can make the design description compact and elegant. Further such constructs enhance the modeling capabilities substantially [Navabi]. These constructs mostly follow their counterparts in C language [Gottfried]. These constructs and certain other facilities that add to the flexibility of test benches are discussed here. Their use is illustrated through appropriate examples.

8.2 `if` AND `if-else` CONSTRUCTS

The **`if`** construct checks a specific condition and decides execution based on the result. Figure 8.1 shows the structure of a segment of a module with an **`if`** statement. After execution of assignment1, the condition specified is checked. If it is satisfied, assignment2 is executed; if not, it is skipped. In either case the execution continues through assignment3, assignment4, *etc*. Execution of assignment2 alone is dependent on the condition. The rest of the sequence remains. The flowchart equivalent of the execution is shown in Figure 8.2. If the

```
. . .
assignment1;
if (condition) assignment2;
assignment3;
assignment4;
. . .
```

Figure 8.1 Use of **if** construct.

Design Through Verilog HDL. T. R. Padmanabhan and B. Bala Tripura Sundari

ISBN: 0-471-44148-1

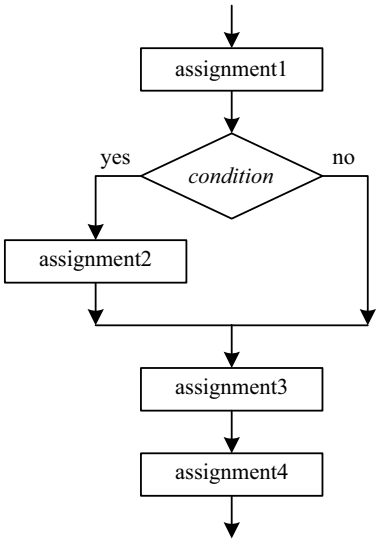

Figure 8.2 Flowchart of the **`if`** loop.

number of assignments associated with the **`if`** condition is more than 1, the whole set of them can be grouped within a **`begin-end`** block. Figure 8.3 shows a segment of a design using the **`if`** construct. It is a ring counter, which shifts one bit right at every clock pulse. The shift operation shifts the a byte right by one bit and fills the vacated bit – a[7] – with a zero. It is set to 1 if the bit shifted out last – a[0] – was a 1. The same is carried out through the **`if`** statement. The **`if-else`** construct is more common and turns out to be more useful than the **`if`** construct taken alone. Figure 8.4 shows the use in a typical design description. Figure 8.5 shows the same in flowchart form. The design description has two branches; the alternative taken is decided by the *condition*:

```
Reg[7:0] a;
Reg c;
always@(posedge clk)
begin
        c = a[0];
        a = a>>1'b1; // Since the vacated bit of a is filled with a zero, it need be
        if(  c ) a[7] = c;// set only if a[0] =1
end
```

Figure 8.3 A Ring counter description using the **`if`** construct.

```
. . .
assignment1;
if(condition)
        begin           // Alternative 1
                assignment2;
                assignment3;
        end
else
        begin           //alternative 2
                assignment4;
                assignment5;
        end
assignment6;
. . .
. . .
```

Figure 8.4 Use of the **`if-else`** construct.

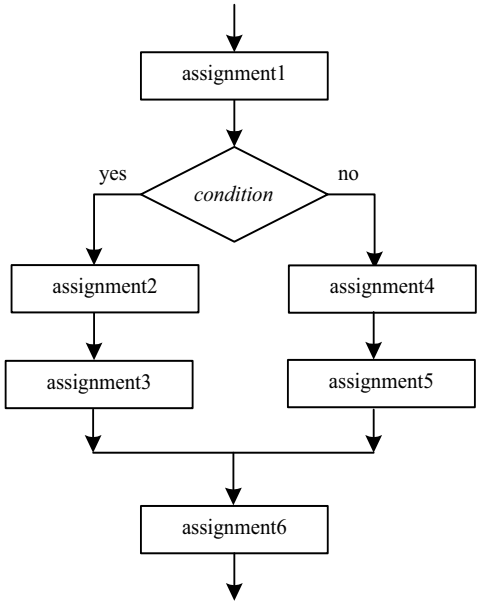

Figure 8.5 Flowchart of execution of the **`if-else`** loop.

- After the execution of assignment1, if the *condition* is satisfied, alternative1 is followed and assignment2 and assignment3 are executed. Assignment4 and assignment 5 are skipped and execution proceeds with assignment6.
- If the *condition* is not satisfied, assignment2 and assignment3 are skipped and assignment4 and assignment5 are executed. Then execution continues with assignment6.

Example 8.1

Figure 8.6 shows a 2 to 4 demux module. The whole demux module is realized through the **`if-else-if`** sequence of constructs. The selected channel is connected to the output, and all other channels are tri-stated. A test bench for the demux module is also shown in the figure. Partial results of simulation are shown in Figure 8.7; the synthesized circuit is shown in Figure 8.8.

In fact the use of **`case`** statement to realize mux, demux, direct encoders, and decoders makes the design description simple and direct – in contrast to the use of **`if-else-if`** construct. But the **`if-else-if`** construct is more general. It can accommodate different types of conditions at each branching. In contrast the **`case`** construct does a direct multiway branching.

```
module demux(a,b,s);
output [3:0]a;
input b;
input[1:0]s;
reg[3:0]a;
always@(b or s)
begin
      if(s==2'b00)
      begin
            a[2'b0]=b;
            a[3:1]=3'bZZZ;
      end
      else if(s==2'b01)
      begin
            a[2'd1]=b;
            {a[3],a[2],a[0]}=3'bZZZ;
      end
else if(s==2'b10)
      begin
            a[2'd2]=b;
            {a[3],a[1],a[0]}=3'bZZZ;
```

continued

continued

```
      end
else
      begin
            a[2'd3]=b;
            a[2:0]=3'bZZZ;
      end
end
endmodule

//tst_bench
module tst_demux();
reg b;
reg[1:0]s;
wire[3:0]a;
demux d1(a,b,s);
initial
b=1'b0;
always
begin
      #2 s=2'b00;b=1'b1;
      #2 s=2'b00;b=1'b0;
      #2 s=2'b01;b=1'b0;
      #2 s=2'b10;b=1'b1;
      #2 s=2'b11;b=1'b0;
end
initial
$monitor("t=%0d, s=%b,b=%b,output =%b",$time,s,b,a);
initial #30 $stop;
endmodule
```

Figure 8.6 A 2-to-4 demux module using the if-else-if construct: A testbench is also shown in the figure.

```
# t=0, s=xx,b=0,output a=0zzz
# t=2, s=00,b=1,output a=zzz1
# t=4, s=00,b=0,output a=zzz0
# t=6, s=01,b=0,output a=zz0z
# t=8, s=10,b=1,output a=z1zz
# t=10, s=11,b=0,output a=0zzz
```

Figure 8.7 Partial results of the simulation of the testbench in Figure 8.6.

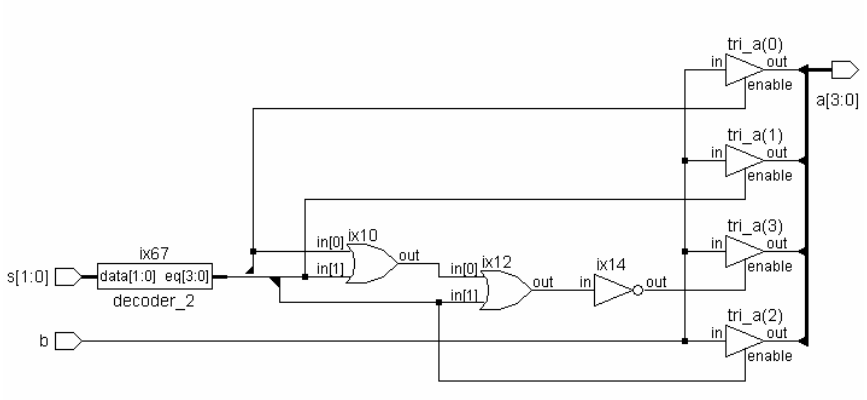

Figure 8.8 The synthesized circuit of the 2-to-4 demux module in Figure 8.6.

Example 8.2

Figure 8.9 shows the design description of a mod-*n* up-counter module along with a test-bench for it. At every clock pulse the counter advances by one bit. As soon as the count reaches the binary value n, the counter is reset to zero. The initial value of n is specified within the module itself. It is changed at a later stage. The simulation results are shown in Figure 8.10 (only partial results are shown).

Observations:

- The **`$write`** is a system task; it is similar to the **`$display`** task except in one respect: When $**`write`** is executed, the simulator does not advance to the new line after the specified display [see Chapter 11 for details].
- The value of n can be changed only from within the module. If necessary, the constraint can be removed by making n as an input to the module.
- The character set '%0d' within the $write statement ensures that the concerned quantity is displayed in decimal form with the minimum number of digits necessary for it. It makes the display elegant.
- For convenience the value of time is displayed in decimal form. Other quantities are in hex form.
- The counter can be easily modified to function as a down counter, a clock divider, or an up / down counter. It can be made more versatile with additional control inputs for Preset, Reset, and Enable.

```
//counter using if else if;
module countif(a,clk);
output[7:0]a;
input clk;
reg[7:0]a,n;
initial
begin
      n=8'h0a;
      a=8'b00000000;
  #45 n=8'h23;
end
always@(posedge clk)
begin
      $write ("time=%0d ",$time);
      if(a==n)
      a=8'h00;
      else a=a+1'b1;
end
endmodule

module tst_countif();//test-bench
reg clk;
wire[7:0]a;
countif c1(a,clk);
initial clk =1'b0;
always
#2clk=~clk;
initial
$monitor("  n=%h, a=%h",c1.n,a);
initial #200 $stop;
endmodule
```

Figure 8.9 A counter realized using the **`if-else`** construct.

8.3 `assign-deassign` CONSTRUCT

A behavior block is activated by the event at the beginning. A proper operation demands that all variables with assignments within the block are to be included in the sensitivity list. The **`assign`** – **`deassign`** constructs allow continuous assignments within a behavioral block. By way of illustration, consider the following simple block:

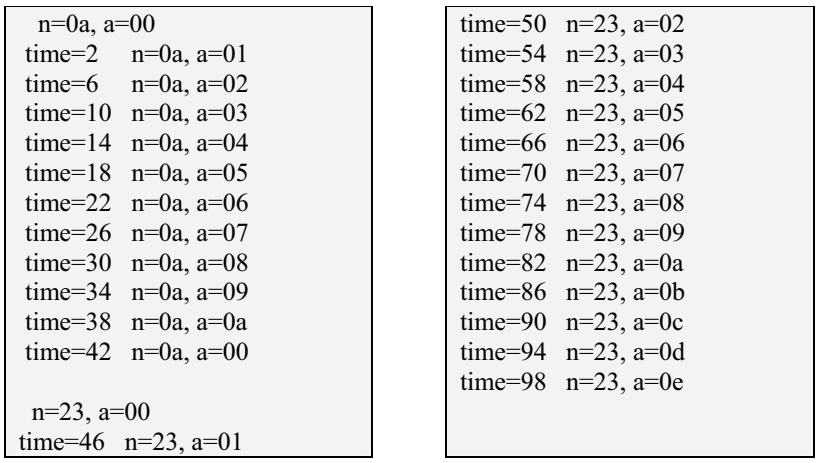

Figure 8.10 Partial results of running the test bench in Figure 8.9.

```
always@(posedge clk) a = b;
```

By way of execution, at the positive edge of clk the value of b is assigned to variable a, and a remains frozen at that value until the next positive edge of clk. Changes in b in the interval are ignored.

As an alternative, consider the block

```
always@(posedge clk) assign c = d;
```

Here at the positive edge of clk, c is assigned the value of d in a continuous manner; subsequent changes in d are directly reflected as changes in variable c: The assignment here is akin to a direct (one way) electrical connection to c from d established at the positive edge of clk.

Again consider an enhanced version of the above block as

```
Always
Begin
        @(posedge clk) assign c = d;
        @(negedge clk) deassign c;
end
```

The above block signifies two activities:

1. At the positive edge of clk, c is assigned the value of d in a continuous manner (as mentioned above).
2. At the following negative edge of clk, the continuous assignment to c is removed; subsequent changes to d are not passed on to c; it is as though c is electrically disconnected from d.

The above sequence of twin activities is repeated cyclically.

In short, assign allows a variable or a net change in the sensitivity list to mandate a subsequent continuous assignment within. **deassign** terminates the assignment done through the **assign-**based statement. The assignment to c in the above two cases is referred to as a "Procedural Continuous Assignment."

Example 8.3 A 2 to 4 Demux through Procedural Continuous Assignment

Consider the mux module in Figure 8.11. It is activated whenever **s** changes. But the assignment is continuous to **reg b**. It is achieved through the use of the

```
//an alternate realization of the demux using the assign construct
module demux1(a0,a1,a2,a3,b,s);
output a0,a1,a2,a3;
input b;
input [1:0]s;
reg a0,a1,a2,a3;
always@(s)
        if(s==2'b00)
        assign {a0,a1,a2,a3}={b,3'oz};
        else if(s==2'b01)
        assign {a0,a1,a2,a3}={1'bz,b,2'bz};
        else if(s==2'b10)
        assign {a0,a1,a2,a3}={2'bz,b,1'bz};
        else if(s==2'b11)
        assign {a0,a1,a2,a3}={3'oz,b};
endmodule

module tst_demux1();
reg b;
reg[1:0]s;
demux1 d2(a0,a1,a2,a3,b,s);
initial begin b=1'b0;s=2'b0; end
always
begin
#1 s=s+1'b1;
$display("t=%0d, s=%b, b=%b,  {a0,a1,a2,a3} =%b",$time,s,b,{a0,a1,a2,a3});
#1b=~b;
$display("t=%0d, s=%b, b=%b,  {a0,a1,a2,a3} =%b",$time,s,b,{a0,a1,a2,a3});
end
initial #14 $stop;
endmodule
```

Figure 8.11 An alternate realization of the demux using the **assign** construct.

```
# t=1, s=01, b=0,  {a0,a1,a2,a3} =0zzz
# t=2, s=01, b=1,  {a0,a1,a2,a3} =z0zz
# t=3, s=10, b=1,  {a0,a1,a2,a3} =z1zz
# t=4, s=10, b=0,  {a0,a1,a2,a3} =zz1z
# t=5, s=11, b=0,  {a0,a1,a2,a3} =zz0z
# t=6, s=11, b=1,  {a0,a1,a2,a3} =zzz0
# t=7, s=00, b=1,  {a0,a1,a2,a3} =zzz1
# t=8, s=00, b=0,  {a0,a1,a2,a3} =1zzz
# t=9, s=01, b=0,  {a0,a1,a2,a3} =0zzz
# t=10, s=01, b=1,  {a0,a1,a2,a3} =z0zz
# t=11, s=10, b=1,  {a0,a1,a2,a3} =z1zz
# t=12, s=10, b=0,  {a0,a1,a2,a3} =zz1z
# t=13, s=11, b=0,  {a0,a1,a2,a3} =zz0z
```

Figure 8.12 Results of simulating the test bench of Figure 8.11.

"**assign**" construct. Specifically, if s = 2'b01, a[1] is connected to b and remains so connected so long as s remains unchanged. If b changes value, a[1] follows it even though b is not included in the sensitivity list. A test bench is also included in Figure 8.11.Simulation results are shown in Figure 8.12.

Example 8.4 A D Flip-Flop through **assign** *–* **deassign** *Constructs*

Consider the module Figure 8.13, which represents a D flip-flop with Preset and Clear. If Clear or Preset becomes true, the output is forced to the Preset or Set condition, respectively. It is ensured by the first always block with the quasi-continuous assignments. If both Preset and Clear are false, the quasi-continuous assignment is removed. The second always block provides the assignment to q at every positive edge of the clk. It can take effect only if the asynchronous set–reset block is not active. Thus the asynchronous set/reset through Preset/Clear override the synchronous set/reset decided by the value of di at the clock edge. A test bench for the D flip-flop module is also included in the figure.

Observations:

- Some (many) synthesizers may not support the quasi-continuous **assign-deassign** constructs.
- The quasi-continuous assignment is made only to a variable (**reg** type); it can be a scalar or a full vector but not a part vector.
- The quasi-continuous assignment overrides all other assignments to the variable.

```
module dffassign(q,qb,di,clk,clr,pr);
output q,qb;
input di,clk,clr,pr;
reg q;
assign qb=~q;
always@(clr or pr)
begin
      if(clr)assign q = 1'b0;//asynchronous clear and
      if(pr) assign q = 1'b1;// preset of FF overrides
      else deassign q;// the synchronous behaviour
end
always@(posedge clk)
      q = di;//synchronous (clocked)value assigned to q
endmodule

//test-bench
module dffassign_tst();
reg di,clk,clr,pr;
wire q,qb;
dffassign dd(q,qb,di,clk,clr,pr);
initial
begin
      clr=1'b1;pr=1'b0;clk=1'b0;di=1'b0;
end
always
begin
      #2 clk=~clk;clr=1'b0;
end
always
# 4 di =~di;
always
#16 pr=1'b1;
always
#20 pr =1'b0;
initial  $monitor("t=%0d, clk=%b, clr=%b, pr=%b,
di=%b, q=%b ",    $time,clk,clr,pr,di,q);
initial #46 $stop;
endmodule
```

Figure 8.13 Design description of a D_flip-flop with Preset and Clear facilities: The module illustrates the use of the **`assign-deassign`** construct.

```
# t=0, clk=0, clr=1, pr=0,  di=0, q=0
# t=2, clk=1, clr=0, pr=0,  di=0, q=0
# t=4, clk=0, clr=0, pr=0,  di=1, q=0
# t=6, clk=1, clr=0, pr=0,  di=1, q=1
# t=8, clk=0, clr=0, pr=0,  di=0, q=1
# t=10, clk=1, clr=0, pr=0,  di=0, q=0
# t=12, clk=0, clr=0, pr=0,  di=1, q=0
# t=14, clk=1, clr=0, pr=0,  di=1, q=1
# t=16, clk=0, clr=0, pr=1,  di=0, q=1
# t=18, clk=1, clr=0, pr=1,  di=0, q=1
# t=20, clk=0, clr=0, pr=0,  di=1, q=1
# t=22, clk=1, clr=0, pr=0,  di=1, q=1
# t=24, clk=0, clr=0, pr=0,  di=0, q=1
# t=26, clk=1, clr=0, pr=0,  di=0, q=0
# t=28, clk=0, clr=0, pr=0,  di=1, q=0
# t=30, clk=1, clr=0, pr=0,  di=1, q=1
# t=32, clk=0, clr=0, pr=1,  di=0, q=1
# t=34, clk=1, clr=0, pr=1,  di=0, q=1
# t=36, clk=0, clr=0, pr=1,  di=1, q=1
# t=38, clk=1, clr=0, pr=1,  di=1, q=1
# t=40, clk=0, clr=0, pr=0,  di=0, q=1
# t=42, clk=1, clr=0, pr=0,  di=0, q=0
# t=44, clk=0, clr=0, pr=0,  di=1, q=0
```

Figure 8.14 Simulation results of the test bench in Figure 8.13.

Example 8.5 Another D Flip-Flop with `if` and `if-else`

Figure 8.15 shows a module of a flip-flop again using the **`if-else-if`** construct. clr, pr, and clk are all included in the sensitivity list itself. A test bench is also included in the figure. The synthesized circuit of the module is shown in Figure 8.16. Simulation results are in Figure 8.17.

```
module dffalter(q,qb,di,clk,clr,pr);
output q,qb;
input di,clk,clr,pr;
reg q;
assign qb =~q;//continous assignment
always@(posedge clr or posedge pr or posedge clk)
      begin
            if(clr) q=1'b0;
```

continued

continued

```
            else if(pr) q=1'b1;
            else q=di;
      end
endmodule

//test-bench
module dffalter_tst();
reg di,clk,clr,pr;
wire q;
dffalter dff(q,qb,di,clk,clr,pr);
initial
begin
      clr=1'b1;pr=1'b0;clk=1'b0;di=1'b0;
end
always
begin
      #2 clk=~clk;clr=1'b0;
end
always  # 4 di =~di;
always  #16 pr=1'b1;
always  #20 pr =1'b0;
initial  $monitor("t=%0d, clk=%b, clr=%b, pr=%b,
di=%b, q=%b ",   $time,clk,clr,pr,di,q);
initial #46 $stop;
endmodule
```

Figure 8.15 An alternate description of the D_FF module and its test bench.

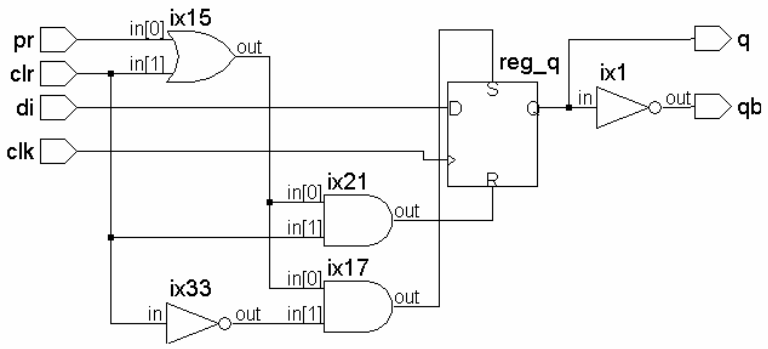

Figure 8.16 Synthesized circuit of the flip-flop of Example 8.5.

```
# 0clk = 0,  clr = 1, pr = 0, di = 0, q = 0
# 2clk = 1,  clr = 0, pr = 0, di = 0, q = 0
# 4clk = 0,  clr = 0, pr = 0, di = 1, q = 0
# 6clk = 1,  clr = 0, pr = 0, di = 1, q = 1
# 8clk = 0,  clr = 0, pr = 0, di = 0, q = 1
# 10clk = 1, clr = 0, pr = 0, di = 0, q = 0
# 12clk = 0, clr = 0, pr = 0, di = 1, q = 0
# 14clk = 1, clr = 0, pr = 0, di = 1, q = 1
# 16clk = 0, clr = 0, pr = 1, di = 0, q = 1
# 18clk = 1, clr = 0, pr = 1, di = 0, q = 1
# 20clk = 0, clr = 0, pr = 0, di = 1, q = 1
# 22clk = 1, clr = 0, pr = 0, di = 1, q = 1
# 24clk = 0, clr = 0, pr = 0, di = 0, q = 1
# 26clk = 1, clr = 0, pr = 0, di = 0, q = 0
# 28clk = 0, clr = 0, pr = 0, di = 1, q = 0
# 30clk = 1, clr = 0, pr = 0, di = 1, q = 1
# 32clk = 0, clr = 0, pr = 1, di = 0, q = 1
# 34clk = 1, clr = 0, pr = 1, di = 0, q = 1
# 36clk = 0, clr = 0, pr = 1, di = 1, q = 1
# 38clk = 1, clr = 0, pr = 1, di = 1, q = 1
# 40clk = 0, clr = 0, pr = 0, di = 0, q = 1
# 42clk = 1, clr = 0, pr = 0, di = 0, q = 0
# 44clk = 0, clr = 0, pr = 0, di = 1, q = 0
```

Figure 8.17 Simulation results for the test bench of Figure 8.15.

Examle 8.6 A Counter with a Continuous Procedural Assignment

Figure 8.18 shows the module of an up counter with Preset and Clear facilities. Preset and Clear are carried out through Procedural Continuous Assignments. If clr goes high, a is reset to zero. If pr goes high, a is set to the number specified as n. Either of these assignments will remain as long as either Clear or Preset is active as the case may be. If both these asynchronous control signals go low, the module increments the value of a at every positive edge of the clock. The module can easily be modified to function as a down counter, an up–down counter, or a counter to any other modulus.

```
module ctr_a(a,n,clr,pr,clk);
output [7:0]a;
input [7:0]n;
input clr,pr,clk;
reg[7:0]a;
initial a =8'h00;
always@(posedge clk)
a=a+1'b1;
always@(clr or pr)
      if (clr)assign a =7'h00;
            else if(pr)assign a =n;
                  else deassign a;
endmodule


module counprclrasgn_tst();//test-bench
reg [7:0]n;
reg clr,pr,clk;
wire[7:0] a;
ctr_a cc(a,n,clr,pr,clk);
initial
begin
      n=8'h55; clr=1'b1;
      pr=1'b0;clk=1'b0;
end
always
begin
      #2 clk=~clk;clr=1'b0;
end
always #16 pr=1'b1;
always #20 pr =1'b0;
initial  $monitor(   $time  ,  "clk  = %b , clr =  %b
, pr  =   %b  ,   a  =  %b ", clk,clr,pr,a);
initial #44 $stop;
endmodule
```

Figure 8.18 Design description of an up counter with Preset and Clear facilities.

```
 0clk = 0 , clr = 1 , pr = 0 , a = 00000000
 2clk = 1 , clr = 0 , pr = 0 , a = 00000001
 4clk = 0 , clr = 0 , pr = 0 , a = 00000001
 6clk = 1 , clr = 0 , pr = 0 , a = 00000010
 8clk = 0 , clr = 0 , pr = 0 , a = 00000010
10clk = 1 , clr = 0 , pr = 0 , a = 00000011
12clk = 0 , clr = 0 , pr = 0 , a = 00000011
14clk = 1 , clr = 0 , pr = 0 , a = 00000100
16clk = 0 , clr = 0 , pr = 1 , a = 01010101
18clk = 1 , clr = 0 , pr = 1 , a = 01010101
20clk = 0 , clr = 0 , pr = 0 , a = 01010101
22clk = 1 , clr = 0 , pr = 0 , a = 01010110
24clk = 0 , clr = 0 , pr = 0 , a = 01010110
26clk = 1 , clr = 0 , pr = 0 , a = 01010111
28clk = 0 , clr = 0 , pr = 0 , a = 01010111
30clk = 1 , clr = 0 , pr = 0 , a = 01011000
32clk = 0 , clr = 0 , pr = 1 , a = 01010101
34clk = 1 , clr = 0 , pr = 1 , a = 01010101
36clk = 0 , clr = 0 , pr = 1 , a = 01010101
38clk = 1 , clr = 0 , pr = 1 , a = 01010101
40clk = 0 , clr = 0 , pr = 0 , a = 01010101
42clk = 1 , clr = 0 , pr = 0 , a = 01010110
```

Figure 8.19 Simulation results of the test bench in Figure 8.18.

Example 8.7

Consider the module in Figure 8.20 which is a variant of the flip-flop in Figure 8.15. A test bench for the flip-flop is also included in the figure. Here the **`always`** block is activated at every positive edge of the clock. At that instant if clr = 1, the flip-flop is cleared. If pr = 1, the flip-flop is set. If clr = 0 and pr = 0, the flip-flop output takes on the value of d. Here all the assignments to q take effect at the positive edge of the clock. Hence the behavior is fully synchronous. This is not necessarily the case with the flip-flop of Figure 8.15. The synthesized circuit of the flip-flop is shown in Figure 8.21.

```
module dff_1beh(q,qb,di,clk,clr,pr);
output q,qb;
input di,clk,clr,pr;
reg q;
assign qb=~q;
always@(posedge clk)
begin
      if(clr)q = 1'b0;
```

continued

continued

```
      else if(pr) q = 1'b1;
      else  q=di;
end
endmodule

//test-bench
module dff_1beh_tst();
reg di,clk,clr,pr;
wire q,qb;
dff_1beh dd(q,qb,di,clk,clr,pr);
initial
begin
clr=1'b1;pr=1'b0;clk=1'b0;di=1'b0;
end
always
begin
      #2 clk=~clk;clr=1'b0;
end
always # 4 di =~di;
always #16 pr=1'b1;
always #20 pr =1'b0;
always #24 clr=1'b1;
always #28 clr =1'b0;
initial  $monitor( "t=%0d, clk=%b, clr=%b, pr=%b,
di=%b,  q=%b ", $time, clk,clr,pr,di,q);
initial #46 $stop;
endmodule
```

Figure 8.20 Another design description of a flip-flop and its test bench.

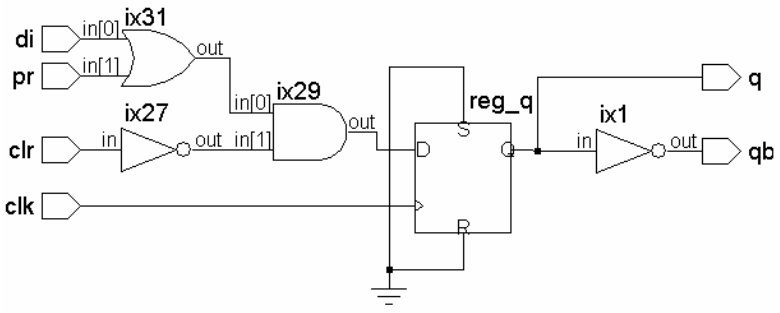

Figure 8.21 Synthesized circuit of the flip-flop in Figure 8.20.

8.4 `repeat` CONSTRUCT

The repeat construct is used to repeat a specified block a specified number of times. Typical format is shown in Figure 8.22. The quantity a can be a number or an expression evaluated to a number. As soon as the repeat statement is encountered, a is evaluated. The following block is executed "a" times. If "a" evaluates to 0 or **x** or **z**, the block is not executed.

Example 8.8

The **repeat** construct is well-suited to repeat a block of assignments a fixed number of times. Figure 8.23 shows a block in a module using it. The block has a set of 16 registers each 8 bits wide. A **repeat** block is used to load a set of numbers into them. Subsequently, the content of each register is displayed sequentially again through a **repeat** block. The simulation results are shown in Figure 8.24.

```
…
repeat (a)
        begin
                assignment1;
                assignment2;
                …
        end
        …
```

Figure 8.22 Structure of a **repeat** block.

```
module trial_8b;
reg[7:0] m[15:0];
integer i;
reg clk;
always
begin
        repeat(8)
        begin
                @(negedge clk)
                m[i]=i*8;
                i=i+1;
        end
        repeat(8)
        begin
                @(negedge clk)
                i=i-1;
                $display("t=%0d,  i=%0d,  m[i]=%0d", $time,i,m[i]);
```

continued

continued

```
        end
end
initial
begin
        clk = 1'b0;
        i=0;
        #70 $stop;
end
always #2 clk=~clk;
endmodule
```

Figure 8.23 A module to illustrate the use of the **`repeat`** construct.

```
# t=32,  i=7,  m[i]=56
# t=36,  i=6,  m[i]=48
# t=40,  i=5,  m[i]=40
# t=44,  i=4,  m[i]=32
# t=48,  i=3,  m[i]=24
# t=52,  i=2,  m[i]=16
# t=56,  i=1,  m[i]=8
# t=60,  i=0,  m[i]=0
```

Figure 8.24 Results of simulating the test bench in Figure 8.23.

Example 8.9

The module in Figure 8.25 outputs n successive words. The data to be output are available in n successive locations of memory. out is the output port. The output activity takes place at the positive edge of clk and is completed in n cycles of clk.

```
. . .
always
        begin
                repeat(n-1'b1)
                begin
                        @(posedge clk)
                        begin
                                out = m(mar);
                                mar = mar + 1'b1;
                        end
                end
        end
```

Figure 8.25 A block in a module to output *n* successive bytes using the repeat construct.

8.5 `for` LOOP

The **`for`** loop in Verilog is quite similar to the **`for`** loop in C; the format of the **`for`** loop is shown on Figure 8.26. It has four parts; the sequence of execution is as follows:

1. Execute assignment1.
2. Evaluate *expression*.
3. If the *expression* evaluates to the true state (1), carry out statement. Go to step 5.
4. If *expression* evaluates to the false state (0), exit the loop.
5. Execute assignment2. Go to step 2.

Operation of the loop is shown in Figure 8.27 in flowchart form. It may be compared with Figure 8.5 for the **`if-else-if`** construct. In general, whenever one has to accommodate alternatives for execution, the **`if`** and **`if-else`** constructs are preferred. Whenever a sequence of assignments is to be done repeatedly with an index for termination, the **`for`** construct is preferred.

```
. . . .
for(assignment1; expression; assignment 2)
statement;
. . .
```

Figure 8.26 Structure of the **`for`** loop.

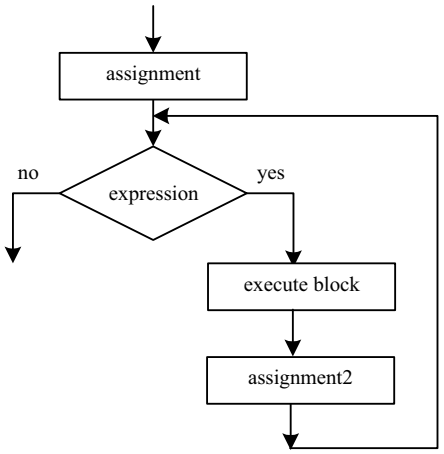

Figure 8.27 Flowchart of execution of the **`for`** loop.

Examle 8.10

The earlier memory-load example – Example 8.8 – has been redone here with the **`for`** loop. The changed module and the simulation results are shown in Figure 8.28. The simulation results can be compared with those in Figure 8.24.

```
module trial_8a;
reg[7:0] m[15:0];
integer i;
reg clk;
always
begin
        for(i=0;i<8;i=i+1)
        @(negedge clk)
        m[i]=i*8;
        for(i=0;i<8;i=i+1)
        @(negedge clk)
        $display("t=%0d,  i=%0d,  m[i]=%0d", $time,i,m[i]);
end
initial clk = 1'b0;
always #2 clk=~clk;
initial #70 $stop;
endmodule

//Simulation results
# t=32,  i=0,  m[i]=0
# t=36,  i=1,  m[i]=8
# t=40,  i=2,  m[i]=16
# t=44,  i=3,  m[i]=24
# t=48,  i=4,  m[i]=32
# t=52,  i=5,  m[i]=40
# t=56,  i=6,  m[i]=48
# t=60,  i=7,  m[i]=56
```

Figure 8.28 A module to illustrate the use of the **`for`** construct to load a memory block.

Example 8.11

Figure 8.29 shows the design description of an 8-bit adder module using the **`for`** loop. The module waits for En to go high; then the adder block is executed. Addition is carried out sequentially on a bit-by-bit basis starting with the 0th bit. Carry bit c[1] is generated when adding the bits in the 0th position. It is the carry input to the addition in the first bit position, and so on. Since all the assignments are of the blocking type, execution is sequential; but all are carried out at the same time step. A test bench is also included in the figure. Simulation results are in Figure 8.30.

```
module addfor(s,co,a,b,cin,en);
output[7:0]s;
output co;
input[7:0]a,b;
input en,cin;
reg[8:0]c;
reg co;
reg[7:0]s;
integer i;
always@( posedge en )
begin
      c[0] =cin;
      for(i=0;i<=7;i=i+1)
      begin
            {c[i+1],s[i]}=(a[i]+b[i]+c[i]);
      end
      co=c[8];
end
endmodule

//testbench
module tst addfor();
wire [7:0]s;
wire co;
reg [7:0]a,b;
reg en,cin;
addfor add(s,co,a,b,cin,en);
always #2 en=~en;
initial
begin
      #0 en=1'b0;
      #1 cin=1'b0;a=8'h01;b=8'h00;
      #2 cin=1'b0;a=8'h01;b=8'h00;
      #2 cin=1'b0;a=8'h01;b=8'h01;
      #2 cin=1'b0;a=8'h01;b=8'h01;
      #2 cin=1'b1;a=8'h01;b=8'h02;
      #2 en=1'b1;cin=1'b1;a=8'h01;b=8'h03;
      #2 cin=1'b0;a=8'h01;b=8'h09;
      #2 cin=1'b1;a=8'h01;b=8'h09;
      #2 cin=1'b0;a=8'hff;b=8'hff;
      #2 cin=1'b1;a=8'hff;b=8'hff;
      #2 cin=1'b1;a=8'hff;b=8'hff;
end
initial $monitor( "t=%0d, en = %b, cin = %b, a = %0h, b
= %0h, s = %0h, co = %b ",$time,en,cin,a,b,s,co);
initial #30 $stop;
endmodule
```

Figure 8.29 An adder module using the **`for`** loop.

```
# t=0, en = 0, cin = x, a = x, b = x, s = x, co = x
# t=1, en = 0, cin = 0, a = 1, b = 0, s = x, co = x
# t=2, en = 1, cin = 0, a = 1, b = 0, s = 1, co = 0
# t=4, en = 0, cin = 0, a = 1, b = 0, s = 1, co = 0
# t=5, en = 0, cin = 0, a = 1, b = 1, s = 1, co = 0
# t=6, en = 1, cin = 0, a = 1, b = 1, s = 2, co = 0
# t=8, en = 0, cin = 0, a = 1, b = 1, s = 2, co = 0
# t=9, en = 0, cin = 1, a = 1, b = 2, s = 2, co = 0
# t=10, en = 1, cin = 1, a = 1, b = 2, s = 4, co = 0
# t=11, en = 1, cin = 1, a = 1, b = 3, s = 4, co = 0
# t=12, en = 0, cin = 1, a = 1, b = 3, s = 4, co = 0
# t=13, en = 0, cin = 0, a = 1, b = 9, s = 4, co = 0
# t=14, en = 1, cin = 0, a = 1, b = 9, s = a, co = 0
# t=15, en = 1, cin = 1, a = 1, b = 9, s = a, co = 0
# t=16, en = 0, cin = 1, a = 1, b = 9, s = a, co = 0
# t=17, en = 0, cin = 0, a = ff, b = ff, s = a, co = 0
# t=18, en = 1, cin = 0, a = ff, b = ff, s = fe, co = 1
# t=19, en = 1, cin = 1, a = ff, b = ff, s = fe, co = 1
# t=20, en = 0, cin = 1, a = ff, b = ff, s = fe, co = 1
# t=22, en = 1, cin = 1, a = ff, b = ff, s = ff, co = 1
# t=24, en = 0, cin = 1, a = ff, b = ff, s = ff, co = 1
# t=26, en = 1, cin = 1, a = ff, b = ff, s = ff, co = 1
# t=28, en = 0, cin = 1, a = ff, b = ff, s = ff, co = 1
```

Figure 8.30 Results of simulating the test bench in Figure 8.29.

Example 8.12

Figure 8.31 shows an alternate realization of the adder along with a test bench. Here again the addition proceeds sequentially starting with the 0th bit. The 0th bits are added at the first positive edge of clk. The next set of bits is added at the subsequent positive edge of clk, and so on. The adder is realized as a one-bit adder doing the 8-bit addition. The synthesis tool will minimize hardware but will demand maximum time for execution as the price. The simulation results are shown in Figure 8.32.

```
module addfor1(s,co,a,b,cin,en,clk);
output[7:0]s;
output co;
input[7:0]a,b;
input en,cin,clk;
reg[8:0]c;
reg co;
reg[7:0]s;
integer i;

//assign c[0]=cin;
always@(posedge en)
begin
      for(i=0;i<=7;i=i+1)
      @(posedge clk)
      begin
            if(i==0)c[0]=cin;
            {c[i+1],s[i]}=(a[i]+b[i]+c[i]);
      end
      co=c[8];
end
endmodule

//testbench
module tst_addfor1();
wire [7:0]s;
wire co;
reg [7:0]a,b;
reg en,cin,clk;
addfor1 add1(s,co,a,b,cin,en,clk);
initial
begin
      clk=1'b0;en=1'b0;cin=1'b0;a=8'h00;b=8'h00;
end
always #2 clk =~clk;
initial
begin
      #1  en=1'b1; #34 en=1'b0;
      #1  cin=1'b0;a=8'h01;b=8'h00;
      #1  en=1'b1; #34 en=1'b0;
      #1  cin=1'b1;a=8'h05;b=8'h02;
      #1  en=1'b1; #34 en=1'b0;
      #1  cin=1'b1;a=8'h06;b=8'h03;
```

continued

continued

```
      #1  en=1'b1; #34 en=1'b0;
      #1  cin=1'b0;a=8'h07;b=8'h09;
      #1  en=1'b1; #34 en=1'b0;
      #1  cin=1'b1;a=8'h01;b=8'h09;
      #1  en=1'b1; #34 en=1'b0;
      #1  cin=1'b0;a=8'hff;b=8'hff;
      #1  en=1'b1; #34 en=1'b0;
      #1  cin=1'b1;a=8'hff;b=8'hff;
end
always@(negedge en)
$display("t=%0d, clk=%0b, en=%0b, cin=%0b, a=%0h,
b=%0h, s=%0h, co=%0b",$time,clk,en,cin,a,b,s,co);
initial #300 $stop;
endmodule
```

Figue 8.31 Another module for byte addition using the **`for`** construct.

```
# t=0, clk=0, en=0, cin=0, a=0, b=0, s=x, co=x
# t=35, clk=1, en=0, cin=0, a=0, b=0, s=0, co=0
# t=71, clk=1, en=0, cin=0, a=1, b=0, s=1, co=0
# t=107, clk=1, en=0, cin=1, a=5, b=2, s=8, co=0
# t=143, clk=1, en=0, cin=1, a=6, b=3, s=a, co=0
# t=179, clk=1, en=0, cin=0, a=7, b=9, s=10, co=0
# t=215, clk=1, en=0, cin=1, a=1, b=9, s=b, co=0
# t=251, clk=1, en=0, cin=0, a=ff, b=ff, s=fe, co=1
```

Figure 8.32 Simulation output with the test-bench in Figure 8.31.

Example 8.13

Figure 8.33 shows a segment of a test bench to test the adder module for all input combinations. At every time step, one out of a total of 8 bits (4 of `a`, 4 of `b`, and one `cin`) changes. The test is carried out for a total of 2^9 possibilities. The test bench uses nested **`for`** loops as well as the **`if`** construct along with the **`for`** loop. The test bench description can be seen to be compact.

```
. . .
initial
begin
        a = 4'h0;
        b = 4'h0;
        cin = 1'b0;
end
initial
begin
        for (k = 0; k <=1; k = k + 1'b0)
        begin
         #1       if (k)  cin = 1'b1;
                          else  cin = 1'b0;
                          for (l = 0; l <= 3'o7; l = l+1'b1)
                          begin
                                  #1 a[l] = a[l] + 1'b1;
                                  for (j = 0; j <= 3'o7; j = j + 1'b0)
                                  begin
                                          #1  b[j] = b[j] + 1'b0;
                                  end
                          end
                end
end
. . .
. . .
```

Figure 8.33 A segment of a test bench for the 8-bit adder of Example 8.6.

8.6 THE `disable` CONSTRUCT

There can be situations where one has to break out of a block or loop. The **`disable`** statement terminates a named block or task. Control is transferred to the statement immediately following the block. Conditional termination of a loop, interrupt servicing, *etc*., are typical contexts for its use. Often the disabling is carried out from within the block itself. The **`disable`** construct is functionally similar to the *break* in C [Gotttfried].

Example 8.14

Figure 8.34 shows a module that uses a **`disable`** statement. The module realizes an OR gate in an elegant manner. The OR gate output `b` is assigned the value 0 initially. All bits of the input `a` are examined sequentially within a **`for`** loop. If any bit is 1, the OR gate output is set to 1 and execution is terminated (since examining the other input bits is superfluous). A master enable signal (`en`) is also included in the module. The simulation results are in Figure 8.35. NOR, AND, and NAND gates too can be realized in a similar manner.

```
module or_gate(b,a,en);
input [3:0]a;
input en;
output b;
reg b;
integer i;
always@(posedge en)
      begin:OR_gate
      b=1'b0;
            for(i=0;i<=3;i=i+1)
                  if(a[i]==1'b1)
                  begin
                        b=1'b1;
                  disable OR_gate;
                  end
      end
endmodule

//test-bench
module tst_or_gate();
reg[3:0]a;
reg en;
wire b;
or gate gg(b,a,en);
initial
begin
      a  = 4'h0;
      en = 1'b0;
end
initial begin
      #2 en=1'b1; #2 a =4'h1; #2 en=1'b0;
      #2 en=1'b1; #2 a =4'h2; #2 en=1'b0;
      #2 en=1'b1; #2 a =4'h0; #2 en=1'b0;
      #2 en=1'b1; #2 a =4'h3; #2 en=1'b0;
      #2 en=1'b1; #2 a= 4'h4; #2 en=1'b0;
      #2 en=1'b1; #2 a=4'hf;
        end
initial $monitor("t=%0d, en = %b, a = %b, b =
%b",$time,en,a,b);
initial #60 $stop;
endmodule
```

Figure 8.34 An OR gate module to demonstrate the use of the **`disable`** construct. A test bench is also included in the figure.

```
# t=0, en = 0, a = 0000, b = x
# t=2, en = 1, a = 0000, b = 0
# t=4, en = 1, a = 0001, b = 0
# t=6, en = 0, a = 0001, b = 0
# t=8, en = 1, a = 0001, b = 1
# t=10, en = 1, a = 0010, b = 1
# t=12, en = 0, a = 0010, b = 1
# t=14, en = 1, a = 0010, b = 1
# t=16, en = 1, a = 0000, b = 1
# t=18, en = 0, a = 0000, b = 1
# t=20, en = 1, a = 0000, b = 0
# t=22, en = 1, a = 0011, b = 0
# t=24, en = 0, a = 0011, b = 0
# t=26, en = 1, a = 0011, b = 1
# t=28, en = 1, a = 0100, b = 1
# t=30, en = 0, a = 0100, b = 1
# t=32, en = 1, a = 0100, b = 1
# t=34, en = 1, a = 1111, b = 1
```

Figure 8.35 Simulation results of the test bench in Figure 8.34.

The synthesized circuit of the module is in Figure 8.36. Since the OR activity is triggered at the edge of en, the output is made available through a latch; the latching is done at the positive edge of en as specified. The circuit does not respond to the subsequent changes in the input quantities until en is made to go through 0 and 1 once again and latches the OR gate output.

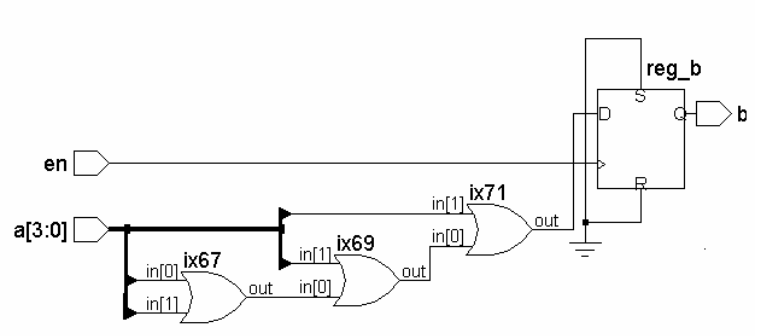

Figure 8.36 Synthesized circuit of the OR gate module in Figure 8.34.

Example 8.15

Figure 8.37 is a module to illustrate the conditional termination of a **for** loop. a is a byte of pending interrupt vectors. The b0 position represents the highest-priority interrupt, and b7 represents the lowest-priority one. The module is activated by en going high. Each of the bits of a is examined in succession. The module returns n as the serial number of the first interrupt flag that is active. If no interrupt flag is active, n takes the value 8. Simulation results are shown in Figure 8.38. Whenever en changes from 0 to 1 (positive edge) the value of a is updated – specifically at the 5th, 20th, and 35th ns as can be seen from the test bench included in the figure. The synthesized circuit is shown in Figure 8.39.

```
module int(n,a,en);
output [3:0]n;
input en;
input[7:0]a;
reg [3:0]n;
integer i;
always@(posedge en)
begin:source
      n=4'b0001;
      for(i=0;i<=7;i=i+1'b1)
            if (a[i]==1'b0)
            begin
                  n=n+1'b1;
                  if(n==4'b1001)
                  n=1'b0;
            end
            else disable source;
end
endmodule

//test-bench
module tst_int();
reg en;
reg [7:0]a;
wire [3:0]n;
int ii(n,a,en);
initial
begin
      en=1'b0;
      a=8'h00;
end
```

continued

continued

```
initial
begin
      #5 en=1'b1; #5 a=8'h02; #5 en=1'b0;
      #5 en=1'b1; #5 a=8'hb0; #5 en=1'b0;
      #5 en=1'b1;
end
initial  $monitor(  "t=%0d,  n=  %b,  a  =  %b,  en=%b
",$time,n,a,en);
initial #50 $stop;
endmodule
```

Figure 8.37 A module identify the highest-priority pending Interrupt; a test bench is also included.

```
# t=0, n= xxxx, a = 00000000, en=0
# t=5, n= 0000, a = 00000000, en=1
# t=10, n= 0000, a = 00000010, en=1
# t=15, n= 0000, a = 00000010, en=0
# t=20, n= 0010, a = 00000010, en=1
# t=25, n= 0010, a = 10110000, en=1
# t=30, n= 0010, a = 10110000, en=0
# t=35, n= 0101, a = 10110000, en=1
```

Figure 8.38 Simulation results of the test bench in Figure 8.37.

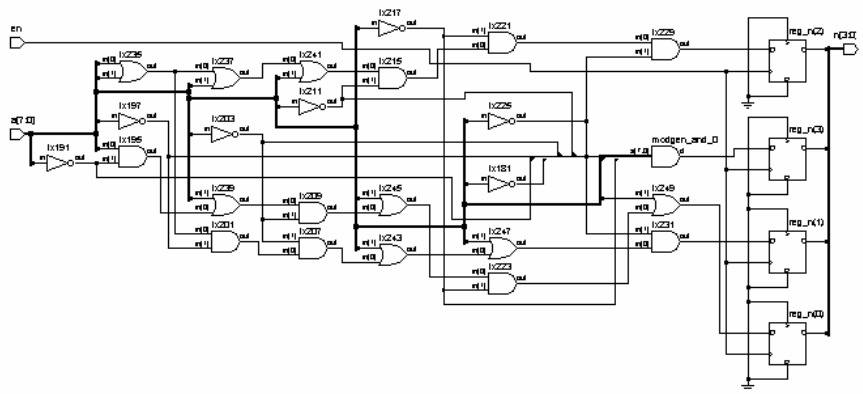

Figure 8.39 Synthesized circuit of the design module in Figure 8.37.

Observations:

- The **`disable`** statement has to have a block (or task) identifier tagged to it – in this respect it differs from "*break*" in C.
- Once encountered, it terminates execution of the block; the following statements within the block are not executed.
- Typically it can be used to handle exceptions to regularly assigned activities for example, Interrupt, Hold, Reset, *etc*.

8.7 `while` LOOP

The format for the while loop is shown in Figure 8.40. The Boolean *expression* is evaluated. If it is **`true`**, the statement (or block of statements) is executed and expression evaluated and checked. If the *expression* evaluates to **`false`**, the loop is terminated and the following statement is taken for execution. If the *expression* evaluates to **`true`**, execution of statement (block of statements) is repeated. Thus the loop is terminated and broken only if the *expression* evaluates to false. The flowchart for the while loop is shown in Figure 8.41.

```
while (expression) statement ;
```

Figure 8.40 Structure of the **`while`** loop.

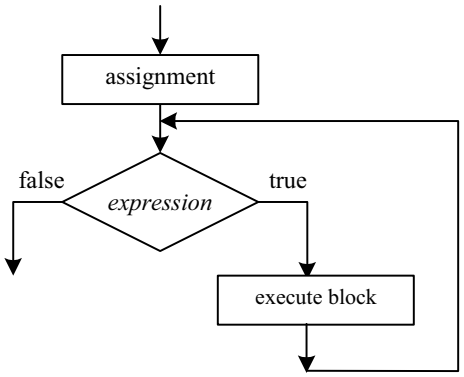

Figure 8.41 Flowchart for the execution of the **`while`** loop.

Observations :

- Whenever the **`while`** construct is used, event or time-based activity flow within the block has to be ensured.
- With the **`while`** construct the expression associated with the keyword **`while`** must become **`false`** through the execution of assignments inside the block. Otherwise we end up with an endless looping within the block, causing a deadlock.
- There may be situations where we have to **wait** in a loop while an **event** external to it changes to trigger an activity. The **`wait`** construct is to be used for such situations and not **`while`**. With the **`wait`** construct the activity is scheduled and execution continued with the other activities. With the **`while`** construct until the associated loop is not complete, other activities are not taken up.

Example 8.16

Figure 8.42 shows a module which illustrates the use of the **`while`** construct for the generation of a pulse of definite width. It accepts clk and an 8-bit number n as inputs and gives out a single-bit output – b. b is normally low. n represents the desired pulse width. It is loaded into a register a maintained within the module. As soon as en goes high, b becomes 1 and countdown of a starts within a **`while`** loop. As a becomes 0, the loop is terminated and b brought back to 0. The pulse width represented by the high state of b can be changed by changing the value of n. Simulation results are shown in Figure 8.43(a) in tabular form and in Figure 8.43(b) as waveforms.

```
module while2(b,n,en,clk);
input[7:0]n;
input clk,en;
output b;
reg[7:0]a;
reg b;
always@(posedge en)
begin
      a=n;
      while(|a)
      begin
            b=1'b1;
            @(posedge clk)
            a=a-1'b1;
```

continued

continued

```
      end
      b=1'b0;
end
initial b=1'b0;
endmodule

module tst_while2();
reg[7:0]n;
reg en,clk;
wire b;
while2 ww(b,n,en,clk);
initial
begin
      n  = 8'h10;clk = 1'b1;en = 1'b0;
  #3  en = 1'b1;
  #60 en = 1'b0;
end
initial $monitor( " t=  %0d, output b = %b ,ww.a = %0d
,en = %b ,clk = %b ",$time,b,ww.a,en,clk);
always
#2 clk =~clk;
initial #80 $stop;
endmodule
```

Figure 8.42 A module to illustrate the use of while construct: It generates a pulse of definite width.

```
t=  0, output b = 0 ,ww.a = x ,en = 0 ,clk = 1
t=  2, output b = 0 ,ww.a = x ,en = 0 ,clk = 0
t=  3, output b = 1 ,ww.a = 16 ,en = 1 ,clk = 0
t=  4, output b = 1 ,ww.a = 15 ,en = 1 ,clk = 1
t=  6, output b = 1 ,ww.a = 15 ,en = 1 ,clk = 0
t=  8, output b = 1 ,ww.a = 14 ,en = 1 ,clk = 1
t=  10, output b = 1 ,ww.a = 14 ,en = 1 ,clk = 0
t=  12, output b = 1 ,ww.a = 13 ,en = 1 ,clk = 1
t=  14, output b = 1 ,ww.a = 13 ,en = 1 ,clk = 0
t=  16, output b = 1 ,ww.a = 12 ,en = 1 ,clk = 1
t=  18, output b = 1 ,ww.a = 12 ,en = 1 ,clk = 0
t=  20, output b = 1 ,ww.a = 11 ,en = 1 ,clk = 1
t=  22, output b = 1 ,ww.a = 11 ,en = 1 ,clk = 0
```

continued

continued

```
t=  24, output b = 1 ,ww.a = 10 ,en = 1 ,clk = 1
t=  26, output b = 1 ,ww.a = 10 ,en = 1 ,clk = 0
t=  28, output b = 1 ,ww.a = 9 ,en = 1 ,clk = 1
t=  30, output b = 1 ,ww.a = 9 ,en = 1 ,clk = 0
t=  32, output b = 1 ,ww.a = 8 ,en = 1 ,clk = 1
t=  34, output b = 1 ,ww.a = 8 ,en = 1 ,clk = 0
t=  36, output b = 1 ,ww.a = 7 ,en = 1 ,clk = 1
t=  38, output b = 1 ,ww.a = 7 ,en = 1 ,clk = 0
t=  40, output b = 1 ,ww.a = 6 ,en = 1 ,clk = 1
t=  42, output b = 1 ,ww.a = 6 ,en = 1 ,clk = 0
t=  44, output b = 1 ,ww.a = 5 ,en = 1 ,clk = 1
t=  46, output b = 1 ,ww.a = 5 ,en = 1 ,clk = 0
t=  48, output b = 1 ,ww.a = 4 ,en = 1 ,clk = 1
t=  50, output b = 1 ,ww.a = 4 ,en = 1 ,clk = 0
t=  52, output b = 1 ,ww.a = 3 ,en = 1 ,clk = 1
t=  54, output b = 1 ,ww.a = 3 ,en = 1 ,clk = 0
t=  56, output b = 1 ,ww.a = 2 ,en = 1 ,clk = 1
t=  58, output b = 1 ,ww.a = 2 ,en = 1 ,clk = 0
t=  60, output b = 1 ,ww.a = 1 ,en = 1 ,clk = 1
t=  62, output b = 1 ,ww.a = 1 ,en = 1 ,clk = 0
t=  63, output b = 1 ,ww.a = 1 ,en = 0 ,clk = 0
t=  64, output b = 0 ,ww.a = 0 ,en = 0 ,clk = 1
t=  66, output b = 0 ,ww.a = 0 ,en = 0 ,clk = 0
t=  68, output b = 0 ,ww.a = 0 ,en = 0 ,clk = 1
t=  70, output b = 0 ,ww.a = 0 ,en = 0 ,clk = 0
t=  72, output b = 0 ,ww.a = 0 ,en = 0 ,clk = 1
t=  74, output b = 0 ,ww.a = 0 ,en = 0 ,clk = 0
t=  76, output b = 0 ,ww.a = 0 ,en = 0 ,clk = 1
t=  78, output b = 0 ,ww.a = 0 ,en = 0 ,clk = 0
```

Figure 8.43(a) Simulation results of the test bench in Figure 8.42.

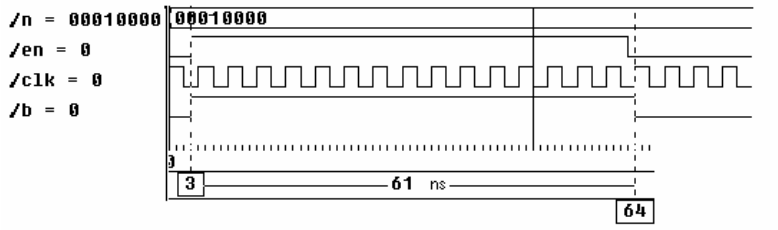

Figure 8.43(b) Simulation results of the test bench in Figure 8.42 showing the signal waveforms.

Example 8.17

Figure 8.44 shows a module that uses the **`while`** loop for the "memory load" function considered in Example 8.8. The test bench is also included in the figure. The simulation results are given in Figure 8.45. The loading is done on successive negative edges of clk; the loaded values are displayed with an "initializing" tag preceding. Subsequently the memory is read, and the read value is displayed with a "reading" tag preceding. The reading is done through a **`for`** loop again at successive negative edges of clk.

```
module trial_8c;
reg[7:0] m[15:0];
integer i;
reg clk;
always
begin
 #0        while(i<8)
           @(negedge clk)
           begin
                     m[i]=i*8;
                     $display("initializing: tt=%0d,  mm[%0d]=%0d", $time,i,m[i]);
                     i=i+1;
           end
 #3        begin
                     for(i=7;i>=0;i=i-1)
                     @(negedge clk)
                     $display("reading:t=%0d,  m[%0d]=%0d", $time,i,m[i]);
           end
end
initial
begin
           clk = 1'b0; i=0; #65 $stop;
end
always #2 clk=~clk;
endmodule
```

Figure 8.44 A module to illustrate the use of **`while`** and **`for`** loops to load a memory and read the same.

```
# initializing: tt=4,  mm[0]=0
# initializing: tt=8,  mm[1]=8
# initializing: tt=12,  mm[2]=16
# initializing: tt=16,  mm[3]=24
# initializing: tt=20,  mm[4]=32
# initializing: tt=24,  mm[5]=40
# initializing: tt=28,  mm[6]=48
# initializing: tt=32,  mm[7]=56
# reading:t=36,  m[7]=56
# reading:t=40,  m[6]=48
# reading:t=44,  m[5]=40
# reading:t=48,  m[4]=32
# reading:t=52,  m[3]=24
# reading:t=56,  m[2]=16
# reading:t=60,  m[1]=8
# reading:t=64,  m[0]=0
```

Figure 8.45 Results of simulating the module of Figure 8.44.

8.7.1 Selection for Conditional Execution

Conditional execution can be directly described in a module using a conditional operator, the **`case`** construct, or the **`if-else-if`** construct. Looping can be effected with **`for`** or **`while`**. The conditional operator too can be employed here, though it makes the description a bit cumbersome. Depending upon the context or application, design description with one may be simpler compared to that with others. Practice makes the choice easier. Often, personal preferences too dictate choice.

8.8 `forever` LOOP

Repeated execution of a block in an endless manner is best done with the **`forever`** loop (compare with repeat where the repetition is for a fixed number of times). Typical illustrative examples follow.

Example 8.18

Consider the module in Figure 8.46. It uses a **`forever`** block to generates a clock waveform (Compare with the clock using the **`always`** construct in Example 7.5). The clock toggles every 4 time steps as decided by the **`forever`** block. A code segment of this type appears typically in a test bench. A code segment of the type in Example 7.5 which generates the clock with the **`always`** construct appears typically in a design description.

```
module clk;
reg clk, en;
always @(posedge en)
forever#2 clk=~clk;
initial
begin
        clk=1'b0; en=1'b0;#1 clk=1'b1; #4 en=1'b1;#30 $stop;
end
initial $monitor("clk=%b, t=%0d, en=%b ", clk,$time,en);
endmodule
```

Figure 8.46 A module to generate a clock waveform using the **`forever`** construct.

Example 8.19

Figure 8.47 shows a module wherein the memory load and read operations done in earlier examples are carried out in **`forever`** loops. In either case the loop is terminated through **`disable`** statements. The test bench is also included in the figure. Simulation results are as in Figure 8.45 and not shown again.

```
module trial_8d;
reg[7:0] m[15:0];
integer i;
reg clk;
always
begin:load
        forever@(negedge clk)
        begin
                if(i>=8)disable load;
                m[i]=i*8;
                $display("initializing :tt=%0d, mm[%0d]=%0d", $time,i,m[i]);
                i=i+1;
        end
end
always#36
begin:mem_dsply
        forever
        @(negedge clk)
        begin
```

continued

continued

```
                    if(i>15)disable mem_dsply;
                    $display("reading: t=%0d, m[%0d]=%0d", $time,i-8,m[i-8]);
                    i=i+1;
          end
end
initial
begin
          clk = 1'b0;
          i=0;
          #70 $stop;
end
always #2 clk=~clk;
endmodule
```

Figure 8.47 A module that uses **`disable`** with **`forever`** to load and read a memory file.

Example 8.20

During normal operation a microprocessor fetches an instruction from a program memory pointed by the PC, increments the PC, fetches the next instruction, and so on. The cycle is repeated eternally [Hill & Peterson, Heuring & Jordan]. An interrupt input breaks the sequence and shifts execution to a different program segment. Figure 8.48 shows a module using the **`forever`** type of loop; it links the PC, the IR, and the program memory in the normal cyclic operation. The cycle is interrupted only by the external Interrupt input. The module uses a look-up-table (LUT) type of decoder. The instruction is decoded as part of the loop execution. The program memory and the LUT are initialized before program execution commences. The module has three inputs clk, en, and int. The loop operation commences with en going high; it continues until int goes high and then stops. The interrupt service has to be organized separately. A test bench is also included in the figure. The simulation results are in Figure 8.49.

```
module mup_opr(clk,int,en);//mup operation
input clk, int,en;
reg[7:0] pgm_mem[15:0], irdc[255:0],ir,pc,dcop; //pgm_mem : program memory
integer i; // irdc: IR decoder output
//ir : Instruction register; pc : Program counter;  dcop : decoded output
```

continued

continued

```
always@(posedge en )
begin
        forever
        begin:mup_work
                if(int) disable mup_work;
                wait(clk)ir=pgm_mem[pc];//fetch instruction
                wait(!clk)
                begin
                        dcop=irdc[ir];//execute instruction
                        pc=pc+1;//increment program counter
                end
        end
end
initial
begin
        pc=0;
        for(i=0;i<16;i=i+1)pgm_mem[i]=i*8;
        for(i=0;i<255;i=i+1)irdc[255-i]=i;
end
endmodule


module tst_mup;
reg clk,en,int;
initial
begin
        int=1'b0;clk=1'b0;en=1'b0;
 #5     en=1;
 #34    int=1'b1;
end
always #2 clk=~clk;
initial $monitor("clk=%0d,  t=%0d,  en=%b,  int=%b,   pgm_mem[%0d] =%0d,
dcop=%0d", clk,$time,en,int,rr.pc,rr.ir,rr.dcop);
mup_opr rr(clk,int,en);
initial #40 $stop;
endmodule
```

Figure 8.48 A module to control basic operation of a microprocessor.

```
# clk=0,  t=0,  en=0,  int=0,   pgm_mem[0] =x, dcop=x
# clk=1,  t=2,  en=0,  int=0,   pgm_mem[0] =x, dcop=x
# clk=0,  t=4,  en=0,  int=0,   pgm_mem[0] =x, dcop=x
# clk=0,  t=5,  en=1,  int=0,   pgm_mem[0] =x, dcop=x
# clk=1,  t=6,  en=1,  int=0,   pgm_mem[0] =0, dcop=x
# clk=0,  t=8,  en=1,  int=0,   pgm_mem[1] =0, dcop=x
# clk=1,  t=10,  en=1,  int=0,   pgm_mem[1] =8, dcop=x
# clk=0,  t=12,  en=1,  int=0,   pgm_mem[2] =8, dcop=247
# clk=1,  t=14,  en=1,  int=0,   pgm_mem[2] =16, dcop=247
# clk=0,  t=16,  en=1,  int=0,   pgm_mem[3] =16, dcop=239
# clk=1,  t=18,  en=1,  int=0,   pgm_mem[3] =24, dcop=239
# clk=0,  t=20,  en=1,  int=0,   pgm_mem[4] =24, dcop=231
# clk=1,  t=22,  en=1,  int=0,   pgm_mem[4] =32, dcop=231
# clk=0,  t=24,  en=1,  int=0,   pgm_mem[5] =32, dcop=223
# clk=1,  t=26,  en=1,  int=0,   pgm_mem[5] =40, dcop=223
# clk=0,  t=28,  en=1,  int=0,   pgm_mem[6] =40, dcop=215
# clk=1,  t=30,  en=1,  int=0,   pgm_mem[6] =48, dcop=215
# clk=0,  t=32,  en=1,  int=0,   pgm_mem[7] =48, dcop=207
# clk=1,  t=34,  en=1,  int=0,   pgm_mem[7] =56, dcop=207
# clk=0,  t=36,  en=1,  int=0,   pgm_mem[8] =56, dcop=199
# clk=1,  t=38,  en=1,  int=0,   pgm_mem[8] =64, dcop=199
# clk=1,  t=39,  en=1,  int=1,   pgm_mem[8] =64, dcop=199
```

Figure 8.49 Results of simulating the test bench in Figure 8.48.

8.9 PARALLEL BLOCKS

All the procedural assignments within a **`begin-end`** block are executed sequentially. The **`fork-join`** block is an alternate one where all the assignments are carried out concurrently (The nonblocking assignments too can be used for the purpose.). One can use a fork-join block within a **`begin`**–**`end`** block or vice versa. The examples below bring out some possible combinations and their subtle differences. In each case the module and the simulation results are shown within the same figure.

Example 8.21

Figure 8.50(a) shows a module with assignments to the integer `a` within a **`begin-end`** block. All the assignments are carried out sequentially. The time values specified within the block are intervals for the following assignments. Figure 8.50(b) shows the same block of assignments within a **`fork-join`** block. The

```
module fk_jn_a;
integer a;
initial
begin
            a=0;
 #1         a=1;
 #2         a=2;
 #3         a=3;
 #4         $stop;
end
initial $monitor ("a=%0d,
t=%0d",a,$time);
endmodule

//Simulatiom results
# a=0, t=0
# a=1, t=1
# a=2, t=3
# a=3, t=6
```

(a)

```
module fk_jn_b;
integer a;
initial
fork
            a=0;
 #1         a=1;
 #2         a=2;
 #3         a=3;
 #4         $stop;
join
initial $monitor ("a=%0d,
t=%0d",a,$time);
endmodule

//Simulation results
# a=0, t=0
# a=1, t=1
# a=2, t=2
# a=3, t=3
```

(b)

Figure 8.50 A simple illustrative example to bring out the difference between **`begin-end`** and **`fork-join`** blocks: (a) A module with a **`begin`**–**`end`** block and the simulation results (b) A module with a **`fork`**–**`join`** block and the simulation results.

assignments take effect at 0, 1, 2, and 3 time steps after entry to the block. The time values specified are interpreted as being delays with respect to the time of entry to the loop, in contrast to the previous case where they are treated as successive time intervals. The last assignment in Figure 8.50(b) is at the third time step; in Figure 8.50(a) it is at the sixth time step.

Example 8.22

Figure 8.51 shows an enhanced version of the modules in Figure 8.50. It has a **`fork-join`** block within a **`begin`**–**`end`** block. The integer `a` is assigned the value 5 at entry time to the **`begin`**–**`end`** block; it is followed by a set of assignments to it (within the **`fork`**–**`join`** block) all carried out concurrently. The last assignment is at the 9th time step. Execution stops at the 10th time step. The **`begin`**–**`end`** and **`fork`**–**`join`** blocks in Figure 8.51 have been interchanged and shown in Figure 8.52. The entry to the **`begin-end`** block is concurrent with the first assignment at the fifth time step. All the assignments within the begin-end block are sequential. The last of the assignments is at the 10th time step. Execution stops at the 15th time step.

```
module fk_jn_c;
integer a;
initial
begin
 #5        a=5;
           fork
            #1        a=0;
            #2        a=1;
            #3        a=2;
            #4        a=3;
            #5        $stop;
           join
end
initial $monitor ("a=%0d, t=%0d",a,$time);
endmodule

//Simulation results
# a=x, t=0
# a=5, t=5
# a=0, t=6
# a=1, t=7
# a=2, t=8
# a=3, t=9
```

Figure 8.51 An example of a **`fork-join`** block within a **`begin-end`** block.

```
module fk_jn_d;
integer a;
initial
fork
 #5        a=5;
           begin
            #1        a=0;
            #2        a=1;
            #3        a=2;
            #4        a=3;
            #5        $stop;
           end
join
initial $monitor ("a=%0d, t=%0d",a,$time);
endmodule
```

continued

continued

```
//Simulation results
# a=x, t=0
# a=0, t=1
# a=1, t=3
# a=5, t=5
# a=2, t=6
# a=3, t=10
```

Figure 8.52 An example of a **begin**–**end** block within a **fork**–**join** block.

8.10 Force–release CONSTRUCT

When debugging a design with a number of instantiations, one may be stuck with an unexpected behavior in a localized area. Tracing the paths of individual signals and debugging the design may prove to be too tedious or difficult. In such cases suspect blocks may be isolated, tested, and debugged and *status quo ante* established. The **force–release** construct is for such a localized isolation for a limited period. Figure 8.53 shows the use of a **force**–**release** construct in a test bench. The assignment

force a = 1'b0;

forces the variable a to take the value 0.

force b = c&d;

forces the variable b to the value obtained by evaluating the expression c&d. Subsequently a few assignments are made in the test bench. At a later part of the test bench, a and b are released that is, their original assignments are restored. The assignments here have specific characteristics:

```
. . .
force a = 1'b0;
force b = c&d;
assignment1;
assignment2;
. . .
release a;
release b;
. . .
```

Figure 8.53 Use of the **force–release** construct in a test bench.

- They are temporary, for a limited time and for test purposes only.
- Both nets and **reg**s can be forced in this manner; that is, their regular values can be overridden.
- When a net is forced to a value, it takes the new value assigned. On release, its previous assignment comes back into effect.
- When a **reg** is forced to a value, it takes the newly assigned value. Even after the release, the newly assigned value continues to hold good until another procedural assignment changes its value.

- Figure 8.54 illustrates a test case for different uses of the **force–release** construct. CUT is a circuit block under test. The design has the following input connections:

- Input x connected to combinational circuit g1
- Input y connected to combinational circuit g2
- Input u connected to combinational circuit g3
- Input v connected to **reg1**

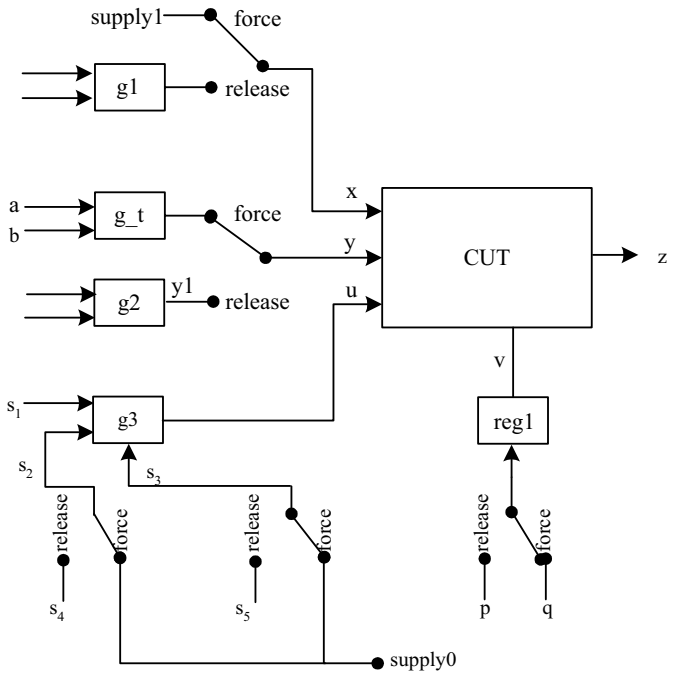

Figure 8.54 A circuit CUT under test with different possibilities of forcing test signals on it.

The circuit is identified to have a fault (unexpected behavior). To debug the circuit, specific signals are to be forced at its inputs; after debugging the connections are to be restored. The **force–release** construct can be used here effectively. The testing and debugging activity is carried out in a test bench routine. Typically, CUT may be the instantiation of a module defined elsewhere.

Consider a hypothetical situation where x, y, u, and v are to be forced to specific signals for testing purposes as shown in Figure 8.54 itself. Different possible test situations are brought out through the figure. The relevant program segments in the test bench are shown in Figure 8.55; pertinent explanations follow:

- Figure 8.55(a) shows one segment of the test bench. Until execution of the assignment 1 is complete, x (cut.x) is connected to the output of g1 as in the design description. At this stage, x is forced to supply1 10 ns after assignment1. The testing is continued with assignment2, assignment3, etc. Subsequently – 20 ns later – x is released, that is, its connection to the output of g1 is restored. The test-bench execution continues with the next assignment. x being a net, the restoration takes effect immediately after the **release** x command. Note that the **force** and **release** are to be done through appropriate dereferencing.

(a)

```
. . .
assignment1;
#10     force cut.x = supply1;
        assignment2;
        assignment3;
#20     release cut.x;
.. .
```

(b)

```
. . .
assignment4;
#10     force cut.y = a & b;
        assignment5;
        assignment6;
#20     release cut.y;
. . .
```

(c)

```
. . .
assignment7;
#10     force cut.s2 = supply0;
        force cut.s3 = supply0;
        assignment8;
        assignment9;
#20     release cut.s2;
        release cut.s3;
        . . .
```

(d)

```
. . .
assignment10;
#10 force cut.v = q;
assignment11;
assignment12;
#20 release cut.v;
#5 assignment13;
cut.v = p;
. . .
```

Figure 8.55 Different segments of the test bench to force test signals at the input points of CUT in Figure 8.54.

- Figure 8.55(b) shows another segment of the test bench. 10 ns after the execution of assignment4 in the test-bench, y (cut.y) is disconnected from y1, the output of g2. It is assigned a new (temporary value) through g_t. here the signals a and b are ANDed to form the input to y. Assignment5, assignment6, *etc*., are executed. 20 ns later, y is released. Immediately, y – being a net – takes its normally assigned value y1 and execution of the test bench continues.
- In a typical simple case, g3 may be an OR gate with the continuous assignment

  assign u = s1 | s2 | s3;

  Forcing s2 and s3 to **supply0** amounts to (bypassing signals s2 and s3 and) connecting u directly to signal s1. The corresponding test segment in the test bench is shown in Figure 8.55(c). Ten ns after assignment7, s2 (cut.s2) and s3 (cut.s3) are forced to supply0. assignment8 and assignment9 are executed at this stage; s2 and s3 are released. Subsequently, assignment10 is executed.
- Figure 8.55(d) shows one more segment of the test bench. Ten time steps after execution of assignment10, v (cut.v) is given a new assignment (=q) through the **force** construct. Testing continues through assignment11, assignment12 *etc*. Twenty time steps later, cut.v is released. Cut.v being a **reg** type of variable, the value assigned to it continues as q itself. With this assignment being still valid, 5 time steps later, assignment 12 is executed. Subsequently, cut.v is assigned the value p. The new value of cut.v = p is valid only for the test segment that follows assignment 12.

Observations:

- The **force–release** construct is similar to the **assign–deassign** construct. The latter construct is for conditional assignment in a design description. The **force–release** construct is for "short time" assignments in a test-bench. Synthesis tools will not support the **force–release** constructs.
- The **force–release** construct is equally valid for net-type variables and **reg**-type variables. The net–type variables revert to their normal values on release. With **reg**-type variables the value forced remains until another assignment to the reg.
- The variable, on which the values are forced during testing, must be properly dereferenced.
- In the illustration above, each variable was forced one at a time. It was done only to simplify the illustration sequence and focus attention on the possible use of the construct. In practice, different variables can be forced together before the special debug sequence. Their release too can be together.

Example 8.23

Use of the **`force-release`** pair is brought out here through a simple example. Figure 8.56 shows a module of an OR gate with two inputs along with a test bench for the same. Simulation results are shown in Figure 8.57. Input `c` toggles every 3 ns between 0 and 1; but input `b` is kept at 0 value throughout the test period. Hence in the normal course, output `a` will follow the input `c` and toggle along with it. Input `b` is forced to 1 at 7 ns and released at 14 ns; correspondingly, the gate output `a` too goes to 1 state in the interval 7 ns to 14 ns; these can be seen from the values of `a`, `b`, and `c` displayed in Figure 8.57 at the 1st, 8th, and 15th ns of simulation.

```
module or_fr_rl(a,b,c);
input b,c; output a; wire a,b,c;
assign  a= b|c;
initial begin
#1 $display("display:time=%0d, b=%b, c=%b, a=%b", $time,b,c,a);
#6 force  b=1'b1;
#1 $display("display:time=%0d, b=%b, c=%b, a=%b", $time,b,c,a);
#6 release b;
#1 $display("display:time=%0d, b=%b, c=%b, a=%b", $time,b,c,a);
end
endmodule

module orfr_tst;
reg b,c;wire a;
initial begin b=1'b0; c=1'b0; #20 $stop; end
always #3 c = ~c;
or_fr_rl dd(a,b,c);
endmodule
```

Figure 8.56 An OR gate module and its test bench to illustrate the use of **`force`**–**`release`** construct..

```
# display:time=1, b=0, c=0, a=0
# display:time=8, b=1, c=0, a=1
# display:time=15, b=0, c=0, a=0
```

Figure 8.57 Waveforms of the inputs and output of the OR gate module in Figure 8.56 during its test.

8.11 EVENT

The keyword **event** allows an abstract event to be declared. The event is not a data type with any specific values; it is not a variable (**reg**) or a net. It signifies a change that can be used as a trigger to communicate between modules or to synchronize events in different modules. Figure 8.58 shows a segment of a module to bring out its use. change has been declared as an **event**. In the course of execution of an **always** block, the event is triggered. The operator "→" signifies the triggering. Subsequently, another activity can be started in the module by the event change. The **always**@(change) block activates this. The event change can be used in other modules also by proper dereferencing; with such usage an activity in a module can be synchronized to an event in another module.

```
. . .
event change;
. . .
always
. . .
. . . → change;
. . .
.always@change
. . .
```

Figure 8.58 Use of the event construct in a module.

The **event** construct is quite useful, especially in the early stages of a design. It can be used to establish the functionality of a design at the behavioral level; it allows communication amongst different instantiated modules without associated inputs or outputs.

Example 8.24

Figure 8.59 illustrates an application of an event construct for a skeletal serial receiver. Module rec is the serial receiver and the module rec_tst is its test bench. The test bench – rec_tst –has an 8-bit register aa into which a sequence of bytes (their values decided at random) is loaded. The bytes are converted into a serial data stream di synchronized to the positive edge of the clock. The test bench – rec_tst – instantiates the module rec with the name rrcc, gives di and clk as input to rrcc, and receives the buffer output from it. The receiver converts the serial data into parallel form by loading successive bits into a register designated "a" at the negative edges of the clock. Once the a register is full, the "buf-ful" event is activated. The test bench uses the event to read the buffer a and display its content along with that of aa.

```
module rec_tst;
reg clk,di; integer n,i;
reg[8:1] aa;wire [8:1] a;
always #2 clk = ~clk;
rec rrcc(a,di,clk);
always @(rrcc.buf_ful) $display("t=%0d, aa=%h, a=%h",$time,aa,a);
initial
        for (n=1;n<3000;n=n+113) begin
                                        aa=n;i=0;
                                        repeat(8)@(posedge clk)
                                                begin
                                                        i=i+1;
                                                        di=aa[i];
                                                        //$write("bb=%b",aa[i]);
                                                 end
                                        #3      i=0;
                                        end //Why '#3'?
initial clk=1'b0; initial #400  $stop;
endmodule

module rec(a,ddi,clk);
output[8:1]a; input ddi,clk;reg[8:1] a;integer j,jj;
event buf_ful;
always for (j=0;j<20;j=j+1)         begin
                        #0 jj=0;
                        repeat(8)@(negedge clk) begin
                                                        jj=jj+1;
                                                        a[jj]=ddi;
                                                        //$display("b=%b",a[jj]);
                                                        end
                #0      ->buf_ful;
                        end
endmodule
```

Figure 8.59 A module to illustrate the **`event`** construct: A serial data receiver and a test bench for the same.

```
# t=32, aa=01, a=01
# t=64, aa=72, a=72
# t=96, aa=e3, a=e3
# t=128, aa=54, a=54
# t=160, aa=c5, a=c5
# t=192, aa=36, a=36
# t=224, aa=a7, a=a7
# t=256, aa=18, a=18
# t=288, aa=89, a=89
# t=320, aa=fa, a=fa
# t=352, aa=6b, a=6b
# t=384, aa=dc, a=dc
```

Figure 8.60 Simulation results of the test bench in Figure 8.59.

8.12 EXERCISES

Prepare design modules for the Exercises 1 to 10 below. In each case prepare a suitable test bench and test the design module [Arnold, Tocci]].

1. An adder to add two eight-digit numbers in BCD form.
2. Add two BCD digits using a look-up table.
3. Multiply two BCD digits using a look-up table.
4. An 8-digit multiplier all the digits being in BCD form.
5. A multiplier to multiply two 32-bit numbers.
6. A module to convert angle in radians to one in degrees.
7. A module to convert a 48-bit number into a decimal one in BCD form.
8. Combine the above two: Form a module to convert an angle in radians into one in degrees in decimal form.
9. A table to give the sines of angles. The given angle is a four-digit decimal number – in degrees in the range 0 to 90 degrees. The given table has two parts – a main table of four digits and a table of mean differences of one digit.
10. The outputs of a set of shift registers are designated as q1, q2, q3, *etc*. A selected set of these is exor'ed and the exor output fed as data input to q1. As the set of registers is clocked, the state vector representing the shift register outputs goes on changing state. With a properly selected set as the input to the exor gates, one can ensure that the state vector sequences through all the possible states in a "pseudo-random" manner. Thus an n stage shift register sequences through $2n - 1$ states.
11. Consider the code block in Figure 8.61(a). Complete the module and test it with the inputs a and b in Figure 8.61(b). Explain the difference in the waveforms of c and d.

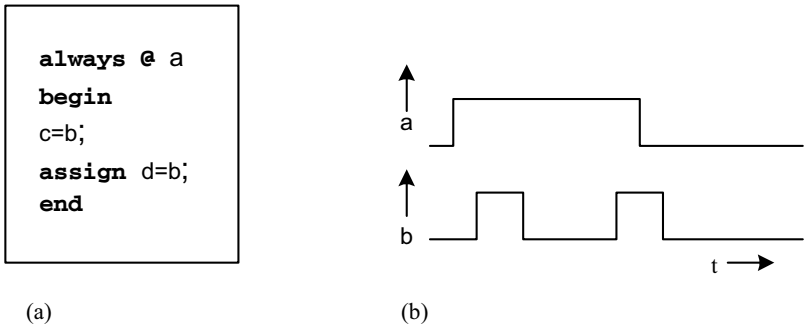

Figure 8.61 The behavioral block and the input waveforms for it for Exercise 11.

12. *A Serial Receiver with* **event**: A serial data stream is coming on an input data line. It is synchronized with a clock signal. Do the following in a module
    - Receive 8 bits and fill a byte-wide receive register.
    - Set **event** REC.
    - Use REC to transfer the received byte to the top of a FIFO.
13. *A Serial Transmitter with* **event**: Tr_buf is a byte-wide buffer. Serially output its content on a serial line. When Tr_buf is empty, set event TR. On TR event, load Tr_buf from bottom of FIFO.
14. Prepare modules to realize the priority encoder using the "**if**" and "**for**" constructs. Simulate and synthesize each.
15. In Example 8.8 the **event** @(**negedge** clk) succeeds **repeat**. Interchange the two and suitably modify the block with additional **begin** and **end** lines. Simulate, compare the results with those in the example, and explain the difference.
16. Complete the "block memory output" module in Figure 8.25. Test it with a suitable test bench.
17. Prepare modules for the following and simulate each with a test-bench:
    - Clear a block of memory.
    - Input a block of bytes to a register file.
    - Move a set of bytes from one to another page of memory with specified starting and ending addresses.
18. Use the **disable** construct and prepare modules for AND, NAND, and NOR functions. Follow the approach in Figure 8.34. Test each with corresponding test benches.

19. Use the **repeat** construct along with the **disable** construct to realize an AND gate. Synthesize the module and compare the synthesized circuits.
20. Repeat the above Exercise with **casez** and **if-else-if** constructs.
21. Repeat the above two Exercises for OR, NOR, and NAND functions.
22. What is the functional difference between the two blocks in Figure 8.62? Illustrate through suitable test benches.

    If the combination

    @(**posedge** en1)
    @(**posedge** en2)

    is replaced by

    @ (**posedge** en1 or **posedge** en2)

    how will the performance differ? Explain through test benches.

```
Initial
 begin
        #1 a=0;
        @(posedge en1)
        @(posedge en2)
        a=1;
end
```

```
Initial   begin
          #1 a=0;
          fork
          @(posedge en1)
          @)(posedge en2)
           join
          a=1;
          end
```

Figure 8.62 Two functional blocks to illustrate the difference between **begin–end** and **fork**–**join** pairs of constructs in Exercise 22.

23. Compare the behavior of the blocks in the above Exercise with one using the **if** construct.
24. A serial link has a clock rate of 1 MHz and a bit rate 1/32 times the clock rate. Set up a receiver to receive 8 successive bytes of data and to load them into a register file. The expected functioning of the unit is to be on the following lines:
    - A clock to function at 1 MHz. A bit rate clock derived from the main clock.
    - A flag En to enable serial reception.
    - A serial data input stream.
    - At the first positive edge of the main clock following En, transmission starts.
    - At every 4th pulse of the main clock, the input data line is to be sensed. A polling of 4 consecutive data bits decides the received output bit value and the status of an error bit.
    - Whenever the error flag goes high, the corresponding byte is made ff.

    Set up a test bench and test the functioning of the link.

25. Figure 8.63 shows a module. Get the waveforms of a and b by simulation.

```
module pulses;
reg [8:0] I;
reg a,b;
initial
while (I<100)
begin
 #1        a= I(0);
           b= I(1);
           I = I + 1;
end
initial I=0;
initial #100 $stop;
endmodule
```

Figure 8.63 A module to generate simple waveforms.

26. Generate three waveforms with the following characteristics (see Figure 8.64):

    - All have a time period of 21 time steps.
    - All are identical.
    - All have a continuous ON period of 5 ns.
    - All are equally phase-shifted.

    Generate the waveforms using **`case`**, **`if-else-if`** and **`for`** constructs.

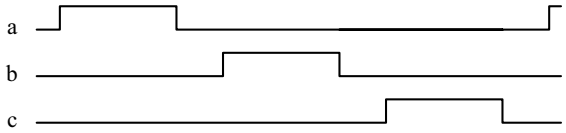

Figure 8.64 Three phase clock waveforms.

27. Generate the waveforms in Figure 8.65 using the **`case`**, **`if`**, and **`for`** loops. Use **`repeat`** and **`forever`** constructs for cyclic repetition.

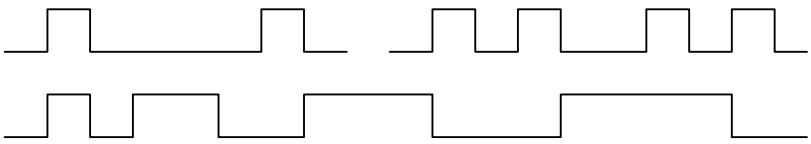

Figure 8.65 Different waveforms for Exercise 27.

28. A module and other modules instantiated within it can have a number of events scheduled for execution at the same time step. The sequence of execution is simulator-dependent. If any particular statement is assigned for execution with a zero time delay, it is executed as the last one in the concerned time step. Consider Example 8.23: The event buf_ful is assigned a zero time delay; delete the delay, simulate the module, and explain the difference in results, if any. The commented $write and $display statements may be activated for this.
29. Again consider example 8.24: The last statement in the block used to generate the serial data stream is assigned a 3 ns time delay. Delete the delay, simulate the module, and explain the difference in results, if any. The commented **`$write`** and **`$display`** statements may be activated for this.

9

FUNCTIONS, TASKS, AND USER-DEFINED PRIMITIVES

9.1 INTRODUCTIUON

Bigger designs are better arranged in small functional blocks; it facilitates debugging and any reorganization. Thus a module can have well-defined sub-modules inside, treated as separate entities. Functions and Tasks are such entities inside modules. They play three broad roles:

- A well-defined structure with a separate identity.
- They can hide some variables.
- They can be repeatedly invoked within the module.

User-defined primitive (UDP) provides an alternative form of a submodule; it can realize specific outputs. The UDP has a specific format. It can be defined by the user and used wherever necessary. The fact that the UDP has a specific format allows a straightforward definition – often at the expense of flexibility.

9.2 FUNCTION

A function is like a subroutine or a procedure in a program. It is defined separately within a module and can be called whenever necessary. When a function is declared with a function name, the system allocates a register for it. The name of the register is that of the function; and its type (as well as size) is also that of the function. When a function is called, the system executes the functional activity and generates the output. Eventually the output is assigned to the register identified for the function. The quantity returned by the function can be used as an operand in an assignment or in an expression. The structure of a function definition is shown in Figure 9.1. The significance of each of the quantities as well as the rules of using them is also explained in the figure. The use of functions is brought out through a set of examples.

Design Through Verilog HDL. T. R. Padmanabhan and B. Bala Tripura Sundari

ISBN: 0-471-44148-1

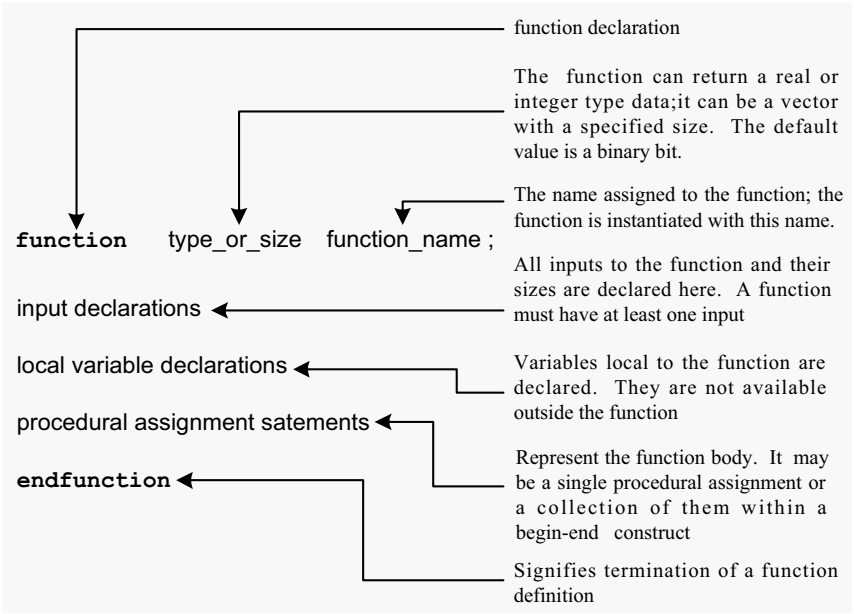

Figure 9.1 Structure for function definition.

Example 9.1

The function odd-parity is defined within the module parity-check in Figure 9.2. It generates a parity bit. The parity bit is 1 if the number of one-bits in the byte is odd. Otherwise it is zero. The module has an 8-bit vector input and a flag input – en. It has an output chk. Whenever the flag goes high, the function odd-parity is called. It returns the parity bit value and assigns it to chk in the module. parity-check is an example with a single-bit output-type function in it. The function has no local variables in it.

```
module parity_chk(a,en,chk);
input[7:0]a;
input en;
output chk;
wire[7:0] a;
reg chk;
always @(posedge en)
begin
        chk=pb(a);
        $display("t=%0d, a = %b, en = %0b, pb = %0b ",$time,a,en,chk);
end
```

continued

continued

```
function pb;
input[7:0]a;
pb=^a;
endfunction

endmodule
module tst_pchk;
reg [7:0]a;
reg en;
wire chk;
integer i;
parity_chk pchk(a,en,chk);
initial #0 en=1'b0;
always #2 en = ~en;
initial
begin
        #1 a=8'h00;
        for(i=0;i<8;i=i+1)
        begin
                #4 a=a+3'o6;
        end
end
initial #40 $stop;
endmodule
```

Figure 9.2 A module for parity generation through a function.

```
# t=2, a = 00000000, en = 1, pb = 0
# t=6, a = 00000110, en = 1, pb = 0
# t=10, a = 00001100, en = 1, pb = 0
# t=14, a = 00010010, en = 1, pb = 0
# t=18, a = 00011000, en = 1, pb = 0
# t=22, a = 00011110, en = 1, pb = 0
# t=26, a = 00100100, en = 1, pb = 0
# t=30, a = 00101010, en = 1, pb = 1
# t=34, a = 00110000, en = 1, pb = 0
# t=38, a = 00110000, en = 1, pb = 0
```

Figure 9.3 Simulation results of the test bench in Figure 9.2.

Example 9.2

Figure 9.4 shows another module for parity generation. The module has a function to count the number of one-bits in the input byte. In the module the parity bit is decided by mod-2 division of the number returned by the function. The function has an integer declared and used within it. (In contrast, in the last example the parity bit was generated directly within the function defined.)

```
module parity(p,a,En);
input[7:0]a;
input En;
output p;
reg p;
always @(posedge En)
begin
        p=n1(a)%2;  //Use n1 & generate the parity bit.
        $display("t=%0d, a = %b, en = %b, p = %b ",$time,a,en,p);
end


function integer n1;  //A function to count the number of 1 bits in a byte
input[7:0]a;
integer i;
        for(i=0;i!=8;i=i+1)
        begin
                if(i==0) n1=0;
                if(a[i]) n1=n1+1;
        end
endfunction
endmodule
```

Figure 9.4 A module to generate a parity bit: The parity bit is generated by counting the number of one-bits in a function and doing a mod-2 division.

Example 9.3

In the module of Figure 9.5 the number of one-bits is decided by shifting out the bits of the input vector and counting the ones in them. Otherwise the module is similar to the one in Figure 9.4. The module (as well as the previous ones) can be easily extended to generate the parity bit for wider binary streams.

```
module parity_a(p,a,En);
input[7:0]a;
input En;
output p;
reg p;
always @(posedge En)
begin
        p=nn(a)%2;
        $display("t=%0d, a = %b, En = %b, p = %b ",$time,a,En,p);
end

function integer nn;
input[7:0]a;
integer i;
begin
        for(i=0;i!=8;i=i+1)
        begin
        if(i==0) nn=0;
        if(a[i]) nn=nn+1;
        a=a>>1;
        end
end
endfunction
endmodule
```

Figure 9.5 Another module to generate a parity bit similar to that in Figure 9.4.

Example 9.4

Figure 9.6 shows an adder module to add two 2-bit numbers. The module has two functions defined in it – a half-adder and a full-adder. Further, one can see that the full-adder function itself calls the half-adder function within it. The module calls the full-adder function repeatedly within itself. A test bench for the adder is also included in the figure. The simulation results are shown in Figure 9.7.

```
module adderfun(r,p,q,En);
input[1:0] p,q; input En; output [2:0] r; reg[2:0]r,c; integer i;
always@(posedge En)
```

continued

continued

```
begin
        for(i=0;i<2;i=i+1)
        begin
                if(i==0) c[i]=1'b0;
                {c[i+1'b1],r[i]}=fa(p[i],q[i],c[i]);
        end
                r[2]=c[2];
                $display("t=%0d, En = %b, p = %b, q = %b, r = %b ",$time
                ,En,p,q,r);
end

function[1:0] ha;
input a,b;
ha={a&b,a^b};
endfunction

function [1:0]fa;
input a,b,c; reg[1:0]a1,a2,aa2;
begin
        a1=ha(a,b);
        aa2=ha(a1[0],c);
        a2[1] =  (aa2[1]|a1[1]);
        a2[0] = aa2[0];
        fa=a2;
end
endfunction
endmodule

module tst_adder_fun; //testbench;
reg [1:0] p,q; reg En; wire [2:0] r;
adderfun aa(r,p,q,En);
always #2 En=~En;
initial    begin
                En=1'b0; p=2'b01;q=2'b00;
                #5 p=2'b10;q=2'b10;
                #4 p=2'b10;q=2'b11;
                #4 p=2'b11;q=2'b11;
                #4 p=2'b01;q=2'b01;
        end
initial #30 $stop;
endmodule
```

Figure 9.6 A module to illustrate a function calling another one; a test bench is also included in the figure.

```
# t=2, En = 1, p = 01, q = 00, r = 001
# t=6, En = 1, p = 10, q = 10, r = 100
# t=10, En = 1, p = 10, q = 11, r = 101
# t=14, En = 1, p = 11, q = 11, r = 110
# t=18, En = 1, p = 01, q = 01, r = 010
# t=22, En = 1, p = 01, q = 01, r = 010
# t=26, En = 1, p = 01, q = 01, r = 010
```

Figure 9.7 Results of running the test bench in Figure 9.6.

Example 9.5

A module to add two 32-bit numbers is shown in Figure 9.8. It is essentially a scaled-up version of the one in Figure 9.6. The addition is initiated by the En input going high; it is carried out in one time step. A test bench is also included in the figure. The simulation results for a specific set of input number combinations are shown in Figure 9.9.

```
module add32(r,p,q,En);
input[31:0] p,q; input En; output [32:0] r; reg[32:0]r,c; integer i;
always@(posedge En) begin
                        for(i=0;i<32;i=i+1)
                        begin
                        if(i==0) c[i]=1'b0;
                        {c[i+1'b1],r[i]}=fa(p[i],q[i],c[i]);
                        end
                        r[32]=c[32];
                        $display( "t=%0d, En = %b, p = %0h, q = %0h, r =
                        %0h ",$time, En,p,q,r);
                    end

function[1:0] ha;
input a,b;
ha={a&b,a^b};
endfunction

function [1:0]fa;
input a,b,c; reg[1:0]a1,a2,aa2;
begin
        a1=ha(a,b);
```

continued

continued

```
        aa2=ha(a1[0],c);
        a2[1] =  (aa2[1]|a1[1]);
        a2[0] =   aa2[0];
        fa=a2;
end
endfunction
endmodule

module tst_add32; //testbench;
reg [31:0] p,q; reg En; wire [32:0] r;
add32 aa(r,p,q,En);
always #2 En=~En;
initial    begin
                #0 En  = 1'b0;
                #3 p    = 32'h1234;        q = 32'h4321;
                #4 p    = 32'h12345678; q = 32'h98765432;
                #4 p    = 32'habcdef12;  q = 32'hbbccddee;
                #4 p    = 32'hfedcba39;  q = 32'h13579bdf;
                #4 p    = 32'h9876abcd; q = 32'hfedc8765;
                #4 p    = 32'hf0e0d0c0;  q = 32'h11020304;
        end
initial #30 $stop;
endmodule
```

Figure 9.8 A scaled-up version of the 2-bit adder in Figure 9.6 to add 32-bit numbers.

```
# t=2, En = 1, p = x, q = x, r = x
# t=6, En = 1, p = 1234, q = 4321, r = 5555
# t=10, En = 1, p = 12345678, q = 98765432, r = aaaaaaaa
# t=14, En = 1, p = abcdef12, q = bbccddee, r = 1679acd00
# t=18, En = 1, p = fedcba39, q = 13579bdf, r = 112345618
# t=22, En = 1, p = 9876abcd, q = fedc8765, r = 197533332
# t=26, En = 1, p = f0e0d0c0, q = 11020304, r = 101e2d3c4
```

Figure 9.9 Results of running the test bench in Figure 9.8.

Example 9.6

A variant of the adder in Example 9.4 is shown in Figure 9.10: After the enable input en goes high, the full-adder function is called repeatedly in successive clock pulses and bit-wise addition is carried out. The figure also includes a test bench. As can be seen from the simulation results in Figure 9.11, each addition is spread over two clock periods.

```
module adderfunb(clk,r,p,q,En);
input[1:0] p,q; input En,clk; output [2:0] r; reg[2:0]r,c; integer i;
always@(posedge En) begin
                        for(i=0;i<2;i=i+1) begin
                                            @(posedge clk)
                                            if(i==0) c[i]=1'b0;
                                            {c[i+1'b1],r[i]}=fa(p[i],q[i],c[i]);
                                             end
                        r[2]=c[2];
                        $display(" t=%0d, clk = %b, En = %b, p = %b, q = %b,
                        r = %b ",$time,clk,En,p,q,r);
                    end
function[1:0] ha;
input a,b;
ha={a&b,a^b};
endfunction

function [1:0]fa;
input a,b,c; reg[1:0]a1,a2,aa2;
begin
        a1=ha(a,b);
        aa2=ha(a1[0],c);
        a2[1] =  (aa2[1]|a1[1]);
        a2[0]=aa2[0];
        fa=a2;
end
endfunction
endmodule

module tst_adder_funb();
reg [1:0] p,q; reg En,clk; wire [2:0] r;
adderfunb bb(clk,r,p,q,En);
always #2 clk=~clk;
initial     begin
                    clk=1'b0;    En=1'b0; p=2'b01; q=2'b00;
                #1 En=1'b1; #6 En=1'b0; p=2'b01; q=2'b10;
                #1 En=1'b1; #7 En=1'b0; p=2'b01; q=2'b01;
                #1 En=1'b1; #7 En=1'b0; p=2'b10; q=2'b01;
                #1 En=1'b1; #7 En=1'b0; p=2'b10; q=2'b10;
                #1 En=1'b1; #7 En=1'b0; p=2'b10; q=2'b11;
                #1 En=1'b1; #7 En=1'b0; p=2'b11; q=2'b11;
                #1 En=1'b1; #7 En=1'b0;
        end
initial #60 $stop;
endmodule
```

Figure 9.10 A variant of the 2-bit adder in Figure 9.6; bit-wise addition is carried out in successive clock pulses.

```
#  t=6, clk = 1, En = 1, p = 01, q = 00, r = 001
#  t=14, clk = 1, En = 1, p = 01, q = 10, r = 011
#  t=22, clk = 1, En = 1, p = 01, q = 01, r = 010
#  t=30, clk = 1, En = 1, p = 10, q = 01, r = 011
#  t=38, clk = 1, En = 1, p = 10, q = 10, r = 100
#  t=46, clk = 1, En = 1, p = 10, q = 11, r = 101
#  t=54, clk = 1, En = 1, p = 11, q = 11, r = 110
```

Figure 9.11 Simulation results of the test bench for the adder module in Figure 9.10.

Example 9.7

A module to add 32-bit numbers is shown in Figure 9.12. It is a scaled-up version of that in the last example. The addition commences after the enable bit En goes high. Starting with the LSB, one bit is added at every succeeding clock pulse. Addition is completed in 32 clock pulses. The simulation results with a set of 32-bit numbers is shown in Figure 9.13.

```
module add32_a(clk,r,p,q,En);
input[31:0] p,q;input En,clk; output [32:0] r; reg[32:0]r,c; integer i;
always@(posedge En)       begin
          for(i=0;i<32;i=i+1)
          begin
                    @(posedge clk)   begin
                                       if(i==0) c[i]=1'b0;
                                       {c[i+1'b1],r[i]}=fa(p[i],q[i],c[i]);
                                       end
          end
          r[32]=c[32];
          $display( "t=%0d,  En = %b,  p = %0h, q = %0h, r = %0h
",$time,En,p,q,r);              end
function[1:0] ha;
input a,b; ha={a&b,a^b};
endfunction

function [1:0]fa;
input a,b,c; reg[1:0]a1,a2,aa2;
begin
          a1     =  ha(a,b);
          aa2    =  ha(a1[0],c);
          a2[1] =  (aa2[1]|a1[1]);
```

continued

continued

```
            a2[0] =  aa2[0];
            fa    =  a2;
end
endfunction
endmodule

module tst_add32a();
reg [31:0] p,q; reg En,clk; wire [32:0] r;
add32_a bb(clk,r,p,q,En);
always #1 clk=~clk;
initial     begin
            clk=1'b0;En=1'b0;p=32'h1234;q=32'h4321;
 #1         En=1'b1;#100 En=1'b0;p=32'h12345678;q=32'h98765432;
 #1         En=1'b1;#99 En=1'b0;p=32'habcdef12;q=32'hbbccddee;
 #1         En=1'b1;#99 En=1'b0;p=32'hfedcba39;q=32'h13579bdf;
 #1         En=1'b1;#99 En=1'b0;p=32'h9876abcd;q=32'hfedc8765;
 #1         En=1'b1;#99 En=1'b0;p=32'hf0e0d0c0;q=32'h11020304;
 #1         En=1'b1;#99 En=1'b0;
            end
initial #900 $stop;
endmodule
```

Figure 9.12 A 32-bit adder with the addition done in successive clock pulses.

```
# t=65,  En = 1,  p = 1234, q = 4321, r = 5555
# t=165,  En = 1,  p = 12345678, q = 98765432, r = aaaaaaaa
# t=265,  En = 1,  p = abcdef12, q = bbccddee, r = 1679acd00
# t=365,  En = 1,  p = fedcba39, q = 13579bdf, r = 112345618
# t=465,  En = 1,  p = 9876abcd, q = fedc8765, r = 197533332
# t=565,  En = 1,  p = f0e0d0c0, q = 11020304, r = 101e2d3c4
```

Figure 9.13 Simulation results of the test bench for the adder in Figure 9.12.

9.2.1 Trade-off Between Hardware and Speed

Examples 9.5 and 9.7 represent two extreme cases of a trade-off between speed and hardware. Minimal hardware is used in Example 9.7 to carry out the addition, but the execution time is a maximum here due to the repeated and sequential use of the same hardware block. In contrast, in Example 9.5 the same hardware is replicated to the maximum extent and the addition is carried out “at one go”, that

is, in minimum time. Circuit-wise, it is a trade-off between silicon area and speed. One can have nibble or byte adders and do nibble-wise or byte-wise addition; these represent intermediate levels of trade-offs. Algebraic or logic operations, register-based operations, *etc*., are other examples calling for similar trade-off decisions. Buswidth, memory organization, and ALU sizing all call for such trade-off decisions. In all such cases a decision may have to be based on considerations of speed of operation, power consumption, development time, cost, *etc*.

9.2.2 Scope of Functions

A few observations on functions and their use are in order here [IEEE].

- A function has only input arguments. It is to have at least one input. When a function with multiple input ports is called, the order of arguments in the calling statement should match that of the input declarations within the function definition.
- A function returns an output. It has no separate output ports.
- A function can have variables declared and used within it – these are variables local to the function.
- A function can be defined anywhere within the module.
- Event or timing based controls are not possible within a function. This restricts the function to be of a combinational logic type.
- A function can be called from within another function. Both the functions are to be defined within the module.
- A function in a module can be called from another module through proper hierarchical referencing.
- A function can be called repeatedly within the module of definition.
- Expressions can be used as arguments while calling a function.
- Definition of a function should not be within any initial or always block. or within another function.
- A function uses a register of the declared type and size to return the value of the output. Such a returned value can be **`real`**, **`integer`**, **`time`**, or **`realtime`** type. It can also be a vector with a range.
- Every variable declared inside a function has a corresponding location inside. These locations are physical entities. Each time a function is called, the same set of locations is reused. This is in contrast to the instantiation of a module where with every instantiation, a fresh set of locations is assigned.

9.2.3 Recursive Functions

Consider a function to compute the sum of the squares of the first n natural numbers: The sum designated as S_n can be expressed as

$S_n = .n^2 + (n - 1)^2 + \cdots + 3^2 + 2^2 + 1^2$
S_n can be expressed as
$S_n = n^2 + S_{n-1}$

where S_{n-1} represents the sum of the squares of the first $(n - 1)$ natural numbers. Thus if S_{n-1} were known, S_n can be obtained by adding n^2 to it. Continuing the same argument one can recursively arrive at the following:

$S_{n-1} = (n - 1)^2 + S_{n-2}$
$S_{n-2} = (n - 2)^2 + S_{n-3}$
…
…
$S_2 = 2^2 + S_1$

We know that

$S_1 = 1$.

The actual computation is carried out in the reverse order; that is, one computes S_1 directly and the subsequent sums S2, S3, *etc*., are computed from it recursively – every sum by adding an increment to the previous sum.

A similar procedure can be adopted to compute factorials, infinite series and so on. Latest version of the LRM (2001) has expanded the scope of Functions to accommodate recursive functions. The keyword **`automatic`** following the keyword **`function`** implies it to be recursive. A recursive function can be called in the same manner as a nonrecursive function. Recursive function call is explained here through an example.

Example 9.8

The module sum_sq in Figure 9.14 computes the sum of the squares the first *n* natural numbers.

```
function automatic integer sum_sq;
input n;
begin
        if(n==1) sum_sq =1;
        else sum_sq = sum_sq + n*n;
end
endfunction
```

Figure 9.14 A module to compute the sum of squares of the first *n* natural numbers.

The term "**automatic**" in the function declaration statement ensures recursive computation. Thus if *n* is assigned the value 4, during compilation sum_sq (4) will be successively replaced by

sum_sq (3) + 4^2,
sum_sq (4) + 3^2 + 4^2,
sum_sq (4) + 2^2 + 3^2 + 4^2 and finally by
1^2 + 2^2 + 3^2 + 4^2.

9.3 TASKS

The role of a task in a module is similar to that of a subroutine in a program. It is defined within a module and can be called as many times as desired within a procedural block. Its scope and role are wider than those of a function.

9.3.1 Task Definition

The task definition is brought out in Figure 9.15. The first statement starts with the keyword task; it is followed by an identifier name and the customary semicolon. The input, inout, and the output declarations follow. Their order is not rigid. The body of the task comprises of a number of behavioral level statements. They may be executed in zero time or at specified time intervals or events. Thus the time of exit from a task can differ from that of entry to it.

9.3.2 Task Enabling

A task is enabled through a statement akin to the instantiation of a gate. It is enabled like a procedural assignment by specifying the task name followed by the list of arguments within brackets followed by the semicolon. A typical enabling statement has the form

Do_it (Expression1, Expression2, . .);

where
Do_it is the name of the task being enabled,
Expression1 is the first argument,
Expression2 is the second argument,
and so on.

The type and order of the arguments should match those of the respective declarations within the definition of the task. In a general case, an argument can be an expression. The following are characteristic of a task:

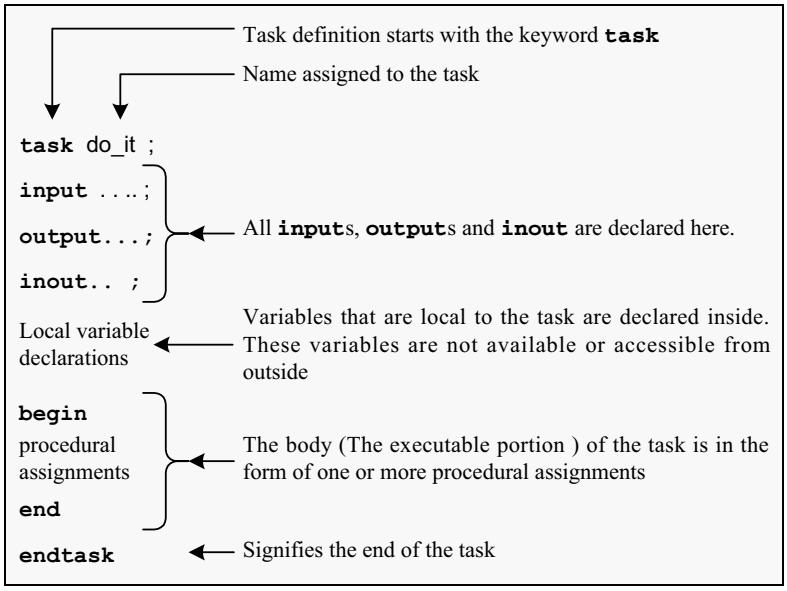

Figure 9.15 Typical structure of a task.

- A task can be activated by an event, sensitivity list, *etc*.
- A task can have activities assigned within it which are event-controlled or time-controlled.
- A task can have input, output and inout; however it need not necessarily have any of these; it can be complete in itself.
- A task can enable other tasks and functions.
- A task can call itself. The latest version of the LRM supports recursion. The keyword **automatic** is added to the keyword task to make it recursive.
- All assignments to a task are passed to it by value and not through a pointer to the argument.
- A task in a module can be invoked from another module through a hierarchical reference.

The arguments passed to a task retain their type within their environment of use. Thus a **wire**-type argument passed to a task as input cannot have its value altered within the task through an assignment.

There are no apparent restrictions on the input arguments of a task. They can be nets, regs, or expressions involving them. But any argument of inout or output type has to be a variable or of a similar type; the restrictions are similar to those on the quantities on the left side of procedural assignments. The use of tasks is illustrated through a set of four examples here.

Example 9.9

Figure 9.16 shows a module to count the total number of 1 bits in a nibble. A task has been defined to do the counting; the task has vector-type input and output; it has an integer defined within. The task has been invoked in the main module. A test-bench is also included in the figure. The simulated results for a set of inputs are given in Figure 9.17.

```
module oness_counter;
reg [3:0]x;reg [2:0]y;
always@(x)onescounter(x,y);

task onescounter;
input [3:0]x; output[2:0]y; integer i;
begin
        y=0;
        for(i=0;i<=3;i=i+1) if (x[i])y= y+1;
end
endtask

initial x=3'b000;
always #3 x=x+2'b11;
initial $monitor(" t=%0d, y= %b, x = %b ",$time,y,x);
initial #30 $stop;
endmodule
```

Figure 9.16 A module to count the number of 1 bits in a nibble.

```
#  t=0, y= 000, x = 0000
#  t=3, y= 010, x = 0011
#  t=6, y= 010, x = 0110
#  t=9, y= 010, x = 1001
#  t=12, y= 010, x = 1100
#  t=15, y= 100, x = 1111
#  t=18, y= 001, x = 0010
#  t=21, y= 010, x = 0101
#  t=24, y= 001, x = 1000
#  t=27, y= 011, x = 1011
```

Figure 9.17 Simulated results with the test bench in Figure 9.16.

Example 9.10

Figure 9.18 shows a module to divide a given clock with a given number. The scaling number can be changed if necessary. The task uses input, output and inout type of quantities. The waveforms of the input clock and the slower output clock obtained by simulating the test bench are shown in Figure 9.19.

```
module clk_tst;
reg clk,sclk;reg [3:0] n,nn;
always #2 clk=~clk;

task  sl_clk;
input clk; input[3:0]nn; inout[3:0] n;
output sclk;
begin
        if(n!=4'h0)         begin
                                    n     = n-1'b1;
                                    sclk = 1'b0;
                            end
        else        begin
                            n     = nn;
                            sclk = 1'b1;
                    end
end
endtask

always @(negedge clk) sl_clk(clk,n,nn,sclk);
initial
begin
        clk=1'b0;nn=4'h2;n=nn; #45$stop;
end
initial $monitor($time, "n=%0d, clk=%0b, sclk=%0b",n,clk,sclk);
endmodule
```

Figure 9.18 A module to generate a slower clock from a given clock input.

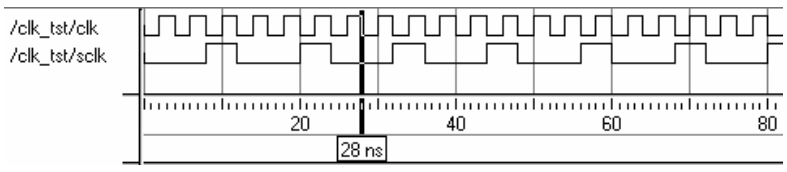

Figure 9.19 Simulation results of the module in Figure 9.18.

Example 9.11

The adder in Example 9.4 has been modified and shown in Figure 9.20. The half- and full-adders have been defined as tasks and invoked to carry out the vector addition. The half-adder has been invoked twice within the full-adder task. The test-bench and simulation results are not repeated here. The module can be directly expanded to add wider numbers.

```
module addertsk(r,p,q,En);
input[1:0] p,q; input En; output [2:0] r;
reg[2:0]r,c; integer i;
always@(posedge En)
        begin
                for(i=0;i<2;i=i+1)
                begin
                        if(i==0) c[i]=1'b0;
                        fa(p[i],q[i],c[i],{c[i+1'b1],r[i]});
                end
                r[2]=c[2];
                $display("t=%0d, En = %b, p = %b, q = %b, r = %b ",$time
                ,En,p,q,r);
        end

task ha;
input a,b; output[1:0] hfsum;
hfsum={a&b,a^b};
endtask

task fa;
input a,b,c; output[1:0]a2; reg[1:0]a1,aa2;
begin
        ha(a,b,a1);
        ha(a1[0],c,aa2);
        a2[1] =  (aa2[1]|a1[1]);
        a2[0] =   aa2[0];
end
endtask
endmodule
```

Figure 9.20 A 2-bit adder using half-adder and full-adder tasks.

Example 9.12

The half-adder and full-adder tasks in Example 9.11 have been used to carry out addition of 2-bit numbers in the module of Figure 9.21. The addition has been carried out in successive clock pulses as with Example 9.6. The test bench and simulation results have been omitted. Once again the module can be redone easily to add wider numbers.

```
module addertskb(clk,r,p,q,En);
input[1:0] p,q;input En,clk;output [2:0] r;
reg[2:0]r,c;integer i;
always@(posedge En)
        begin
                for(i=0;i<2;i=i+1)
                begin
                        @(posedge clk)
                        if(i==0) c[i]=1'b0;
                        fatsk(p[i],q[i],c[i],{c[i+1'b1],r[i]});
                end
                r[2]=c[2];
                $display(" t=%0d, clk = %b, En = %b, p = %b, q = %b, r = %b
",$time,clk,En,p,q,r);
        end

task hatsk;
input a,b;output[1:0]ha;
ha={a&b,a^b};
endtask

task fatsk;
input a,b,c;output[1:0]a2;reg[1:0]a1,aa2;
begin
        hatsk(a,b,a1);
        hatsk(a1[0],c,aa2);
        a2[1] =  (aa2[1]|a1[1]);
        a2[0] =  aa2[0];
end
endtask
endmodule
```

Figure 9.21 Another 2-bit adder using half-adder and full-adder tasks.

9.4 USER-DEFINED PRIMITIVES (UDP)

The primitives available in Verilog are all of the gate or switch types. Verilog has the provision for the user to define primitives – called "user defined primitive (UDP)" and use them. A UDP can be defined anywhere in a source text and instantiated in any of the modules. Their definition is in the form of a table in a specific format. It makes the UDP types of functions simple, elegant, and attractive. UDPs are basically of two types – combinational and sequential. A combinational UDP is used to define a combinational scalar function and a sequential UDP for a sequential function.

9.4.1 Combinational UDPs

A combinational UDP accepts a set of scalar inputs and gives a scalar output. An **inout** declaration is not supported by a UDP. The UDP definition is on par with that of a module; that is, it is defined independently like a module and can be used in any other module. The definition cannot be within any other module.

Definition of a combinational type of UDP is illustrated through an example in Figure 9.22; it shows a simple UDP for an AND operation. The following are noteworthy:

- The first statement starts with the keyword "primitive", it is followed by the name assigned to the primitive and the port declarations.
- A UDP can have only one output port. It has to be the first in the port list.
- All the ports following the first are input ports and are all scalars.
- **inout** ports are not permitted in a UDP definition.
- Output and input are declared in the body of the UDP.

```
primitive udp_and (out, in1, in2);
output out;
input in1, in2;
table
//          In1         In2         Out
            0           0:          0;
            0           1:          0;
            1           0:          0;
            1           1:          1;
endtable
endprimitive
```

Figure 9.22 A two-input AND gate defined as a UDP.

- The behavior block of the primitive is given in the form of a table. It is specified between keywords **table** and **endtable**.
- The combinational function is defined as a set of rows (akin to the truth table).
- All the input values are specified first – each in a separate field in the same order as they appear in the port declaration.
- A colon and then the output value follow the set of input values. The statement ends with a semicolon – as with every statement in Verilog.
- A comment line is inserted in the example following the "**table**" entry**.** It facilitates understanding the tabular entries.
- All the inputs are nets – **wire**-type. Hence there is no need for a separate type definition.
- Output can be of the net or **reg** type depending upon the type of primitive – explained later.
- The last keyword statement – "**endprimitive**" – signifies the end of the definition.

9.4.2 More General Combinational UDPs

The UDP for the AND gate in Figure 9.22 specifies output values only for definite values of the inputs but not for their **x** states. A full and general definition of a UDP is characterized by the following additional factors:

- The output can take on only three values – **0**, **1**, or **x**. It cannot take the value **z**.
- Outputs can be defined for **0**, **1**, or **x** values of the inputs but not for the **z** state. However if an input takes the value **z**, it is taken as **x**.
- All the undefined input combinations lead to **x** state in the output. Hence it is desirable to specify outputs for all the possible input combinations.

Figure 9.23 shows the UDP definition of an AND gate with all the input combinations included. A test-bench for the UDP and the simulation results are shown in Figure 9.24.

A two-input UDP has nine rows of tabular entries; their number increases rapidly as the number of input logic variables increases. LRM has the provision to make the UDP definition more compact. The symbol "?" can be used to signify all the possible values – that is, 0, 1, or **x**. Figure 9.25 shows the elaborate AND gate UDP of Figure 9.23 made compact in this manner. Wherever possible, one can use the symbol "**b**" to signify "0" or "1" values and reduce the table size further.

```
Primitive udp_and (out, in1, in2);
Output out; //UDP of an AND gate defined fully
Input in1, in2;
Table
//          In1         In2         Out
            0           0:          0;
            0           1:          0;
            1           0:          0;
            1           1:          1;
            X           0:          0;
            X           1:          X;
            X           X:          X;
            0           X:          0;
            1           X:          X;
Endtable
Endprimitive
```

Figure 9.23 A more exhaustive definition of the two2-input AND gate UDP of Figure 9.21.

```
module tst_udp_and();
reg in1,in2; wire out;
udp_and uand(out,in1,in2);
initial begin in1=1'b0;in2=1'b0; end
always   begin
         #2 in1=1'b0;in2=1'b1;
         #2 in1=1'b1;in2=1'b0;
         #2 in1=1'b1;in2=1'b1;
         end
initial $monitor($time ,"in1 = %b ,in2 = %b ,out = %b ",in1,in2,out);
initial #18 $stop;
endmodule

Simulation results
//#          0in1 = 0 ,  in2 = 0 ,  out = 0
//#          2in1 = 0 ,  in2 = 1 ,  out = 0
//#          4in1 = 1 ,  in2 = 0 ,  out = 0
//#          6in1 = 1 ,  in2 = 1 ,  out = 1
//#          8in1 = 0 ,  in2 = 1 ,  out = 0
//#         10in1 = 1 , in2 = 0 ,  out = 0
//#         12in1 = 1 , in2 = 1 ,  out = 1
//#         14in1 = 0 , in2 = 1 ,  out = 0
//#         16in1 = 1 , in2 = 0 ,  out = 0
```

Figure 9.24 A test bench for the UDP module of Figure 9.23 and the simulation results.

```
Primitive udp_and_b (out, in1, in2);
Output out; // UDP of an AND gate defined compactly
Input in1, in2;
Table
 //          In1          In2          Out
             ?            0:           0;
             0            ?:           0;
             x            X            x
             1            1:           1;
Endtable
Endprimitive
```

Figure 9.25 The UDP of Figure 9.22 made compact using the symbol "?".

9.4.3 Instantiation of an UDP

UDPs are instantiated in the same manner as gate primitives (see the test bench in Figure 9.24). It is further illustrated here through an example.

Example 9.13

The full adder accepts three input bits and outputs two bits – a sum bit and a carry bit. Figure 9.26 shows UDPs for the sum and the carry bits as well as a full adder module using them. Figure 9.27 shows a test-bench for the Full Adder as well as the simulation results.

```
primitive udpsum(sum, in1,in2,carryi);
output sum;
input in1, in2, carryi;
table
//       in1      in2      carryi:     sum
          0        0        0:          0;
          1        1        0:          0;
          0        1        1:          0;
          1        0        1:          0;
          1        0        0:          1;
          0        1        0:          1;
          0        0        1:          1;
          1        1        1:          1;
endtable
endprimitive
```

continued

continued

```
primitive udpcar(caro,in1,in2,cari); // This udp is  for carryout
output caro; input in1, in2, cari;

table
//        in1       in2       cari      caro
           0         0         ? :       0 ;
           0         ?         0 :       0 ;
           ?         0         0 :       0 ;
           b         1         1 :       1 ;
           1         b         1 :       1 ;
           1         1         b :       1 ;
endtable
endprimitive

module fa (car_o, sum_o, in1, in2, car_i);
input in1, in2, car_i; output car_o, sum_o;
udpcar aa(car_o,in1,in2,car_i);
udpsum bb(sum_o, in1,in2,car_i);
endmodule
```

Figure 9.26 A full adder module with the sum and carry bits generated through UDPs.

```
module fa_tst;
reg [2:0] a;wire c,s;integer i;
fa cc(c,s,a[0],a[1],a[2]);
initial for(i=1;i<8;i=i+1)
begin
            a=i;
 #1         $display($time, "a=%b, cs=%b%b",a, c, s);
end
initial #10 $stop;
endmodule

Simulation results
#                  1a=001, cs=01
#                  2a=010, cs=01
#                  3a=011, cs=10
#                  4a=100, cs=01
#                  5a=101, cs=10
#                  6a=110, cs=10
#                  7a=111, cs=11
```

Figure 9.27 A test bench for the full adder module of Figure 9.26 and the simulation results for the same.

Observations:

- With three inputs and three states for each input (0, 1, and **x**), the full table of definition has 27 entries. Such definitions become cumbersome as the number of inputs increase to even moderate values – say 4 or 5.
- Only the entries essential to the definition of the primitive are included here. Others which lead to **x** output are left out intentionally. Thus with the carry primitive if any two inputs have **x** values, the output car_o too has **x** value. Hence such a row has not been specified.
- "?" and "**b**" have been used in the primitive definition to make the tables more compact

9.4.4 Combinational UDP and Function

Definition-wise, UDP and function are similar, though their formats differ (*i.e.*, a UDP definition is in the form of a table while the function definition is as a sequence of procedural assignments). UDPs are stand-alone-type primitives and can be instantiated in any module. In contrast, a function is defined within a module; it cannot be accessed anywhere outside the module of definition.

9.4.5 Sequential UDPs

Any sequential circuit has a set of possible states. When it is in one of the specified states, the next state to be taken is described as a function of the input logic variables and the present state [Wakerly]. A positive or a negative going edge or a simple change in a logic variable can trigger the transition from the present state of the circuit to the next state. A sequential UDP can accommodate all these. The definition still remains tabular as with the combinational UDP. The next state can be specified in terms of the present state, the values of input logic variables and their transitions. The definition differs from that of a combinational UDP in two respects:

- The output has to be defined as a **reg**. If a change in any of the inputs so demands, the output can change.
- Values of all the input variables as well as the present state of the output can affect the next state of the output. In each row the input values are entered in different fields in the same sequence as they are specified in the input port list. It is followed by a colon (:). The value of the present state is entered in the next field which is again followed by a colon (:). The next state value of the output occupies the last field. A semicolon (;) signifies the end of a row definition (see the examples below).

As can be seen from the UDPs considered so far, its definition apparently calls for the use of a large number of tabular statements; it is all the more true of the sequential UDPs. Some shorthand notations are possible to make the UDP table more compact. All the notations that can be used are given in Table 9.1. Judicious selection and use of the symbols can make the tables compact.

Two examples of sequential UDPs are considered here – one being level-sensitive and the other edge-sensitive.

Example 9.14 A UDP for a D Latch

Figure 9.28 shows a UDP for a D latch (and a test bench for the same). It is an example of a level sensitive sequential UDP. The tabular description for the latch has been made succinct with the use of symbols – and ?. Any undefined input combination results in **x** value for the output; hence the output has not been separately defined for the **x** value of input in the table. Repeated use of the symbol ? has made the UDP table compact. The three rows of the table signify the following:

1. When clk = 1, if din = 0, the next state (qn) is also at 0 whatever be the value of present state (qp).
2. When clk = 1, if din = 1, the next state (qn) is also at 1 whatever be the value of present state (qp).
3. When clk = 0, the output (next state) does not change even if din changes.

Simulation results are shown in Figure 9.29.

Table 9.1 Symbols for UDP tabular rows

Symbol	Significance	Restrictions of use
B or **b**	0 and 1 values	Only in the input or current state fields
?	0, 1 or ,**x** value	Only in the input or current state fields
–	No change	Only in the output field of sequential UDP
(*mn*)	Change of value from *m* to *n*	Only in the input field. *m* & *n* can be 0, 1, **x**, **b**, or ?
*	Same as (??)	Only in the input field
r	Same as (01)	
f	Same as (10)	
p	Rise from 0 or **x** to **x** or 1	
n	Fall from 1 or **x** to **x** or 0	

```
primitive dlatch(q,din,clk);
output q; input din,clk; reg q;

table
//      din     clk     qp      qn
        0       1 :     ? :     0;      // If clk is at 1 state, the output
        1       1 :     ? :     1;      //follows the input.  If clk is at 0
        ?       0 :     ? :     -;      // state, the output remains frozen
endtable
endprimitive

module dlatch_tst;
wire q; reg din,clk;
dlatch ll(q,din,clk);
initial
begin
        clk=1'b1;din=1'b0;
        repeat (2)begin #4 din=1'b1; #4 din=1'b0; end
        clk=1'b0;repeat (2)begin #4 din=1'b1; #4 din=1'b0; end
        $stop;
end
initial $monitor($time ,"clk = %b, din = %b, q = %b ",clk,din,q);
endmodule
```

Figure 9.28 A D-latch module described as a level-sensitive UDP and a test bench for it.

```
#                0clk = 1, din = 0, q = 0
#                4clk = 1, din = 1, q = 1
#                8clk = 1, din = 0, q = 0
#               12clk = 1, din = 1, q = 1
#               16clk = 0, din = 0, q = 0
#               20clk = 0, din = 1, q = 0
#               24clk = 0, din = 0, q = 0
#               28clk = 0, din = 1, q = 0
```

Figure 9.29 Simulation results of running the test bench for the UDP of Figure 9.28.

Example 9.15 A UDP for an Edge-Triggered Flip-Flop

Figure 9.30 shows the UDP definition of a positive edge-triggered flip-flop with a clear facility. In the table, (01) signifies the 0-to-1 transition edge of the clk – that is, its positive edge. Other edge transitions too can be interpreted in a similar manner. The simulation results are shown in Figure 9.31. From the simulation results, one can see that as long as the Clear input is low, data input is latched in at the positive going edge of the clock. But if the Clear input is high, its effect prevails and the flip-flop output remains low and does not respond to changes in the input data line.

```
primitive dff_pos(q,din,clk,clr);
output q;
input din,clk,clr;
reg q;
//initial q = 1'b0;

table
//      din     clk     clr     qp      qn      Whatever be the present
        0       (01)    0:      ?:      0;      // state of the output, at the
        1       (01)    0:      ?:      1;      // positive edge of clk input
        ?       (10)    0:      ?:      -;      // value is latched and
       (??)      ?      0:      ?:      -;      // output made equal to
        ?       ?       1:      ?:      0;      // that if clr = 0.  IF clr=1,
        ?       ?       *:      ?:      0;      // q .is made 0.
endtable
endprimitive
module dff_pos_tst;
wire q;
reg din,clk,clr;
dff_pos ll(q,din,clk,clr);
initial
begin
          clr=1'b0;din=1'b0;clk=1'b0;#3din=1'b1;
          repeat (2)begin #4 din=1'b1; #4 din=1'b0; end
          clr=1'b1;repeat (2) begin#4 din=1'b1; #4 din=1'b0; end
          $stop;
end
always   #2 clk=~clk;
initial $monitor($time ,"clr=%b, clk = %b, din = %b, q = %b ",clr,clk,din,q);
endmodule
```

Figure 9.30 An UDP for an edge-triggered flip-flop with clear facility: A test bench is also included in the figure.

```
#                0clr=0, clk = 0, din = 0, q = 0
#                2clr=0, clk = 1, din = 0, q = 0
#                3clr=0, clk = 1, din = 1, q = 0
#                4clr=0, clk = 0, din = 1, q = 0
#                6clr=0, clk = 1, din = 1, q = 1
#                8clr=0, clk = 0, din = 1, q = 1
#               10clr=0, clk = 1, din = 1, q = 1
#               11clr=0, clk = 1, din = 0, q = 1
#               12clr=0, clk = 0, din = 0, q = 1
#               14clr=0, clk = 1, din = 0, q = 0
#               15clr=0, clk = 1, din = 1, q = 0
#               16clr=0, clk = 0, din = 1, q = 0
#               18clr=0, clk = 1, din = 1, q = 1
#               19clr=1, clk = 1, din = 0, q = 0
#               20clr=1, clk = 0, din = 0, q = 0
#               22clr=1, clk = 1, din = 0, q = 0
#               23clr=1, clk = 1, din = 1, q = 0
#               24clr=1, clk = 0, din = 1, q = 0
#               26clr=1, clk = 1, din = 1, q = 0
#               27clr=1, clk = 1, din = 0, q = 0
#               28clr=1, clk = 0, din = 0, q = 0
#               30clr=1, clk = 1, din = 0, q = 0
#               31clr=1, clk = 1, din = 1, q = 0
#               32clr=1, clk = 0, din = 1, q = 0
#               34clr=1, clk = 1, din = 1, q = 0
```

Figure 9.31 Simulation results of running the test bench for the UDP of Figure 9.30.

There can be situations where an edge sensitive entry in a UDP table clashes with a level-sensitive entry. In such situations of conflict, the level-sensitive entry dominates and decides the next state. The UDP in Figure 9.29 is sensitive to the level changes in one input (clr) and the edge in the other (clk). One can also have UDPs sensitive only to the edges in the inputs.

Observations:

- Only one edge transition can be specified in one line of the UDP definition. All other inputs are to be defined as state levels.
- If one edge of an input is used to specify a transition in the output, the output transition has to be defined for all possible edges of all the inputs.

A sequential UDP specifies the next state in terms of the present state and inputs. If necessary, one can specify an initial state and avoid ambiguity in

operation at start. The initial declaration can be used here. Such an initial statement has to be a single procedural assignment. It can assign a 1(1'**b**1), a 0(1'**b**0), or an **x** value to the output **reg** of the UDP.

9.4.6 Sequential UDPs and Tasks

Sequential UDPs and tasks are functionally similar. Tasks are defined inside modules and used inside the module of definition. They are not accessible to other modules. In contrast, sequential UDPs are like other primitives and modules. They can be instantiated in any other module of a design.

9.4.7 UDP Instantiation with Delays

Outputs of UDPs also can take on values with time delays. The delays can be specified separately for the rising and falling transitions on the output. For example, an instantiation as

```
udp_and_b # (1, 2) g1(out, in1, in2);
```

can be used to instantiate the UDF of Figure 9.25 for carry output generation. Here the output transition to 1 (rising edge) takes effect with a time delay of 1 ns. The output transition to 0 (falling edge) takes effect with a time delay of 2 ns. If only one time delay were specified, the same holds good for the rising as well as the falling edges of the output transition.

9.4.8 Vector-Type Instantiation of UDP

UDP definitions are scalar in nature. They can be used with vectors with proper declarations. For example, the full-adder module fa in Figure 9.26 can be instantiated as an 8-bit vector to form an 8-bit adder. The instantiation statement can be

```
fa  [7:0] aa(co, s,  a,  b, {co[6:0],1'b0});
```

s (sum), co (carry output), a (first input), and b(second input) are all 8-bit vectors here. The vector type of instantiation makes the design description compact; however, it may not be supported by some simulators.

9.5 EXERCISES

1. Define half-adder and full-adder as tasks and prepare a 32-bit adder using them. Test it through a suitable test bench.
2. Form a UDP for an A-O-I gate and test it through a test bench.

3. Form a UDP for a 3-to-1 mux and test it through a test bench.
4. b0, b1, and b2 represent the three bits of a mod-8 counter. The counter is to count at the positive edge of a clock input. Form UDPs for b0, b1, and b2; instantiate them in a module to form a counter. Test the counter using a test bench.
5. A 3-bit number is to advance through the following cyclic sequence:

   0 – 3 – 5 – 4 – 1 – 0 – 3. . .

   Form UDPs for the 3 bits; form the sequencer module by instantiating the UDPs. Test the module through a test bench.
6. Form a microcontroller core as follows:
   - Have a set of 4 registers designated r1, r2, r3, and r4.
   - Define a set of 6 algebraic / logic operations – Add, 1's complement, NAND, EXOR, left shift, and right shift
   - Have an 8-bit instruction opcode as *ssddpaaa*. Here *ss*, *dd* and *aaa* specify the source address, the destination address and the 3-bit code for the algebraic/logic opreration, respectively. P is a single-bit mode selector – if $p = 0$, data are to be transferred from source to the destination; if $p = 1$, the algebraic/logic operation is to be done.
   - Define each of the operations above as a function or as a task.

   Realize the ALU functions as UDPs. Realize the whole module using the **`case`** statement. For example, 01111101 stands for taking data from r1 and r3, adding them and putting the result in r1. Use r1 to store the result. Have a separate status register with carry bit and zero bit: set them whenever necessary. Write a test bench for the microcontroller, and test each of the instructions and instruction sequences.
7. Consider Figure 9.12: Shift the statement r[32] = c[32]; ahead by one line. Include a **`$display`** statement in both cases: Simulate the test bench. Explain any difference.

10

SWITCH LEVEL MODELING

10.1 INTRODUCTION

In today's environment the MOS transistor is the basic element around which a VLSI is built. Designers familiar with logic gates and their configurations at the circuit level may choose to do their designs using MOS transistors. Verilog has the provision to do the design description at the switch level using such MOS transistors, which is the theme of the present chapter. Switch level modeling forms the basic level of modeling digital circuits. The switches are available as primitives in Verilog; they are central to design description at this level. Basic gates can be defined in terms of such switches. By repeated and successive instantiation of such switches, more involved circuits can be modeled – on the same lines as was done with the gate level primitives in Chapters 4 and 5.

Designers familiar with logic gates, digital functional blocks, and their interplay can successfully carry out a complete VLSI design without any involvement at the switch level. Hence the switch level design was deferred to the present chapter.

10.2 BASIC TRANSISTOR SWITCHES

Consider an NMOS transistor of the depletion type. When used in a digital circuit, it can be in one of three modes:

- $V_G < V_S$ where V_G and V_S are the gate and source voltages with respect to the drain: The transistor is OFF and offers very high impedance across the source and the drain. It is in the *z* state.
- $V_G \cong V_S$: The transistor is in the active region. It presents a resistance between the source and the drain. The value depends on the technology. Such a resistive state of the transistor can be modeled in Verilog. A transistor in this mode can be represented as a resistance in Verilog – as **`pull1`** or **`pull0`** depending on whether the drain is connected to **`supply1`** or source is connected to **`supply0`**.

Design Through Verilog HDL. T. R. Padmanabhan and B. Bala Tripura Sundari

ISBN: 0-471-44148-1

- $V_G > V_S$: The transistor is fully turned on. It presents very low resistance (~0 Ω) between the source and drain.
- An enhanced-mode NMOS transistor also has the above three modes of operation. It is OFF when $V_G \cong V_S$. It is moderately ON or in the active region when V_G is slightly greater than V_S, representing a resistive (**pull1** or **pull0**) mode of operation. When V_G is sufficiently greater than V_S, the transistor is in the on state representing very low (~0 Ω) resistance. Similar modes are possible for the PMOS transistor also. The modes and the voltage levels for each are summarized in Table 10.1.

The table is more for information and not of direct relevance to design description in Verilog. Whenever a switch primitive is present in a design, necessary biasing will be done automatically. The designer need not worry about it – at least at this stage.

10.2.1 Basic Switch Primitives

Different switch primitives are available in Verilog. Consider an **nmos** switch. A typical instantiation has the form

nmos (out, in, control);

nmos – a keyword – represents an NMOS transistor functioning as a switch. The switch has three terminals (see Figure 10.1) – in, out, and control. When the control input is at 1 (high) state, the switch is on. It connects the input lead to the output side and offers zero impedance. When the control input is low, the switch is OFF and output is left floating (**z** state). If the control is in the **z** or the **x** state, output may take corresponding values. Table 10.2 summarizes the input / output combinations. In the table the symbol "**L**" stands for 0 or **z** state. The symbol **H** stands for the 1 or **z** state.

The keyword **pmos** represents a PMOS transistor functioning as a switch. The PMOS switch has three terminals (see Figure 10.2). A typical instantiation of the switch has the form

Table 10.1 Operating voltages for different modes of operation of MOS switches

Mode		NMOS		PMOS	
		Depletion	Enhancement	Depletion	Enhancement
$V_D - V_S$ for normal operation (Range: 1.5V to 5V)		Positive	Positive	Negative	Negative
Range of $V_G - V_S$ for	OFF (**z**) state	Negative	≅ 0	Positive	Positive
	Resistive state (**pull1**, **pull0**)	≅ 0	Mildly positive	≅ 0	Mildly negative
	ON state (0Ω)	Positive	Fully positive	Negative	Fully negative

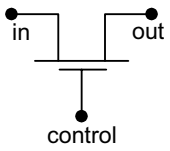

Figure 10.1 An NMOS switch with terminals.

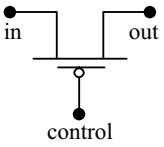

Figure 10.2 A PMOS switch with terminals.

pmos (out, in, control);

When the control is at 1 (high) state, the switch is off. Output is left floating. When control is at 0 (low) state, the switch is on, input is connected to output, and output is at the same state as input. For other input values, output is at other values. The output values for all possible input and control values are shown in Table 10.3. The symbols **L** and **H** have the same significance as in Table 10.2.

Observations:

- When in the on state, the switch makes its output available at the same strength as the input. There is only one exception to it: When the input is of strength **supply**, the output is of strength **strong**. It is true of **supply1** as well as **supply0**.
- When instantiating an **nmos** or a **pmos** switch, a name can be assigned to the switch. But the name is not essential. (The same is true of the other primitives discussed in the following sections as well.)
- The **nmos** and **pmos** switches function as unidirectional switches.

Table 10.2 Output values of an nmos switch for different values of signal and control inputs

		Control (input)			
		0	1	X	z
(Data) input	0	Z	0	L	L
	1	Z	1	H	H
	X	Z	X	X	X
	z	z	z	z	z

Table 10.3 Output values of a pmos switch for different values of signal and control inputs

		Control (input)			
		0	1	X	z
(Data) input	0	Z	0	L	L
	1	Z	1	H	H
	X	Z	X	X	X
	z	z	z	z	z

10.2.2 Resistive Switches

nmos and **pmos** represent switches of low impedance in the on-state. **rnmos** and **rpmos** represent the resistive counterparts of these respectively. Typical instantiations have the form

rnmos (output1, input1, control1);
rpmos (output2, input2, control2);

With **rnmos** if the control1 input is at 1 (high) state, the switch is ON and functions as a definite resistance. It connects input1 to output1 through a resistance. When control1 is at the 0 (low) state, the switch is OFF and leaves output1 floating. The set of output values for all combinations of input1 and control1 values remain identical to those of the **nmos** switch given in Table 10.2.

The **rpmos** switch is ON when control2 is at 0 (low) state. It inserts a definite resistance between the input and the output signals but retains the signal value. The output values for different input values remain identical to those in Table 10.3 for the **pmos** switch.

Observations:

- Because **rpmos** and **rnmos** are resistive switches, they reduce the signal strength when in the on state. The reduced strength is mostly one level below the original strength. The only exceptions are **small** and **hi-z**. For these the strength and the state remain unaltered (see Table 10.4).
- The **rpmos** and **rnmos** switches function as unidirectional switches; the signal flow is from the input to the output side.

Table 10.4 Output-side strength levels for different input-side strength values of rnmos, rpmos, and rcmos switches

Input strength	Output strength
Supply – drive	Pull – drive
Strong – drive	Pull – drive
Pull – drive	Weak – drive
Weak – drive	Medium – capacitive
Large – capacitive	Medium – capacitive
Medium – capacitive	Small – capacitive
Small – capacitive	Small – capacitive
High impedance	High impedance

10.2.3 `pullup` and `pulldown`

A MOS transistor functions as a resistive element when in the active state. Realization of resistance in this form takes less silicon area in the IC as compared to a resistance realized directly. **`pullup`** and **`pulldown`** represent such resistive elements. A typical instantiation here has the form

```
pullup (x);
```

Here the net `x` is pulled up to the **`supply1`** through a resistance. Similarly, the instantiation

```
pulldown(y);
```

pulls `y` down to the **`supply0`** level through a resistance. The **`pullup`** and **`pulldown`** primitives can be used as loads for switches or to connect the unused input ports to V_{CC} or GND, respectively. They can also form loads of switches in logic circuits. The default strengths for **`pullup`** and **`pulldown`** are **`pull1`** and **`pull0`** respectively. One can also specify strength values for the respective nets. For example,

```
pullup (strong1) (x)
```

specifies a resistive **`pullup`** of net `x` to **`supply1`**. One can also assign names to the **`pullup`** and **`pulldown`** primitives. Thus

```
pullup (strong1) rs(x)
```

represents an instantiation of **`pullup`** designated `rs` having strength **`strong1`**.

Difference between **`tri`** and **`pullup`** or **`pulldown`** is to be understood clearly. **`pullup`** is a functional element; it represents a resistive connection to **`supply1`**. In contrast **`tri1`** is a type of net; in the absence of an assignment, it remains connected to **`supply1`**. A similar difference exists between **`pulldown`** and **`tri0`**. The example below brings out the differences.

Example 10.1

Figure 10.3 shows two simple circuits that are apparently identical: Figure 10.3 (a) has the net `o1` declared as tr1 and is pulled up in case it is left open. With the circuit in Figure 10.3(b), `o2` is a wire type of net; it has a resistive element connecting it to **`supply1`**. Figure 10.4 shows a module incorporating both the circuits and a test bench for them. Note that the module instantiates the primitive **`bufif1`** for the controlled buffer (discussed in Chapter 4). The test bench has specific assignments to the two input signals which bring out the difference in contention resolution. For identical input signal values, the outputs `o1` and `o2` can differ in certain cases.

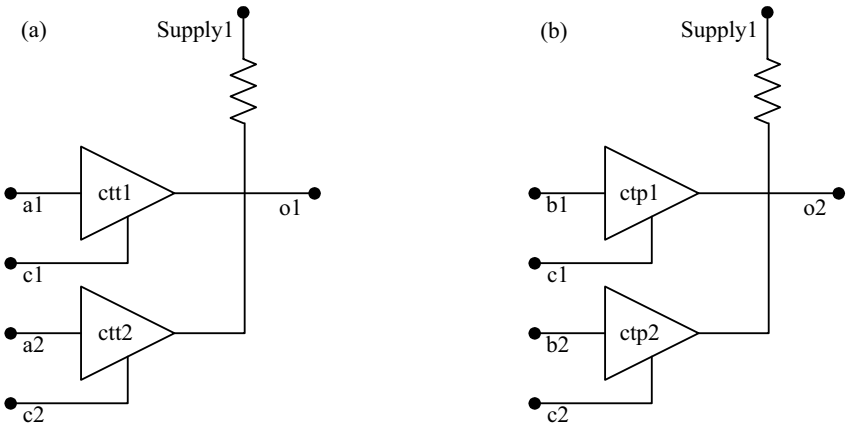

Figure 10.3 Two circuits to demonstrate the difference between **`tri1`** and **`pullup`.**

- At t = 0, all the inputs are at **`x`** state; o1 is also at **`x`** state: But o2=1 because of the **`pullup`**.
- At t = 1, all the switches are turned off; o1 and o2 are at 1 state.
- At t = 2, all the switches are on; the outputs follow the inputs and both of them are at 1 state.
- At t = 3, ctt1 & ctp1 are on; ctt2 & ctp2 are off; o1=a1=1 but o2=1 since its value is the result of contention resolution between b1 at **`weak0`** & the stronger pp at **`pull1`**.
- At t = 4, all switches are on; all inputs are at 0; o1=0 but o2=1 since the stronger pull1 prevails over the weaker 0's of b1 & b2.

```
module swt_aa (o1,o2, a1,a2,b1,b2,c1,c2);
output o1,o2; input a1,a2,b1,b2,c1,c2;
wire o2; tri1 o1;
bufif1  ctt1(o1,a1,c1), ctt2(o1,a2,c2);
bufif1 (weak1, weak0) ctp1(o2,b1,c1), ctp2(o2,b2,c2);
pullup pp(o2);
endmodule

module swt_aa_tb;
reg [5:0] rx; wire o1,o2, a1,a2,b1,b2,c1,c2;
assign {a1,a2,b1,b2,c1,c2} = rx;
swt_aa aa(o1,o2, a1,a2,b1,b2,c1,c2);
initial begin
```

continued

continued

```
#1      rx=6'o00;
#1      rx=6'o77;
#1      rx=6'o42;
#1      rx=6'o03;
#1      $stop;
    end
initial $monitor("time=%0d, c1=%b, o1=%b, a1=%b, a2=%b, c2=%b, o2=%b,
b1=%b, b2=%b",$time, c1,o1,a1,a2,c2,o2,b1,b2);
endmodule
```

#t	c1	o1	a1	a2	c2	o2	b1	b2
#0	x	x	x	x	x	1	x	x
#1	0	1	0	0	0	1	0	0
#2	1	1	1	1	1	1	1	1
#3	1	1	1	0	0	1	0	0
#4	1	0	0	0	1	1	0	0

Figure 10.4 Design module for the circuits of Figure 1.3 and its test bench.

Example 10.2 CMOS inverter

A CMOS inverter is formed by connecting an **`nmos`** and a **`pmos`** switch in series across the supply (see Figure 10.5). The output terminals are joined together to form the common output. Similarly, the input is used as the common control input to both the switches. Referring to the figure, we can see the following:

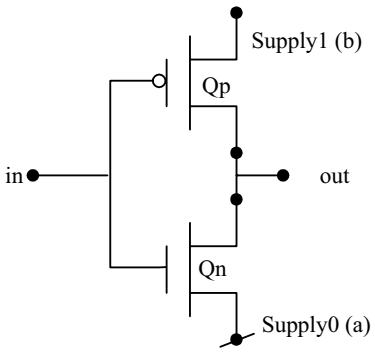

Figure 10.5 A CMOS inverter formed by connecting an NMOS and a PMOS set of transistors in series across the supply.

- When the input is low (0 V), transistor Q_n is off. But Q_p is on. **supply1** is connected to the output. Hence the output is high.
- When the input is high (5 V), transistor Q_p is off. But Q_n is on. **supply0** is connected to the output. Hence the output is low.

The inverter operation is clear from the above. The design description for the corresponding CMOS inverter is shown in Figure 10.6. The leads a and b are declared as nets – **supply0** and **supply1** respectively; *i.e*., they are connected to ground and V_{CC} respectively. The two instantiations together describe the inverter operation.

Observations:

- Under steady-state operation of the CMOS inverter, only Q_p or Q_n is ON at a time. Hence the inverter does not draw any quiescent current from the supply. Current is drawn only to charge the internal capacitor associated with the transistors during the transition.
- The input and output sides of the switches refer to the signal flow directions and not that of the current flow. Thus for the NMOS switch under the ON condition, current flows from out to **supply0**. But the signal from a (at **supply0**) is made available at out.

Example 10.3 CMOS NOR gate

A CMOS nor gate with two inputs is shown in Figure 10.7. It employs four transistors.

- When only in1 is high, Q_{n1} is ON pulling out to **suppy0**. Output is zero. Q_{p2} is also on. But since in2 is low, Q_{p1} is off. Hence no current can be drawn from **supply1.**
- When only in2 is high, Q_{n2} is ON pulling out to **suppy0**. Output is zero. Q_{p1} is also on. But since in1 is low, Q_{p2} is off. Hence no current can be drawn from **supply1**.

```
module inv (in, out );
output out;
input in;
supply0 a;
supply1 b;
nmos (out, a, in );
pmos (out, b, in);
endmodule
```

Figure 10.6 design description of a CMOS inverter formed by connecting an NMOS transistor and a PMOS transistor in series.

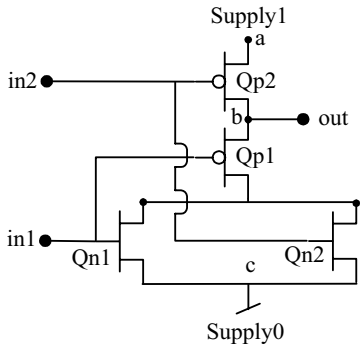

Figure 10.7 A 2 input CMOS NOR gate.

- When both in1 and in2 are low, Q_{n1} and Q_{n2} are OFF. Q_{p1} and Q_{p2} are ON and out is connected to **supply1**. But no current is drawn from the supply.

When both in1 and in2 are high, Q_{n1} and Q_{n2} are on. Out is grounded at c. Since Q_{p1} and Q_{p2} are off, no current is drawn from supply1. The design description for the NOR gate is shown in Figure 10.8. It has four instantiations – two of **pmos** and two of **nmos**, respectively.

```
module npnor_2(out, in1, in2 );
output out;
input in1, in2;
supply1 a;
supply0 c;
wire b;
pmos(b, a, in2), (out, b, in1);
nmos (out, c, in1), (out, c, in2) ;
endmodule
```

Figure 10.8 design description of a CMOS NOR gate.

Observations:

- A three-input NOR gate has three NMOS transistors in parallel on the ground side and three PMOS transistors in series on the V_{CC} side. Although the number of inputs can be increased in this manner, circuit operation is not satisfactory for more than two or three inputs [Bogart].
- NAND gate is formed by connecting the NMOS transistors in series on the ground side and the PMOS transistors in parallel on the V_{CC} side (NAND is the dual of NOR).

- Because NAND and NOR are universal gates, all other logic gates can be realized in terms of them.

Example 10.4 NMOS Inverter with Pull up Load

Figure 10.9 shows an NMOS inverter. Q_1 is the NMOS transistor. Q_2 properly biased, forms an active resistance and is the load on Q_1. The design description for the inverter is shown in Figure 10.10. When in = 0, Q_1 is OFF and out is pulled up and is at state 1. When in = 1, Q_1 is ON and out is pulled down to 0. A test bench and the results of simulating the test bench are also included in the figure.

Observations:

- When Q_1 is ON (in = 1), the gate has a standing current; contrast this with CMOS inverter in Example 10.2, where the quiescent current is always zero. The output is available as **`strong0`**.
- When Q_1 is OFF, the standing current is zero. But the output is available as **`pull1`**. Thus there is a difference in the strengths of the two states.
- If necessary, a different strength value can be assigned to **`pullup`**.

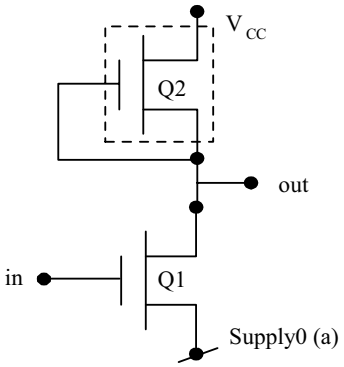

Figure 10.9 An NMOS inverter with an active pull up load.

```
module NMOSinv(out,in);
output out;input in;supply0 a;
pullup (out);
nmos(out,a,in);
endmodule
module tst_nm_in();
reg in;wire out;
NMOSinv nmv(out,in);
initial
in =1'b1;
always
#3 in =~in;
initial $monitor($time , " in = %b, output = %b ",in,out);
initial #30 $stop;
endmodule
```

```
#                0 in = 1, output = 0
#                3 in = 0, output = 1
#                6 in = 1, output = 0
#                9 in = 0, output = 1
#               12 in = 1, output = 0
#               15 in = 0, output = 1
#               18 in = 1, output = 0
#               21 in = 0, output = 1
#               24 in = 1, output = 0
#               27 in = 0, output = 1
```

Figure 10.10 Design description of an NMOS inverter gate: A test bench for the inverter and the simulation results are also shown in the figure.

Example 10.5 An NMOS Three Input NOR Gate

Figure 10.11 shows a three-input NMOS NOR gate with Q4 – properly biased – forming a resistive pullup load. Output b is high when all the inputs – in1, in2 and in3 are low – keeping the respective mos transistors – Q1, Q2, and Q3 – off. If any one of the three inputs goes high, the corresponding NMOS transistor turns ON and the output b is pulled down to zero. When output is in 1 state, it has strength **`pull1`**. When in the zero state, it has strength **`strong0`**. The design description for the gate is shown in Figure 10.12. Simulation results are given in Figure 10.13.

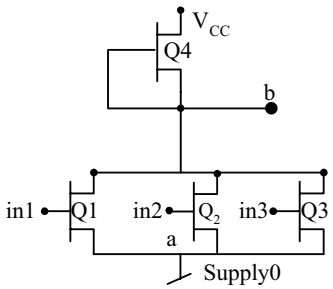

Figure 10.11 An NMOS NOR gate with active pull up.

```
module nor3NMOS(in1,in2,in3,b);
output b;
input in1,in2,in3;
supply0 a; wire b;
nmos(b,a,in1),(b,a,in2),(b,a,in3);
pullup(b);
endmodule

module tst_nor3NMOS();
reg in1,in2,in3;wire b;
nor3NMOS nn(in1,in2,in3,b);
initial
begin
in1=1'b1;in2=1'b1;in3=1'b1;
end
always #2 in1=~in1;
always #3 in2=~in2;
always #5 in3=~in3;
initial $monitor($time , "in1 = %b , in2 = %b , in3 = %b , output = %b
",in1,in2,in3,b);
initial #24 $stop;
endmodule
module  (b, in1, in2, in3 );
output b;
input in1, in2, in3;
supply0 a;
wire b;
nmos (b, a, in1), (b, a, in2), (b, a, in3) ;
pullup (b ) ;
endmodule
```

Figure 10.12 Design description of an NMOS NOR gate with active pull up.

```
#          0in1 = 1 , in2 = 1 , in3 = 1 , output = 0
#          2in1 = 0 , in2 = 1 , in3 = 1 , output = 0
#          3in1 = 0 , in2 = 0 , in3 = 1 , output = 0
#          4in1 = 1 , in2 = 0 , in3 = 1 , output = 0
#          5in1 = 1 , in2 = 0 , in3 = 0 , output = 0
#          6in1 = 0 , in2 = 1 , in3 = 0 , output = 0
#          8in1 = 1 , in2 = 1 , in3 = 0 , output = 0
#          9in1 = 1 , in2 = 0 , in3 = 0 , output = 0
#         10in1 = 0 , in2 = 0 , in3 = 1 , output = 0
#         12in1 = 1 , in2 = 1 , in3 = 1 , output = 0
#         14in1 = 0 , in2 = 1 , in3 = 1 , output = 0
#         15in1 = 0 , in2 = 0 , in3 = 0 , output = 1
#         16in1 = 1 , in2 = 0 , in3 = 0 , output = 0
#         18in1 = 0 , in2 = 1 , in3 = 0 , output = 0
#         20in1 = 1 , in2 = 1 , in3 = 1 , output = 0
#         21in1 = 1 , in2 = 0 , in3 = 1 , output = 0
#         22in1 = 0 , in2 = 0 , in3 = 1 , output = 0
```

Figure 10.13 Results of running the test bench in Figure 10.12.

Observations:

- When any of the inputs is high, the corresponding transistor is ON and the gate has a standing current. The standing current is zero only when all the three inputs are at zero state and Q_1, Q_2, and Q_3 are off. The standing current makes the power dissipation in the device much higher than that for its CMOS counterpart.
- Adding transistors in parallel can increase the number of inputs.
- NAND gate can be formed by connecting the NMOS transistors controlled by the inputs in series. However, NOR remains the preferred universal gate element with NMOS logic.
- One can use a **`pmos`**-type switch at the top with a **`pulldown`** type of load to the ground. It forms a PMOS inverter (see Figure 10.14). The different logic gates of PMOS technology can be built with it. Here again, due to the standing current, the power consumption of the device will be much higher than that of its CMOS counterpart.
- For any logic function the **`nmos`** or the **`pmos`** gate uses a much smaller number of transistors than does the CMOS gate. Despite this CMOS logic family stands out due to two reasons:-
  - Lowest power consumption
  - Uniformity in the element patterns

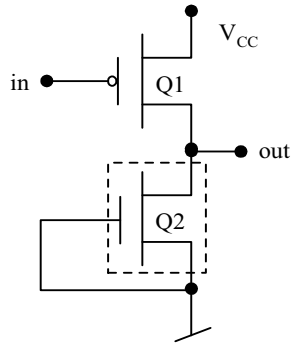

Figure 10.14 A PMOS inverter with active pull down load.

The advantages of CMOS technology often far outweigh the apparent complexity of the larger number of devices required on a per gate basis. Hence CMOS has proved to be much more popular.

10.3 CMOS SWITCH

A CMOS switch is formed by connecting a PMOS and an NMOS switch in parallel – the input leads are connected together on the one side and the output leads are connected together on the other side. Figure 10.15 shows the switch so formed. It has two control inputs:

- N_control turns ON the NMOS transistor and keeps it ON when it is in the 1 state.
- P_control turns ON the PMOS transistor and keeps it ON when it is in the 0 state.

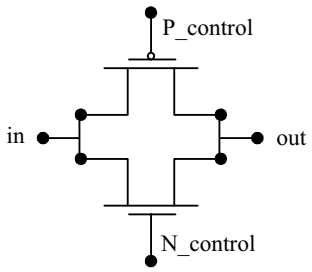

Figure 10.15 A CMOS switch formed by connecting a PMOS transistor and an NMOS transistor in parallel.

The CMOS switch is instantiated as shown below.
cmos csw (out, in, N_control, P_control);

Significance of the different terms is as follows:

- **cmos :** The keyword for the switch instantiation
- csw: Name assigned to the switch in the instantiation
- out: Name assigned to the output variable in the instantiation
- in: Name assigned to the input variable in the instantiation
- N_control: Name assigned to the control variable of the NMOS transistor in the instantiation
- P_control: Name assigned to the control variable of the PMOS transistor in the instantiation

Example 10.6 CMOS Switch – 1

Being a parallel combination of a PMOS and an NMOS switch, the CMOS switch can be realized by instantiating these to form a parallel switch. Design description of such a switch is shown in Figure 10.16 along with a test bench. The controls for the NMOS and the PMOS sides are separate in the primitive. The (partial) simulation results are shown in Figure 10.17.

```
module CMOSsw(out,in,n_ctr,p_ctr);
output out; input  in,n_ctr,p_ctr;
nmos gn(out,in,n_ctr);
pmos gp(out,in,p_ctr);
endmodule

module tst_CMOSsw();
reg in,n_ctr,p_ctr; wire out;
CMOSsw  cmsw(out,in,n_ctr,p_ctr);
initial begin in=1'b0;n_ctr=1'b1;p_ctr=~n_ctr; end
always #5 in =~in;
always begin #3 n_ctr=~n_ctr; #0p_ctr =~n_ctr; end
initial $monitor($time , "in = %b , n_ctr = %b , p_ctr = %b , output = %b
",in,n_ctr,p_ctr,out);
initial #39 $stop;
endmodule
```

Figure 10.16 Design description of a CMOS switch formed by paralleling a pair of NMOS and PMOS switches.

```
#          0in = 0 , n_ctr = 1 , p_ctr = 0 , output = 0
#          3in = 0 , n_ctr = 0 , p_ctr = 1 , output = z
#          5in = 1 , n_ctr = 0 , p_ctr = 1 , output = z
#          6in = 1 , n_ctr = 1 , p_ctr = 0 , output = 1
#          9in = 1 , n_ctr = 0 , p_ctr = 1 , output = z
#         10in = 0 , n_ctr = 0 , p_ctr = 1 , output = z
#         12in = 0 , n_ctr = 1 , p_ctr = 0 , output = 0
```

Figure 10.17 Partial results of simulating the test bench for the CMOS switch in Figure 10.16.

Example 10.7 CMOS Switch – 2

In normal use of a CMOS switch, the same control line drives the gates of the PMOS and the NMOS switches (as shown in Figure 10.18). With this change the switch becomes more compact for description as well. The module for the compact switch is shown in Figure 10.19; the figure also includes a test bench for it. The design module uses an instantiation of the NOT gate for generating P_control from con – the control input. The (partial) simulation results are in Figure 10.20.

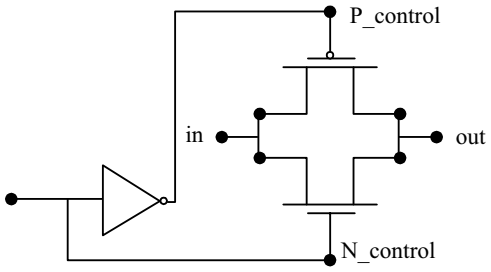

Figure 10.18 A CMOS switch with a single control line.

```
module CMOSsw1(out,in,con);
output out; input  in,con; wire p_ctr;
not gn(p_ctr,con);
cmos gc(out,in,con,p-ctr);
endmodule
```

continued

continued

```
module tst_CMOSsw1();
reg in,con; wire out;
CMOSsw1  cmsw(out,in,con);
initial begin in=1'b0;con=1'b1; end
always #5 in =~in;
always #3 con=~con;
initial $monitor($time , "in = %b , con = %b ,  output = %b " ,in,con,out);
initial #40 $stop;
endmodule
```

Figure 10.19 Design description of a CMOS switch with a single control input.

```
#                0in = 0 , con = 1 ,  output = 0
#                3in = 0 , con = 0 ,  output = x
#                5in = 1 , con = 0 ,  output = x
#                6in = 1 , con = 1 ,  output = 1
#                9in = 1 , con = 0 ,  output = x
#               10in = 0 , con = 0 ,  output = x
#               12in = 0 , con = 1 ,  output = 0
```

Figure 10.20 Partial results of simulating the test bench for the CMOS switch in Figure 10. 19.

Example 10.8 A RAM Cell

Figure 10.21 shows a basic ram cell with facilities for writing data, storing data, and reading data. When switch sw2 is on, qb – the output of inverter g1 – forms the input to the inverter g2 and vice versa. The g1-g2 combination functions as a latch and freezes the last state entry before sw2 turns on. The step-by-step function of the cell is as follows (see the waveforms in Figure 10.22):

- When wsb (write/store) is high, switch sw1 is ON, and switch sw2 OFF. With sw1 on, input Din is connected to the input of gate g1 and remains so connected.
- When wsb goes low, din is isolated, since sw1 is OFF. But sw2 is ON and the data remains latched in the latch formed by g1-g2. In other words the data Din is stored in the RAM cell formed by g1-g2.
- When RD (Read) goes active (=1), the latched state is available as output Do. Reading is normally done when the latch is in the stored state.

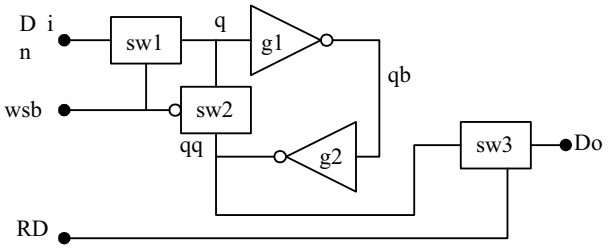

Figure 10.21 Basic RAM cell in block diagram form.

The design description for the ram cell as well as a test bench for it is given in Figure 10.23. It instantiates a csw module which is a basic CMOS switch with a single control line. If necessary, the **not** gate can be separately defined as a CMOS gate module and instantiated. Note that the output of gate g2 – qq- has been declared as a **trireg** type of net. It is to ensure that the q2 output is stored when sw2 is OFF. It avoids any error during transition – that is, sw2 turning off with a delay compared to that of sw1. The (partial) simulation results are in Figure 10.24. A full-fledged memory can be built using the ram cell. The memory address decoders are to form the enable signals to the write and read control signals here.

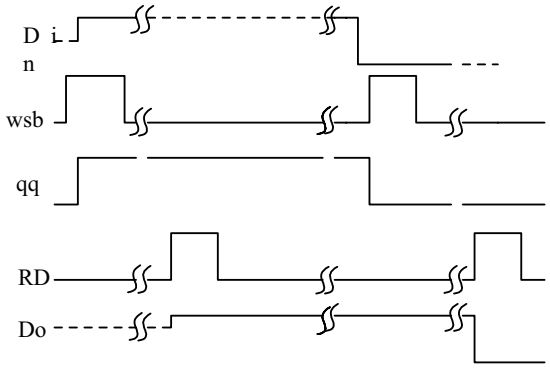

Figure 10.22 Waveforms of different signals in the operation of the basic RAM cell of Figure 10.21.

```
module ram_cell(do,din,wsb,rd);
output do; input din,wsb,rd; wire sb; wire q,qq; tri do;
csw sw1(q,din,wsb),sw2(q,qq,sb),sw3(do,q,rd);
not n1(sb,wsb),n2(qb,q),n3(qq,qb);
endmodule

module csw(out,in,n_ctr);
output out; input in,n_ctr; wire p_ctr;
assign p_ctr =~n_ctr;
cmos csw(out,in,n_ctr,p_ctr);
endmodule

module tst_ramcell();
reg din,wsb,rd; wire do;
ram_cell mc(do,din,wsb,rd);
initial begin din=1'b0;wsb=1'b0;rd=1'b0; end
always #10 din =~din;
always begin #3wsb=1'b1; #8wsb=1'b0; end
always begin #2 rd=1'b1; #5 rd =1'b0; end
initial $monitor ($time," rd= %b ,wsb = %b ,din = %b ,do = %b ",rd,wsb,din,do);
initial #40 $stop;
endmodule
```

Figure 10.23 Design description of a basic RAM cell.

```
#                0 rd= 0 ,wsb = 0 ,din = 0 ,do = z
#                2 rd= 1 ,wsb = 0 ,din = 0 ,do = x
#                3 rd= 1 ,wsb = 1 ,din = 0 ,do = 0
#                7 rd= 0 ,wsb = 1 ,din = 0 ,do = z
#                9 rd= 1 ,wsb = 1 ,din = 0 ,do = 0
#               10 rd= 1 ,wsb = 1 ,din = 1 ,do = 1
#               11 rd= 1 ,wsb = 0 ,din = 1 ,do = 1
#               14 rd= 0 ,wsb = 1 ,din = 1 ,do = z
#               16 rd= 1 ,wsb = 1 ,din = 1 ,do = 1
#               20 rd= 1 ,wsb = 1 ,din = 0 ,do = 0
#               21 rd= 0 ,wsb = 1 ,din = 0 ,do = z
```

Figure 10.24 Partial results of simulating the test bench for the CMOS switch in Figure 10.23.

Example 10.9 An Alternate RAM Cell Realization

Figure 10.25 shows an alternate and apparently simpler version of the ram cell (ram_1). The two inverters are connected permanently in a back-to-back fashion. Their output strength levels are **pull1** and **pull0**. Din can be of strength **strong**. Hence when the data write switch (sww) is turned ON, Din prevails and forces q to its own state. The condition is latched and remains so after switch sww is turned OFF. Another data can be written again by turning ON switch sww after making the new data available at Din. Data can be read out of the latch by turning on the switch – swr. It has the control line RD.

The module of the ram_1 cell is shown in Figure 10.26; the figure also includes a test bench. The design uses two instantiations of the **not** gate with strength **pull1** and **pull0**. The switches sww and swr are realized through instantiations of the CMOS switch modules csw. (Alternately, the same can be defined as a function inside the ram1 module and used as such.) Partial simulation results are shown in Figure 10.27. By adding address decoding and clock, the cell can be used as the basis for forming a full-fledged ram.

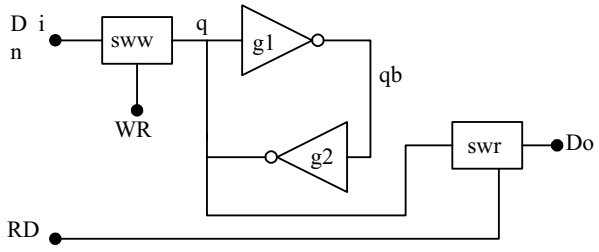

Figure 10.25 An alternate version of the RAM cell in block diagram form.

```
module ram1(do,din,wr,rd);
output do; input din,wr,rd; wire qb,q; tri do;
scw sww(q,din,wr),swr(do,q,rd);
not(pull1,pull0)n1(qb,q),n2(q,q);
endmodule

module scw(out,in,n_ctr);
output out; input in,n_ctr; wire p_ctr;
assign p_ctr =~n_ctr;
cmos sw(out,in,n_ctr,p_ctr);
endmodule
```

continued

continued

```
//test-bench
module tst_ram1();
reg din,wr,rd; wire do;
ram1 mm(do,din,wr,rd);
initial begin din=1'b0;wr=1'b0;rd=1'b0; end
always #10 din =~din;
always begin #3wr=1'b1;  #8wr=1'b0; end
always  begin #2 rd=1'b1; #5 rd =1'b0; end
initial $monitor ($time," rd= %b ,wr = %b ,din = %b ,do = %b ",rd,wr,din,do);
initial #40 $stop;
endmodule
```

Figure 10.26 Design description of the RAM cell of Figure 10.24.

```
#                0 rd= 0 ,wr = 0 ,din = 0 ,do = z
#                2 rd= 1 ,wr = 0 ,din = 0 ,do = x
#                3 rd= 1 ,wr = 1 ,din = 0 ,do = 0
#                7 rd= 0 ,wr = 1 ,din = 0 ,do = z
#                9 rd= 1 ,wr = 1 ,din = 0 ,do = 0
#               10 rd= 1 ,wr = 1 ,din = 1 ,do = 1
```

Figure 10.27 Partial results of simulating the test bench for the CMOS switch in Figure 10.26.

Example 10.10 A Dynamic Shift Register

Figure 10.28 shows three successive stages of a dynamic shift register. It is operated through a two-phase clock system – $\phi 1$ and $\phi 2$. Each stage has a CMOS inverter. Successive stages are given input through CMOS switches (sw1, sw2, *etc*.). $\phi 1$ and $\phi 2$ are symmetric clock waveforms in anti-phase. Two successive stages together form one storage element.

- When $\phi 2$ is ON AND $\phi 1$ is OFF. Din is input to stage 1 through switch swd. sw1 and sw3 are OFF and sw2 is ON. State of stage 2 (attained when $\phi 1$ was high last) is coupled as input to stage 3 through switch sw2, and stage 3 takes up the new state.
- In the next half cycle, $\phi 1$ is ON and $\phi 2$ is OFF. sw1 and sw3 are ON and sw2 is OFF. State of stage 1 (attained when $\phi 2$ was high last) is coupled as input to stage 2 through switch sw1 and Do takes up the new state from stage3 through sw3.

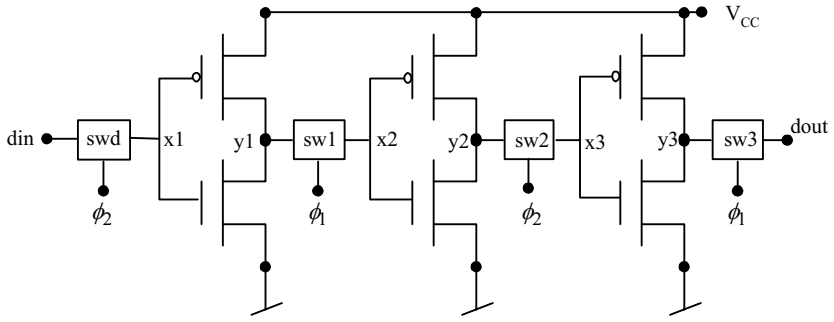

Figure 10.28 The basic functional unit of a dynamic shift register.

The data in the input line are latched and shifted right on successive clock cycles. The stages together act as a shift register stage. The design description of a shift-register module with a two-phase clock is shown in Figure 10.29 along with a test-bench for the same. The two-phase clock and switches are defined as functions. These are repeatedly called to realize the shift register. Figure 10.30 shows partial simulation results.

The shift register can be modified to suit a variety of needs:

- Dynamic logic incorporating NAND / NOR gates.
- Dynamic RAM with row and column select lines and refresh functions.
- A shift register to function as a right- or a left-shift-type shift register; a direction select bit can be used to alter the shift direction.

```
module shreg1(dout,din,phi1);
output dout;//tested ok on 22nd Non 2001
input din,phi1;
wire phi2;
trireg[3:0] x,y;
trireg dout;
assign phi2=~phi1;
cmos switch0(x[0],din,phi1,phi2), switch1(x[1],y[0],phi2,phi1),
switch2(x[2],y[1],phi1,phi2), switch3(x[3],y[2],phi2,phi1),
switch4(dout,y[3],phi1,phi2);
cell cc0(y[0],x[0]), cc1(y[1],x[1]), cc2(y[2],x[2]), cc3(y[3],x[3]);
endmodule

module cell(op,ip);
output op;
input ip;
```

continued

continued

```
supply1 pwr;
supply0 gnd;
nmos(op,gnd,ip);
pmos(op,pwr,ip);
endmodule

module tst_shreg1;
reg din,phi1;
wire dout;
shreg1 shr(dout,din,phi1);
initial {din,phi1}=2'B00;
always
begin
  #1       din=1'b1;   #2      din=1'b1;   #2      din=1'b0;
  #2       din=1'b0;   #2      din=1'b0;   #2      din=1'b1;
  #2       din=1'b1;
end
always # 2 phi1=~phi1;
initial $monitor($time," din= %b,    dout= %b,  phi1= %b", din,dout,phi1);
endmodule
```

Figure 10.29 Design description of the dynamic shift register of Figure 10.28.

```
#                0  din= 0,    dout= x,  phi1= 0
#                1  din= 1,    dout= x,  phi1= 0
#                2  din= 1,    dout= x,  phi1= 1
#                4  din= 1,    dout= x,  phi1= 0
#                5  din= 0,    dout= x,  phi1= 0
#                6  din= 0,    dout= x,  phi1= 1
#                8  din= 0,    dout= x,  phi1= 0
#               10  din= 0,    dout= 1,  phi1= 1
#               11  din= 1,    dout= 1,  phi1= 1
#               12  din= 1,    dout= 1,  phi1= 0
#               14  din= 1,    dout= 0,  phi1= 1
#               16  din= 1,    dout= 0,  phi1= 0
#               18  din= 0,    dout= 1,  phi1= 1
#               20  din= 0,    dout= 1,  phi1= 0
```

Figure 10.30 Partial results of running the test bench in Figure 10.29.

10.4 BI-DIRECTIONAL GATES

The gates discussed so far (**nmos**, **pmos**, **rnmos**, **rpmos**, **rcmos**) are all unidirectional gates. When turned ON, the gate establishes a connection and makes the signal at the input side available at the output side. Verilog has a set of primitives for bi-directional switches as well. They connect the nets on either side when ON and isolate them when OFF. The signal flow can be in either direction. None of the continuous-type assignments at higher levels dealt with so far has a functionality equivalent to the bi-directional gates. There are six types of bi-directional gates.

10.4.1 `tran` and `rtran`

The **tran** gate is a bi-directional gate of two ports. When instantiated, it connects the two ports directly. Thus the instantiation

tran (s1, s2);

connects the signal lines s1 and s2. Either line can be **input**, **inout** or **output**. **rtran** is the resistive counterpart of **tran**.

10.4.2 tranif1 and rtranif1

tranif1 is a bi-directional switch turned ON/OFF through a control line. It is in the ON-state when the control signal is at 1 (high) state. When the control line is at state 0 (low), the switch is in the OFF state. A typical instantiation has the form

tranif1 (s1, s2, c);

Here c is the control line. If c=1, s1 and s2 are connected and signal transmission can be in either direction. **rtranif1** is the resistive counterpart of **tranif1**. It is instantiated in an identical manner.

10.4.3 tranif0 and rtranif0

tranif0 and **rtranif0** are again bi-directional switches. The switch is OFF if the control line is in the 1 (high) state, and it is ON when the control line is in the 0 (low) state. A typical instantiation has the form

tranif0 (s1, s2, c);

With the above instantiation, if c = 0, s1 and s2 are connected and signal transmission can be in either direction. If c = 1, the switch is OFF and s1 and s2 are isolated from each other. **rtranif0** is the resistive counterpart of **tranif0**.

Observations:

- Any instantiation of a bi-directional switch of the above types can be given a name. But a name is not essential. It is true of the other switches also.
- With the bi-directional switches the signal on either side can be of **`input`**, **`output`**, or **`inout`** type. They can be nets or appearing as ports in the module. But the type declaration on the two sides has to be consistent.
- The connections to the bi-directional terminals of each of the bi-directional switches have to be scalars or individual bits of vectors and not vector themselves.
- In the above instantiation s1 can be an input port in a module. In that case, s2 has to be a net forming an input to another instantiated module or circuit block. s2 can be of **`output`** or **`inout`** type also. But it cannot be another input port.
  - s1 and s2 – both cannot be output ports.
  - s1 and s2 – both can be inout ports.
- With **`tran`**, **`tranif1`**, and **`tranif0`** bi-directional switches if the input signal has strength **`supply1`** (**`supply0`**), the output side signal has strength **`strong1`** (**`strong0`**). For all other strength values of the input signal, the strength value of the output side signal retains the strength of the input side signal.
- With **`rtran`**, **`rtranif1`** and **`rtranif0`** switches the output side signal strength is less than that of the input side signal. The strength reduction is on the lines shown in Table 10.4 for **`rnmos`**, **`rpmos`**, and **`rcmos`** switches.

Features of all the bi-directional switches are shown summarized in Table 10.5.

Example 10.11 Bus Switching

Figure 10.31 shows the circuit of a single-data line bus with the possibility of two-way data transfer; the module bus_tran in Figure 10.32 is the Verilog description of the circuit at the switch level. c is a **`tran`**-type switch with the possibility of connecting a and b. ar and br are registers which can be switched ON to the lines

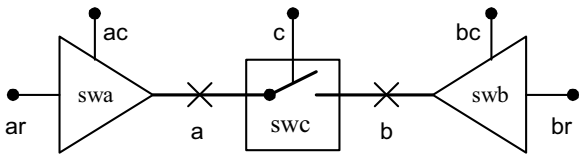

Figure 10.31 A circuit to demonstrate two-way signal transfer.

Table 10.5 Different bi-directional switches and their features

Type of Bi-directional switch	Typical instantiation	Condition to be ON	Remarks
2 port	**tran** (a, b);	Always ON (if instantiated)	Acts essentially as a buffer
	rtran (a, b);	– do –	Acts essentially as a buffer with reduction in the strength of the signal
3 port	**tranif1** (a, b, c);	ON if c = 1	Acts as a buffer if ON. Otherwise provides isolation
	tranif0 (a, b, c);	ON if c = 0	– do –
	rtranif1 (a, b, c);	ON if c = 1	Acts as a buffer if ON. Otherwise provides isolation; signal strength on the output side is lower than that on the input side
	rtranif0 (a, b, c);	ON if c = 0	– do –

```
module bus_tran(a,b,c);
inout  a,b; input c; wire a,b,c;
tranif1 gg (a,b,c);
endmodule

module bus_tst;
reg ar,br,ac,bc,c;wire a,b;
bufif1 swa(a,ar,ac), swb(b,br,bc);
bus_tran bs(a,b,c);
initial    begin
           $display("t\tar\tac\ta\tc\tb\tbc\tbr");
 #1        {ar,ac,c,bc,br}=5'b01100; repeat(3) #1 ar=~ar;
 #1        {ar,ac,c,bc,br}=5'b00110; repeat(3) #2 br=~br;
 #1        {ar,ac,c,bc,br}=5'b11010; repeat(3) #1 ar=~ar;
           repeat(3) #2 br=~br;
 #1        $stop;
           end
initial $monitor("%0d\t%b\t%b\t%b\t%b\t%b\t%b\t%b",$time,ar,ac,a,c,b,bc,br);
endmodule
```

Figure 10.32 Design and test modules for the circuit of Figure 10.31.

a and b. Two-way signal transmission is demonstrated through the test bench in the figure; the simulation results reproduced in Figure 10.33 bring out the following:

- Up to 4 ns, switch swa is ON, swb is OFF, and swc is ON. Data in ar – ar toggles 3 times and is available on a and b.
- During 5 ns to 11 ns, switch swa is OFF, swb is ON, and swc is ON. Data in br – br toggles 3 times and is available on b and a.
- During 1 2ns to 21 ns, switch swc is ON, swa and swb are OFF; a follows ar while b follows br.

#t	ar	ac	a	c	b	bc	br
#0	x	x	x	x	x	x	x
#1	0	1	0	1	0	0	0
#2	1	1	1	1	1	0	0
#3	0	1	0	1	0	0	0
#4	1	1	1	1	1	0	0
#5	0	0	0	1	0	1	0
#7	0	0	1	1	1	1	1
#9	0	0	0	1	0	1	0
#11	0	0	1	1	1	1	1
#12	1	1	1	0	0	1	0
#13	0	1	0	0	0	1	0
#14	1	1	1	0	0	1	0
#15	0	1	0	0	0	1	0
#17	0	1	0	0	1	1	1
#19	0	1	0	0	0	1	0
#21	0	1	0	0	1	1	1

Figure 10.33 Simulation results with the test bench for the circuit in Figure 10.31.

Example 10.12 Another RAM Cell

Figure 10.34 shows a single RAM cell. It can be instantiated in vector form to form a full-fledged ram. a_d is the decoded address line. When active, it turns on the bi-directional switch g3 and establishes a two-way connection between net ddd and net q. g1 and g2 together form a latch in feedback fashion. When g3 is OFF, the latch stores the state it was last in. It is connected to ddd through g3 by activating a_d for writing and reading. The design description for the RAM is shown in Figure 10.35. The simulation results are (partially) reproduced in Figure 10.36. The following are possible after such selection and connection:

- When wr = 1, **cmos** gate g4 turns ON; the data at the input port di (with strength **strong0** / **strong1**) are connected to q through ddd. It forces the latch to its state – since q has strength **pull0** / **pull1** only – di prevails here. This constitutes the write operation.
- When rd = 1, **cmos** gate g5 turns ON. The net ddd is connected to the output net do. The data stored in the latch are made available at the output port do. This constitutes the read operation.

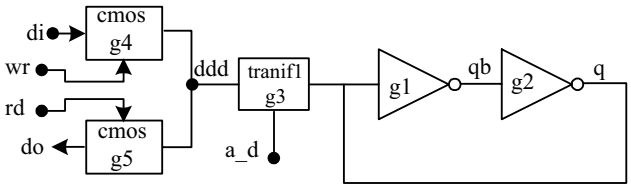

Figure 10.34 Circuit of a RAM cell,in block diagram form.

```
module ram_cell1(do,di,wr,rd,a_d);
output do; input di,wr,rd,a_d; wire ddd,q,qb,wrb,rdb;
not(rdb,rd),(wrb,wr);
not(pull1,pull0)(q,qb),(qb,q);
tranif1 g3(ddd,q,a_d);
cmos g4(ddd,di,wr,wrb),g5(do,ddd,rd,rdb);
endmodule

//test bench
module tst_ramcell1();
reg din,wr,rd,a_d; wire do;
ram_cell1 rmc1(do,din,wr,rd,a_d);
initial begin a_d=1'b0;din=1'b0;wr=1'b0;rd=1'b0; end
always #3a_d=1'b1;
always #10 din =~din;
always begin #3wr=1'b1; #8 wr=1'b0; end
always begin #2 rd=1'b1; #5 rd =1'b0; end
initial $monitor ($time," rd= %b ,wr = %b ,din = %b ,a_d = %b ,do = %b
",rd,wr,din,a_d,do);
initial #40 $stop;
endmodule
```

Figure 10.35 Design description of the RAM cell of Figure 10.34.

```
#          0 rd= 0 ,wr = 0 ,din = 0 ,a_d = 0 ,do = z
#          2 rd= 1 ,wr = 0 ,din = 0 ,a_d = 0 ,do = z
#          3 rd= 1 ,wr = 1 ,din = 0 ,a_d = 1 ,do = 0
#          7 rd= 0 ,wr = 1 ,din = 0 ,a_d = 1 ,do = z
#          9 rd= 1 ,wr = 1 ,din = 0 ,a_d = 1 ,do = 0
#         10 rd= 1 ,wr = 1 ,din = 1 ,a_d = 1 ,do = 1
#         11 rd= 1 ,wr = 0 ,din = 1 ,a_d = 1 ,do = 1
#         14 rd= 0 ,wr = 1 ,din = 1 ,a_d = 1 ,do = z
#         16 rd= 1 ,wr = 1 ,din = 1 ,a_d = 1 ,do = 1
#         20 rd= 1 ,wr = 1 ,din = 0 ,a_d = 1 ,do = 0
#         21 rd= 0 ,wr = 1 ,din = 0 ,a_d = 1 ,do = z
```

Figure 10.36 Partial results of simulating the test bench for the CMOS switch in Figure 10.35.

10.5 TIME DELAYS WITH SWITCH PRIMITIVES

Propagation delays can be specified for switch primitives on the same lines as was done with the gate primitives in Chapter 5. For example, an NMOS switch instantiated as

```
nmos g1 (out, in, ctrl );
```

has no delay associated with it. The instantiation

```
nmos (delay1) g2 (out, in, ctrl );
```

has delay1 as the delay for the output to rise, fall, and turn OFF. The instantiation

```
nmos  (delay_r, delay_f)  g3 (out, in, ctrl );
```

has delay_r as the rise-time for the output. delay_f is the fall-time for the output. The turn-off time is zero. The instantiation

```
nmos  (delay_r, delay_f, delay_o)  g4 (out, in, ctrl );
```

has delay_r as the rise-time for the output. delay_f is the fall-time for the output delay_o is the time to turn OFF when the control signal ctrl goes from 0 to 1. Delays can be assigned to the other uni-directional gates (**rcmos**, **pmos**, **rpmos**, **cmos**, and **rcmos**) in a similar manner. Bi-directional switches do not delay transmission – their rise- and fall-times are zero. They can have only turn-on and turn-off delays associated with them. **tran** has no delay associated with it.

```
tranif1 (delay_r, delay_f) g5 (out, in, ctrl );
```

represents an instantiation of the controlled bi-directional switch. When control changes from 0 to 1, the switch turns on with a delay of delay_r. When control changes from 1 to 0, the switch turns off with a delay of delay_f.

transif1 (delay0) g2 (out, in, ctrl);

represents an instantiation with delay0 as the delay for the switch to turn on when control changes from 0 to 1, with the same delay for it to turn off when control changes from 1 to 0. When a delay value is not specified in an instantiation, the turn-on and turn-off are considered to be ideal that is, instantaneous. Delay values similar to the above illustrations can be associated with **rtranif1**, **tranif0**, and **rtranif0** as well.

10.6 INSTANTIATIONS WITH STRENGTHS AND DELAYS

In the most general form of instantiation, strength values and delay values can be combined. For example, the instantiation

nmos (**strong1**, **strong0**) (delay_r, delay_f, delay_o) gg (s1, s2, ctrl) ;

means the following:

- It has strength **strong0** when in the low state and strength **strong1when** in the high state.
- When output changes state from low to high, it has a delay time of delay_r.
- When the output changes state from high to low, it has a delay time of delay_f.
- When output turns-off it has a turn-off delay time of delay _o.

rnmos, **pmos**, and **rpmos** switches too can be instantiated in the general form in the same manner. The general instantiation for the bi-directional gates too can be done similarly.

10.7 STRENGTH CONTENTION WITH TRIREG NETS

As was explained in Chapter 5, nets declared as **trireg** can have capacitive storage. Such storage can be assigned one of three strengths – **large**, **medium**, or **small**. Driving such a net from different sources can lead to contention; the relative strength levels of the sources also have a say in the signal level taken by the net. The contention resolution is brought out here through an illustrative example. A similar procedure of analysis can be followed in other cases as well.

Example 10.13

Figure 10.37 shows a circuit where a set of switches connect nets in series to a signal source. Strengths have been assigned to the nets and a test bench to bring out contention shown in Figure 10.38. The thicker line representation of net a3 in Figure 10.37 signifies that the capacitive storage strength of net a3 is stronger than that of net a2. The progress of simulation is depicted in Figure 10.39 showing the switch status and corresponding signal values at different times. Simulation results are shown in Figure 10.40. One can see that whenever a2 and a3 are connected (but isolated from a1), the stronger a3 prevails.

Observations:

- When a net is connected to a single signal source through intervening switches and capacitive nets, the source decides the value of the signal on the net.
- When 2 capacitive nets are connected, in case of a contention the stronger one prevails.
- When a signal source and a capacitive net drive another net, in case of a contention the signal value is dictated by the stronger of the two (see Table 5.5).

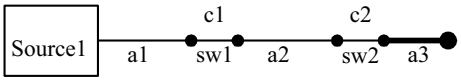

Figure 10.37 A simple circuit to demonstrate contention resolution with **`trireg`** nets.

```
module demo_1;
trireg(large)a3; trireg(small)a2; wire a1; reg c1,c2,b;
buf(strong1,strong0) source1(a1,b);
tranif1 sw1(a2,a1,c1), sw2(a3,a2,c2);
initial begin
          $display("t\ta1\tc1\ta2\tc2\ta3");
    #0   {c1,c2,b}=3'b111; #1 {c1,c2,b}=3'b011; #1 {c1,c2,b}=3'b001;
    #1   {c1,c2,b}=3'b000; #1 {c1,c2,b}=3'b100; #1 {c1,c2,b}=3'b000;
    #1   {c1,c2,b}=3'b010; #1 {c1,c2,b}=3'b000; #1 {c1,c2,b}=3'b100;
    #1   {c1,c2,b}=3'b000; #1 {c1,c2,b}=3'b010; #1 {c1,c2,b}=3'b000;
    #1   {c1,c2,b}=3'b001; #1 {c1,c2,b}=3'b101; #1 {c1,c2,b}=3'b111;
    #1   $stop;
      end
initial $monitor("%0d\t%b\t%b\t%b\t%b\t%b",$time,a1,c1,a2,c2,a3);
endmodule
```

Figure 10.38 A test bench for the circuit in Figure 10.37.

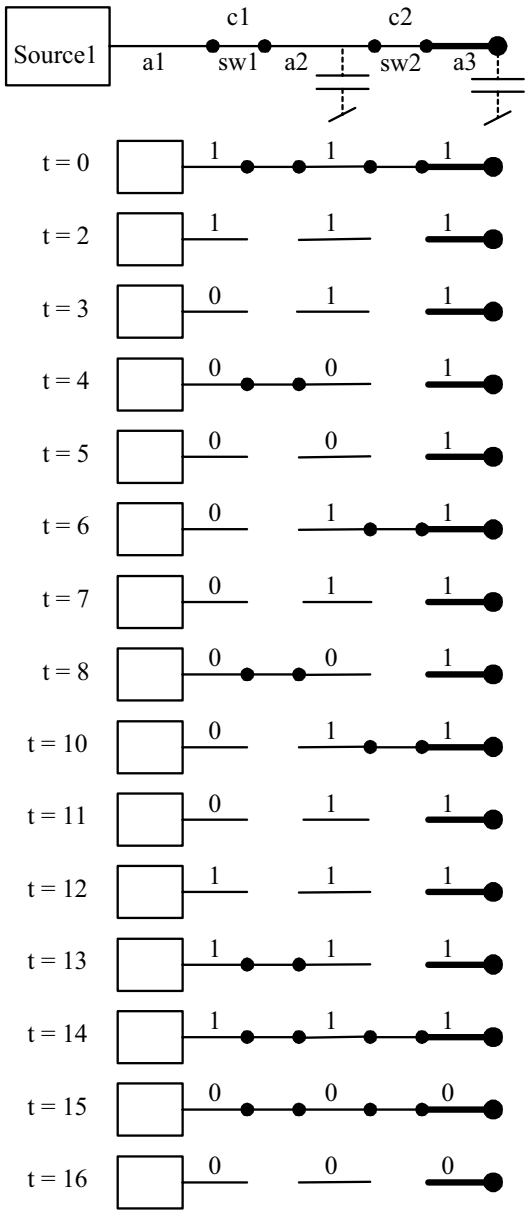

Figure 10.39 Changes in signal values at different times in Example 10.13 as the status of switches changes.

#t	a1	c1	a2	c	a3
#0	1	1	1	1	1
#1	1	0	1	1	1
#2	1	0	1	0	1
#3	0	0	1	0	1
#4	0	1	0	0	1
#5	0	0	0	0	1
#6	0	0	1	1	1
#7	0	0	1	0	1
#8	0	1	0	0	1
#9	0	0	0	0	1
#10	0	0	1	1	1
#11	0	0	1	0	1
#12	1	0	1	0	1
#13	1	1	1	0	1
#14	1	1	1	1	1

Figure 10.40 Results of the simulation-run with the test bench in Figure 10.38.

10.8 EXERCISES

1. Implement NAND, AND, OR GATES using MOS switches; test it with a suitable test-bench.
2. Implement a pseudo-NMOS 4-input NOR logic gate. Write a test bench and test it.
3. Implement a dynamic logic NAND gate for 4 inputs; the pullup is to be a precharge transistor, and the pulldown is to be an evaluation transistor, with the output being precharged in precharge phase of the clock. The output should be available during the evaluation phase. Write a test bench and test the switch level dynamic gate.
4. Implement a 4-to-1 MUX using CMOS transmission gates.
5. Build a dynamic 2-to-4 NOR gate based decoder and a dynamic 2-to-4 NAND gate-based decoder using NMOS switches and PMOS switches.
6. Implement a one-bit full adder using CMOS logic and test it using a test bench.
7. Implement a 4-bit look-ahead adder using CMOS logic and test it with a test bench.
8. Implement a 4-bit barrel shifter using NMOS switches.

9. Form an edge-triggered flip-flop; using it, form an 8-bit port as shown in Figure 10.40. Form a latch and modify it to provide two flags – data input flag (DIF) and data output flag (DOF). Normally, Wr and Rd are low; old state is retained. If Wr goes high, Di bits are loaded into the port at the next clock pulse. DIF flag is set. DOF is at zero state. If RD goes high, Do bits are loaded into the port at the next clock pulse. DOF Flag is set. DIF is at zero state. Design the port module; test it with a test bench.

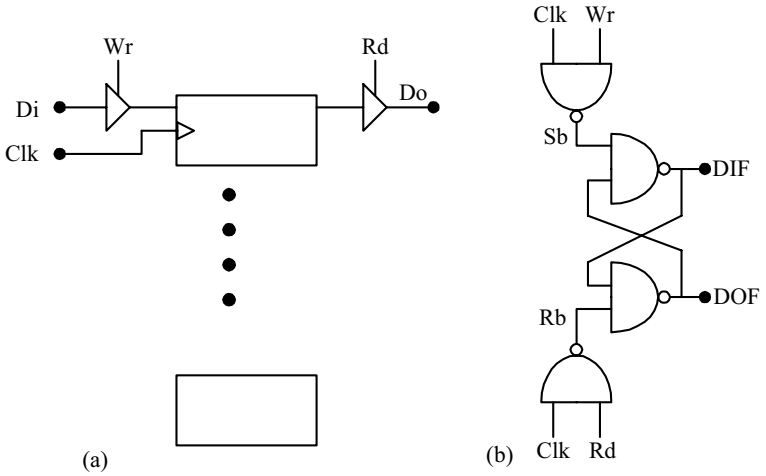

Figure 10.40 Figure for Exercise 9.