MODULE – 3

Gate-Level Modeling: Modeling using basic Verilog gate primitives, description of and/or and buf/not type gates, rise, fall and turn-off delays, min, max, and typical delays.

Dataflow Modeling: Continuous assignments, delay specification, expressions, operators, operands, operator types.

3.1 Gate Types

A logic circuit can be designed by use of logic gates. Verilog supports basic logic gates as predefined primitives. These primitives are instantiated like modules except that they are predefined in Verilog and do not need a module definition. All logic circuits can be designed by using basic gates. There are two classes of basic gates: **and/or gates and buf/not gates.**

3.1.1 And /Or Gates

And/or gates have one scalar output and multiple scalar inputs. The first terminal in the list of gate terminals is an output and the other terminals are inputs. The output of a gate is evaluated as soon as one of the inputs changes. The and/or gates available in Verilog are: **and, or, xor, nand, nor, xnor.**

The corresponding logic symbols for these gates are shown in Figure 3.1. Consider the gates with two inputs. The output terminal is denoted by out. Input terminals are denoted by i1 and i2.

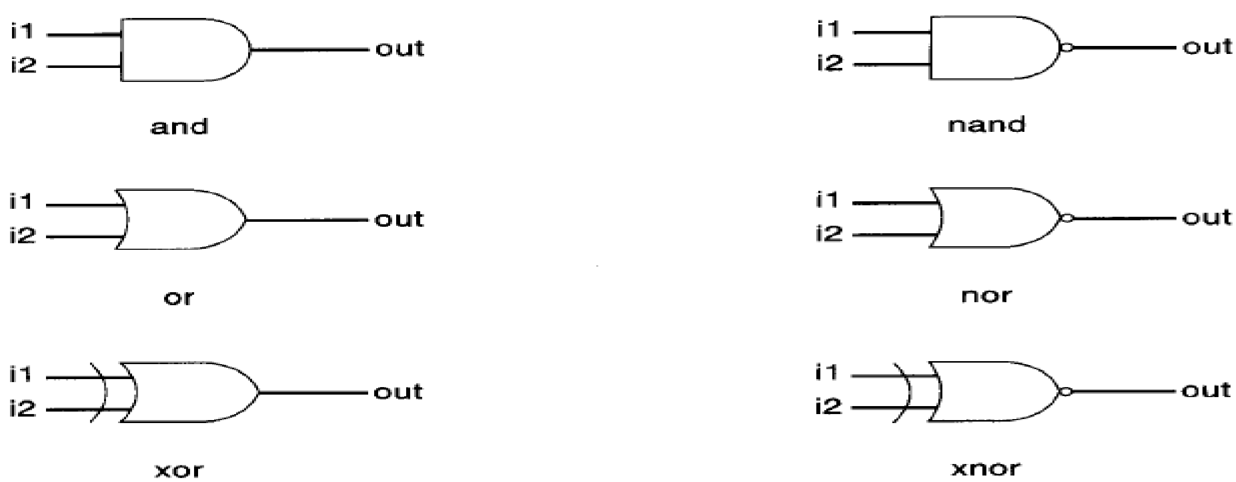


Figure 3.1: Basic gates

These gates are instantiated to build logic circuits in Verilog. Examples of gate instantiations are shown below. In Example 3.1, for all instances, OUT is connected to the output out, and IN1 and IN2 are connected to the two inputs i1 and i2 of the gate primitives. Note that the instance name does not need to be specified for primitives. This lets the designer instantiate hundreds of gates without giving them a name. More than two inputs can be specified in a gate instantiation. Gates with more than two inputs are instantiated by simply adding more input ports in the gate instantiation. Verilog automatically instantiates the appropriate gate.

Example 3.1: Gate Instantiation of And/Or Gates

```
wire OUT, IN1, IN2;
// basic gate instantiations.
        and a1(OUT, IN1, IN2);
        nand na1(OUT, IN1, IN2);
        or or1(OUT, IN1, IN2);
        nor nor1(OUT, IN1, IN2);
        xor x1(OUT, IN1, IN2);
        xnor nx1(OUT, IN1, IN2);
// More than two inputs;
        nand na1_3inp(OUT, IN1, IN2, IN3);
// gate instantiation without instance name
        and (OUT, IN1, IN2); // legal gate instantiation
```

Table 3.1: Truth tables for And/0r gates.

		i1			
	and	0	1	x	z
i2	0	0	0	0	0
	1	0	1	x	x
	x	0	x	x	x
	z	0	x	x	x

		i1			
	nand	0	1	x	z
i2	0	1	1	1	1
	1	1	0	x	x
	x	1	x	x	x
	z	1	x	x	x

		i1			
	or	0	1	x	z
i2	0	0	1	x	x
	1	1	1	1	1
	x	x	1	x	x
	z	x	1	x	x

		i1			
	nor	0	1	x	z
i2	0	1	0	x	x
	1	0	0	0	0
	x	x	0	x	x
	z	x	0	x	x

		i1			
	xor	0	1	x	z
i2	0	0	1	x	x
	1	1	0	x	x
	x	x	x	x	x
	z	x	x	x	x

		i1			
	xnor	0	1	x	z
i2	0	1	0	x	x
	1	0	1	x	x
	x	x	x	x	x
	z	x	x	x	x

The truth tables for these gates define how outputs for the gates are computed from the inputs. Truth tables are defined assuming two inputs. The truth tables for these gates are shown in Table 3.1. Outputs of gates with more than two inputs are computed by applying the truth table iteratively.

3.1.2 Buf/Not Gates

Buf/not gates have one scalar input and one or more scalar outputs. The last terminal in the port list is connected to the input. Other terminals are connected to the outputs. We will discuss gates that have one input and one output. Two basic buf/not gate primitives are provided in Verilog: **buf** **not**

The symbols for these logic gates are shown in Figure 3.2.

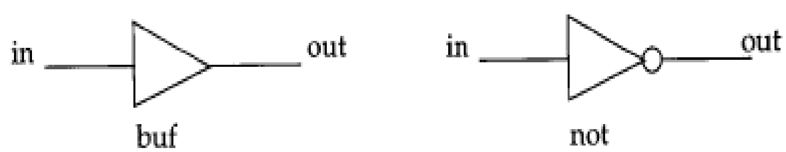


Figure 3.2: Buf and not gates.

These gates are instantiated in Verilog as shown Example 3.2. Notice that these gates can have multiple outputs but exactly one input, which is the last terminal in the port list.

Example 3.2: Gate Instantiations of Buf/Not Gates

```
// basic gate instantiations. buf b1(OUT1, IN);
not n1(OUT1, IN);
// More than two outputs
buf b1_2out(OUT1, OUT2, IN);
// gate instantiation without instance name not (OUT1, IN); // legal gate instantiation
```

Truth tables for gates with one input and one output are shown in Table 3.2.

Table 3.2: Truth tables for buf/not gates.

buf	in	out
	0	0
	1	1
	x	x
	z	x

not	in	out
	0	1
	1	0
	x	x
	z	x

Bufif/notif

Gates with an additional control signal on buf and not gates are also available.

bufif1 **notif1** **bufif0** **notif0**

These gates propagate only if their control signal is asserted. They propagate z if their control signal is deasserted. Symbols for bufif/notif are shown in Figure 3.3.

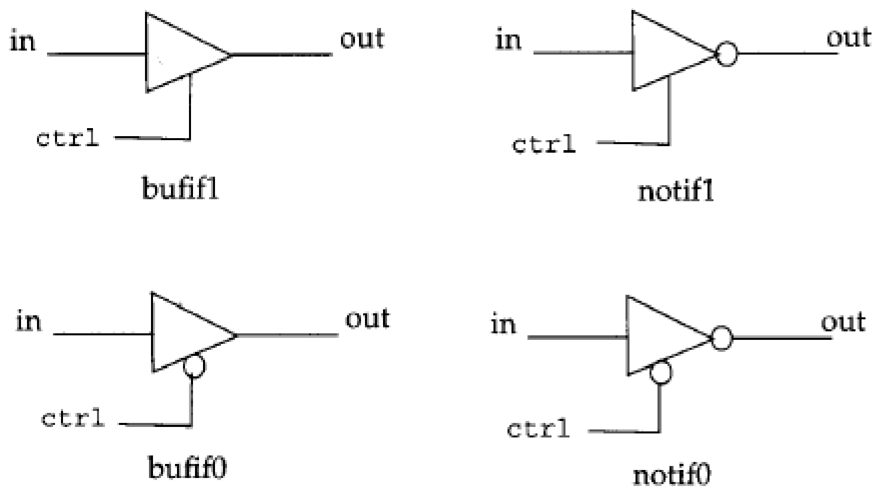


Figure 3.3: Gates bufif and notif

The truth tables for these gates are shown in Table 3.3.

Table 3.3: Truth tables for bufif / notif gates

bufif1	ctrl 0	ctrl 1	ctrl x	ctrl z
in 0	z	0	L	L
in 1	z	1	H	H
in x	z	x	x	x
in z	z	x	x	x

bufif0	ctrl 0	ctrl 1	ctrl x	ctrl z
in 0	0	z	L	L
in 1	1	z	H	H
in x	x	z	x	x
in z	x	z	x	x

notif1	ctrl 0	ctrl 1	ctrl x	ctrl z
in 0	z	1	H	H
in 1	z	0	L	L
in x	z	x	x	x
in z	z	x	x	x

notif0	ctrl 0	ctrl 1	ctrl x	ctrl z
in 0	1	z	H	H
in 1	0	z	L	L
in x	x	z	x	x
in z	x	z	x	x

These gates are used when a signal is to be driven only when the control signal is asserted. Such a situation is applicable when multiple drivers drive the signal. These drivers are designed to drive the signal on mutually exclusive control signals. Example 3-3 shows examples of instantiation of bufif and notif gates.

Example 3.3: Gate instantiation of bufif / notif gates

```
//Instantiation of bufif gates.
        bufif1 b1 (out, in, ctrl);
        bufif0 b0 (out, in, ctrl);
//Instantiation of notif gates
        notif1 n1 (out, in, ctrl);
        notif0 n0 (out, in, ctrl);
```

3.1.3 Array of Instances

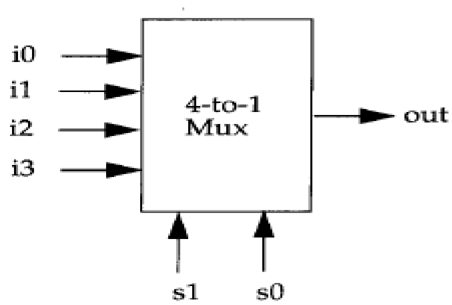
There are many situations when repetitive instances are required. These instances differ from each other only by the index of the vector to which they are connected. To simplify specification of such instances, Verilog HDL allows an array of primitive instances to be defined. Example3.4 shows an example of an array of instances.

Example 3.4: Simple Array of Primitive Instances

```
wire [7:0] OUT, IN1, IN2;
// basic gate instantiations. nand n_gate[7:0](OUT, IN1, IN2);
// This is equivalent to the following 8 instantiations
nand n_gate0(OUT[0], IN1[0], IN2[0]);
nand n_gate1(OUT[1], IN1[1], IN2[1]);
nand n_gate2(OUT[2], IN1[2], IN2[2]);
nand n_gate3(OUT[3], IN1[3], IN2[3]);
nand n_gate4(OUT[4], IN1[4], IN2[4]);
nand n_gate4(OUT[4], IN1[4], IN2[4]);
nand n_gate4(OUT[5], IN1[5], IN2[5]);
nand n_gate4(OUT[6], IN1[6], IN2[6]);
nand n_gate4(OUT[7], IN1[7], IN2[7]);
```

3.1.4 Examples

Gate-level multiplexer: Design of 4-to-1 multiplexer with 2 select signals. The I/O diagram and the truth table for the multiplexer are shown in Figure 3.4.



s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3

Figure 3.4: 4 to 1 multiplexer

Implement the logic for the multiplexer using basic logic gates. The logic diagram for the multiplexer is shown in Figure 3.5.

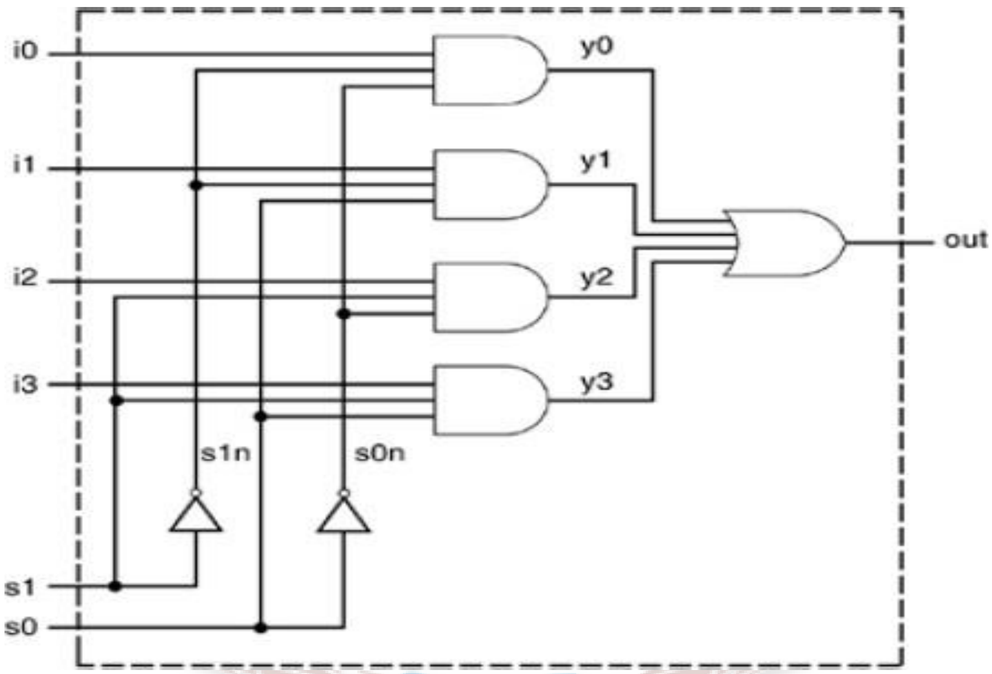


Figure 3.5: Logic diagram for multiplexer

The Verilog description for the multiplexer is shown in Example 3-5. Two intermediate nets, s0n and s1n, are created; they are complements of input signals s1 and s0. Internal nets y0, y1, y2, y3 are also required.

Example 3.5: Verilog description for multiplexer

```
// Module 4-to-1 multiplexer. Port list is taken exactly from the I/O diagram.
        module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
 // Port declarations from the I/O diagram output out;
        input i0, i1, i2, i3; input s1, s0;
// Internal wire declarations
        wire s1n, s0n;
```

```
        wire y0, y1, y2, y3;
 // Gate instantiations. Create s1n and s0n signals.
        not (s1n, s1);
        not (s0n, s0);
// 3-input and gates instantiated
        and (y0, i0, s1n, s0n);
        and (y1, i1, s1n, s0);
        and (y2, i2, s1, s0n);
        and (y3, i3, s1, s0);
 // 4-input or gate instantiated
        or (out, y0, y1, y2, y3);
        endmodule
```

This multiplexer can be tested with the stimulus shown in Example 3.6.

Example 3.6: Stimulus for multiplexer

```
// Define the stimulus module (no ports)
        module stimulus;
//Declare variables to be connected to i/p. Declare output wire.
        reg IN0, IN1, IN2, IN3;
        reg S1, S0;
        wire OUTPUT;
// Instantiate the multiplexer
        mux4_to_1 mymux(OUTPUT, IN0, IN1, IN2, IN3, S1, S0);
// Stimulate the inputs. Define the stimulus module (no ports)
        initial
        begin
// set input lines
        IN0 = 1; IN1 = 0; IN2 = 1; IN3 = 0;
        #1 $display("IN0= %b, IN1= %b, IN2= %b,  IN3= %b\n", IN0,IN1,IN2,IN3);
// choose IN0
        S1 = 0; S0 = 0;
        #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
// choose IN1
```

```
        S1 = 0; S0 = 1;
        #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
// choose IN2
        S1 = 1; S0 = 0;
        #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
 // choose IN3 S1 = 1; S0 = 1;
        #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
        end
        endmodule
```

4-bit Ripple Carry Full Adder

Design of a 4-bit full adder: The basic building block is a 1-bit full adder. The logic diagram for a 1-bit full adder is shown in Figure 3.6.

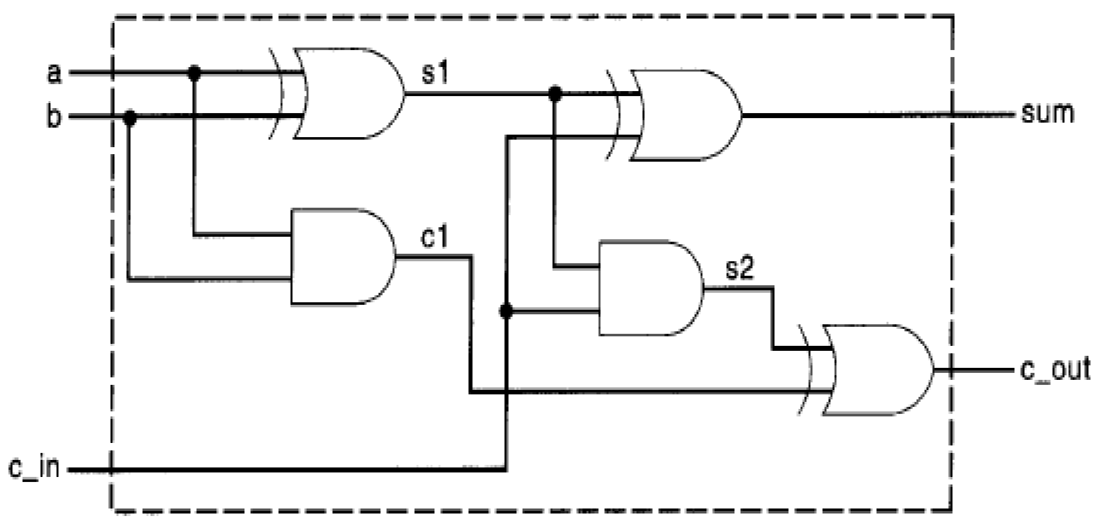


Figure 5.6: 1-bit full adder

This logic diagram for the 1-bit full adder is converted to a Verilog description, shown in Example 3.7.

Example 3.7: Verilog Description for 1-bit Full Adder

```
// Define a 1-bit full adder
        module fulladd(sum, c_out, a, b, c_in);
 // I/O port declarations
        output sum,  c_out;
        input a, b, c_in;
// Internal nets wire s1, c1, c2;
```

```
// Instantiate logic gate primitives
        xor (s1, a, b);
        and (c1, a, b);
        xor (sum, s1, c_in);
        and (c2, s1, c_in);
        xor (c_out, c2, c1);
        endmodule
```

A 4-bit ripple carry full adder can be constructed from four 1-bit full adders, as shown in Figure 3.7. Notice that fa0, fa1, fa2, and fa3 are instances of the module fulladd (1-bit full adder).

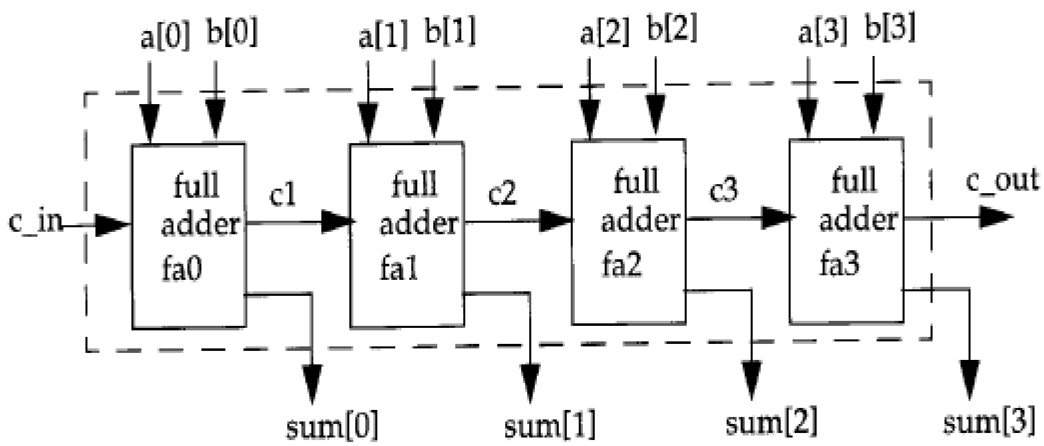


Figure 3.7: 4-bit ripple carry full adder

This structure can be translated to Verilog as shown in Example 3-8.

Example 3.8: Verilog description for 4-bit ripple carry full adder.

```
// Define a 4-bit full adder
        module fulladd4(sum, c_out, a, b, c_in);
 // I/O port declarations
        output [3:0]  sum;
        output  c_out;
        input[3:0] a, b;
        input c_in;
 // Internal nets wire c1, c2, c3;
// Instantiate four 1-bit full adders.
        fulladd fa0(sum[0], c1, a[0], b[0], c_in);
```

```
fulladd fa1(sum[1], c2, a[1], b[1], c1);
fulladd fa2(sum[2], c3, a[2], b[2], c2);
fulladd fa3(sum[3], c_out, a[3], b[3], c3);
endmodule
```

Finally, the design must be checked by applying stimulus, as shown in Example 3.9.

Example 3.9: Stimulus for 4-bit ripple carry full adder

```
// Define the stimulus (top level module)
module stimulus;
// Set up variables
reg [3:0] A, B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;
// Instantiate the 4-bit full adder. call it FA1_4
fulladd4 FA1_4(SUM, C_OUT, A, B, C_IN);
// Set up the monitoring for the signal values
initial
begin
$monitor($time," A= %b, B=%b, C_IN= %b, --- C_OUT= %b, SUM= %b\n", A, B, C_IN,
_OUT, SUM);
end
// Stimulate inputs initial
begin
A = 4'd0; B = 4'd0; C_IN = 1'b0;
#5 A = 4'd3; B = 4'd4;
#5 A = 4'd2; B = 4'd5;
#5 A = 4'd9; B = 4'd9;
#5 A = 4'd10; B = 4'd15;
#5 A = 4'd10; B = 4'd5;
end
endmodule
```

The output of the simulation is shown below:

0 A= 0000, B=0000, C_IN= 0, --- C_OUT= 0, SUM= 0000

5 A= 0011, B=0100, C_IN= 0, --- C_OUT= 0, SUM= 0111

10 A= 0010, B=0101, C_IN= 0, --- C_OUT= 0, SUM= 0111

15 A= 1001, B=1001, C_IN= 0, --- C_OUT= 1, SUM= 0010

20 A= 1010, B=1111, C_IN= 0, --- C_OUT= 1, SUM= 1001

25 A= 1010, B=0101, C_IN= 0, --- C_OUT= 1, SUM= 0000

3.2 Gate Delays

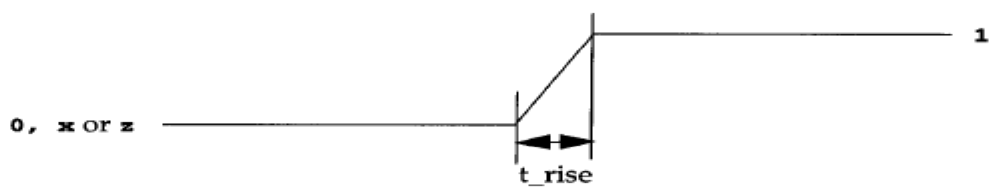
Until now, circuits are described without any delays (i.e., zero delay). In real circuits, logic gates have delays associated with them. Gate delays allow the Verilog user to specify delays through the logic circuits. Pin-to-pin delays can also be specified in Verilog.

3.2.1 Rise, Fall, and Turn-off Delays

There are three types of delays from the inputs to the output of a primitive gate.

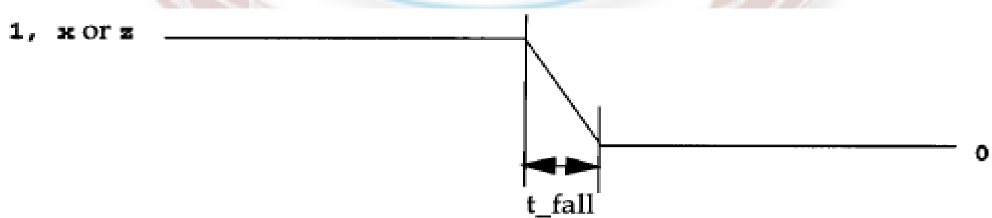
Rise delay

The rise delay is associated with a gate output transition to a 1 from another value.



Fall delay

The fall delay is associated with a gate output transition to a 0 from another value.



Turn-off delay

The turn-off delay is associated with a gate output transition to the high impedance value (z) from another value. If the value changes to x, the minimum of the three delays is considered.

Three types of delay specifications are allowed. If only one delay is specified, this value is used for all transitions. If two delays are specified, they refer to the rise and fall delay values. The turn-off delay is the minimum of the two delays. If all three delays are specified, they refer to rise, fall, and turn-off delay values. If no delays are specified, the default value is zero. Examples of delay specification are shown in Example 3.10.

Example 3.10: Types of Delay Specification:

```
// Delay of delay_time for all transitions
        and #(delay_time) a1(out, i1, i2);
// Rise and Fall Delay Specification.
        and #(rise_val, fall_val) a2(out, i1, i2);
// Rise, Fall, and Turn-off Delay Specification
        bufif0 #(rise_val, fall_val, turnoff_val) b1 (out, in, control);
```

Examples of delay specification:

```
and #(5) a1(out, i1, i2); //Delay of 5 for all transitions
and #(4,6) a2(out, i1, i2); // Rise = 4, Fall = 6
bufif0 #(3,4,5) b1 (out, in, control); // Rise = 3, Fall = 4, Turn-off= 5
```

Min/Typ/Max Values

Verilog provides an additional level of control for each type of delay mentioned above. For each type of delay-rise, fall, and turn-off-three values, min, typ, and max, can be specified. Any one value can be chosen at the start of the simulation. Min/typ/max values are used to model devices whose delays vary within a minimum and maximum range because of the IC fabrication process variations.

Min value

The min value is the minimum delay value that the designer expects the gate to have.

Typ val

The typ value is the typical delay value that the designer expects the gate to have.

Max value

The max value is the maximum delay value that the designer expects the gate to have.

Min, typ, or max values can be chosen at Verilog run time. Method of choosing a min/typ/max value may vary for different simulators or operating systems. (For Verilog- XL, the values are chosen by specifying options +maxdelays, +typdelays, and +mindelays at run time. If no option is specified, the typical delay value is the default). This allows the designers the flexibility of building three delay values for each transition into their design. The designer can experiment with delay values without modifying the design.

Examples of min, typ, and max value specification for Verilog-XL are shown in Example3-11.

Example 3.11: Min, typ, and max delay values

```
// One delay
 // if +mindelays, delay= 4
 // if +typdelays, delay= 5
 // if +maxdelays, delay= 6
        and #(4:5:6) a1(out, i1, i2);
// Two delays
 // if +mindelays, rise= 3, fall= 5, turn-off = min(3,5)
 // if +typdelays, rise= 4, fall= 6, turn-off = min(4,6)
 // if +maxdelays, rise= 5, fall= 7, turn-off = min(5,7)
        and #(3:4:5, 5:6:7) a2(out, i1, i2);
 // Three delays
// if +mindelays, rise= 3, fall= 5, turn-off = 2
 // if +typdelays, rise= 4, fall= 6, turn-off = 3
 // if +maxdelays, rise= 5, fall= 7, turn-off = 4
        and #(3:4:5, 5:6:7, 2:3:4 ) a3(out, i1, i2);
```

3.2.3 Delay Example

Let us consider a simple example to illustrate the use of gate delays to model timing in the logic circuits. A simple module called D implements the following logic equations:

out = (a b) + c

The gate-level implementation is shown in Module D (Figure 3.8). The module contains two gates with delays of 5 and 4 time units.

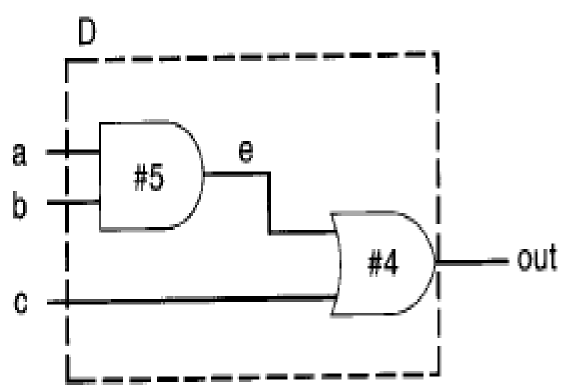


Figure 3.8: Module D

The module D is defined in Verilog as shown in Example 3.12.

Example 3.12: Verilog definition for module D with delay

```
// Define a simple combination module called D
        module D (out, a, b, c);
// I/O port declarations
        output out;
        input a,b,c;
 // Internal nets
        wire e;
// Instantiate primitive gates to build the circuit
        and #(5) a1(e, a, b); //Delay of 5 on gate a1
        or #(4) o1(out, e,c); //Delay of 4 on gate o1
        endmodule
```

This module is tested by the stimulus file shown in Example 3.13.

Example 3.13: Stimulus for Module D with Delay

```
// Stimulus (top-level module)
        module stimulus;
// Declare variables reg A, B, C;
        wire OUT;
 // Instantiate the module D
        D d1( OUT, A, B, C);
// Stimulate the inputs. Finish the simulation at 40 time units.
        initial
        begin
        A= 1'b0; B= 1'b0; C= 1'b0;
        #10    A= 1'b1; B= 1'b1; C= 1'b1;
        #10 A= 1'b1; B= 1'b0; C= 1'b0;
        #20 $finish;
        end
        endmodule
```

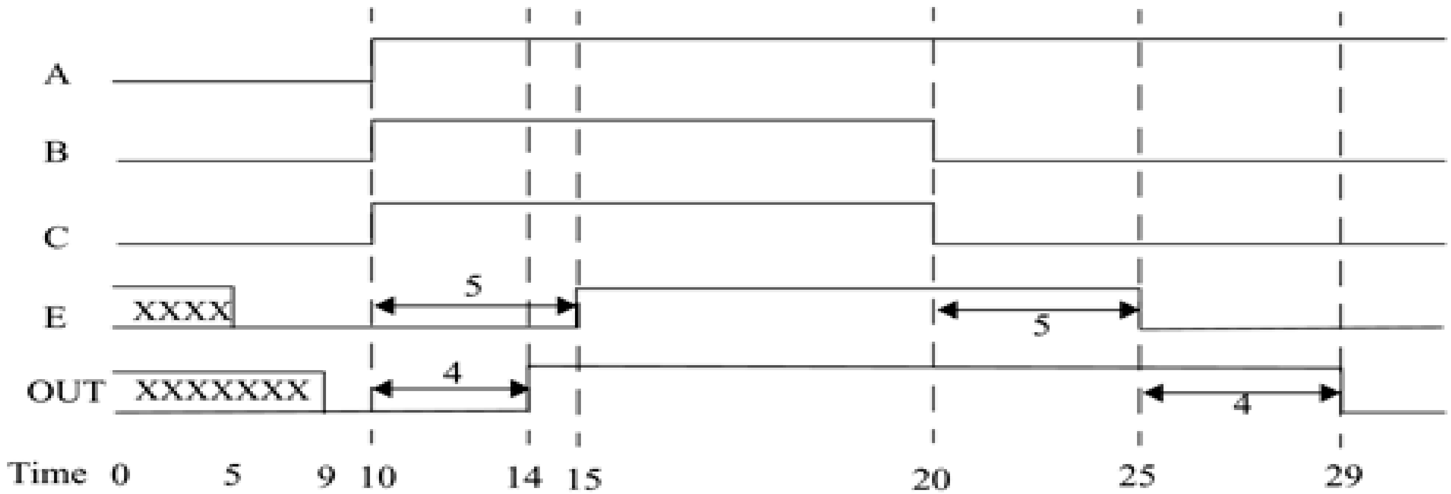


Figure 3.9: Waveforms for delay simulation.

The waveforms from the simulation are shown in Figure 3.9 to illustrate the effect of specifying delays on gates. The waveforms are not drawn to scale. However, simulation time at each transition is specified below the transition.

1. The outputs E and OUT are initially unknown.
2. At time 10, after A, B, and C all transition to 1, OUT transitions to 1 after a delay of 4 time units and E changes value to 1 after 5 time units.
3. At time 20, B and C transition to 0. E changes value to 0 after 5 time units, and OUT transitions to 0, 4 time units after E changes.

3.4 Dataflow Modeling

For small circuits, the gate-level modeling approach works very well because the number of gates is limited and the designer can instantiate and connects every gate individually. Also, gate-level modeling is very intuitive to a designer with a basic knowledge of digital logic design. However, in complex designs the number of gates is very large. Thus, designers can design more effectively if they concentrate on implementing the function at a level of abstraction higher than gate level. Dataflow modeling provides a powerful way to implement a design. Verilog allows a circuit to be designed in terms of the data flow between registers and how a design processes data rather than instantiation of individual gates.

3.4.1 Continuous Assignments

A continuous assignment is the most basic statement in dataflow modeling, used to drive a value onto a net. This assignment replaces gates in the description of the circuit and describes the circuit at a higher level of abstraction. The assignment statement starts with the keyword assign. The syntax of an **assign** statement is as follows.

continuous_assign ::= assign [drive_strength] [delay3] list_of_net_assignments ;
list_of_net_assignments ::= net_assignment { , net_assignment }
net_assignment ::= net_lvalue = expression

The default value for drive strength is strong1 and strong0. The delay value is also optional and can be used to specify delay on the assign statement. This is like specifying delays for gates. Continuous assignments have the following characteristics:

1. The left hand side of an assignment must always be a scalar or vector net or a concatenation of scalar and vector nets. It cannot be a scalar or vector register.
2. Continuous assignments are always active. The assignment expression is evaluated as soon as one of the right- hand-side operands changes and the value is assigned to the left-hand-side net.
3. The operands on the right-hand side can be registers or nets or function calls. Registers or nets can be scalars or vectors.
4. Delay values can be specified for assignments in terms of time units. Delay values are used to control the time when a net is assigned the evaluated value. This feature is similar to specifying delays for gates. It is very useful in modeling timing behavior in real circuits.

Example 3.14: Examples of Continuous Assignment

```
// Continuous assign. out is a net. i1 and i2 are nets.
        assign out = i1 & i2;
// Continuous assign for vector nets. addr is a 16-bit vector ne addr1 and addr2 are 16-bit
//vector registers.
        assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];
// Concatenation. Left-hand side is a concatenation of a scalar net and a vector net.
        assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
```

3.14.2 Implicit Continuous Assignment

Instead of declaring a net and then writing a continuous assignment on the net, Verilog provides a shortcut by which a continuous assignment can be placed on a net when it is declared. There can be only one implicit declaration assignment per net because a net is declared only once. In the example below, an implicit continuous assignment is contrasted with a regular continuous assignment.

```
//Regular continuous assignment
```

```
    wire out;
    assign out = in1 & in2;
//Same effect is achieved by an implicit continuous assignment
    wire out = in1 & in2;
```

3.14.3 Implicit Net Declaration

If a signal name is used to the left of the continuous assignment, an implicit net declaration will be inferred for that signal name. If the net is connected to a module port, the width of the inferred net is equal to the width of the module port.

```
// Continuous assign. out is a net.
wire i1, i2;
assign out = i1 & i2; //Note that out was not declared as a wire but an implicit wire
                      // declaration for out is done by the simulator
```

3.5 Delays

Delay values control the time between the change in a right-hand-side operand and when the new value is assigned to the left-hand side. Three ways of specifying delays in continuous assignment statements are *regular assignment delay, implicit continuous assignment delay,* and *net declaration delay.*

3.5.1 Regular Assignment Delay

The first method is to assign a delay value in a continuous assignment statement. The delay value is specified after the keyword assign. Any change in values of in1 or in2 will result in a delay of 10 time units before re-computation of the expression in1 & in2, and the result will be assigned to out. If in1 or in2 changes value again before 10 time units when the result propagates to out, the values of in1 and in2 at the time of re-computation are considered. This property is called inertial delay. An input pulse that is shorter than the delay of the assignment statement does not propagate to the output.

```
assign #10 out = in1 & in2; // Delay in a continuous assign
```

3.5.2 Implicit Continuous Assignment Delay

An equivalent method is to use an implicit continuous assignment to specify both a delay and an assignment on the net.

```
//implicit continuous assignment delay
    wire #10 out = in1 & in2; //
```

same as

```
wire out;
assign #10 out = in1 & in2;
```

The declaration above has the same effect as defining a wire out and declaring a continuous assignment on out.

3.5.3 Net Declaration Delay

A delay can be specified on a net when it is declared without putting a continuous assignment on the net. If a delay is specified on a net out, then any value change applied to the net out is delayed accordingly. Net declaration delays can also be used in gate-level modeling.

```
//Net Delays
	wire # 10 out;
	assign out = in1 & in2;
//The above statement has the same effect as the following:
	wire out;
	assign #10 out = in1 & in2;
```

3.6 Expressions, Operators, and Operands

Dataflow modeling describes the design in terms of expressions instead of primitive gates. Expressions, operators, and operands form the basis of dataflow modeling.

3.6.1 Expressions

Expressions are constructs that combine operators and operands to produce a result.

```
// Examples of expressions. Combines operands and operators
	a ^ b
	addr1[20:17] + addr2[20:17]
	in1 | in2
```

3.6.2 Operands

Operands can be any one of the data types defined, Data Types. Some constructs will take only certain types of operands. Operands can be constants, integers, real numbers, nets, registers, times, bit-select (one bit of vector net or a vector register), part-select (selected bits of the vector net or register vector), and memories or function calls.

```
integer count, final_count;
	final_count = count + 1;//count is an integer operand
	real a, b, c;
```

```
c = a - b; //a and b are real operands
reg [15:0] reg1, reg2;
reg [3:0] reg_out;
reg_out = reg1[3:0] ^ reg2[3:0]; //reg1[3:0] and reg2[3:0] are part-select reg operands
reg ret_value;
ret_value = calculate_parity(A, B); /   /calculate_parity is a function type operand
```

3.6.3 Operators

Operators act on the operands to produce desired results. Verilog provides various types of operators.

```
d1 && d2 // && is an operator on operands d1 and d2.
!a[0] // ! is an operator on operand a[0]
B >> 1 // >> is an operator on operands B and 1
```

3.7 Operator Types

Verilog provides many different operator types. Operators can be arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, or conditional. Some of these operators are similar to the operators used in the C programming language. Each operator type is denoted by a symbol. Table shows the complete listing of operator symbols classified by category.

Table 3.4: Operator types and symbols

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Shift	>> <<	Right shift Left shift	two two
Concatenation	{ }	Concatenation	any number
Replication	{ { } }	Replication	any number
Conditional	?:	Conditional	three

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
Logical	!	logical negation	one
	&&	logical and	two
	||	logical or	two
Relational	>	greater than	two
	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two
Equality	==	equality	two
	!=	inequality	two
	===	case equality	two
	!==	case inequality	two
Bitwise	~	bitwise negation	one
	&	bitwise and	two
	|	bitwise or	two
	^	bitwise xor	two
	^~ or ~^	bitwise xnor	two
Reduction	&	reduction and	one
	~&	reduction nand	one
	|	reduction or	one
	~|	reduction nor	one
	^	reduction xor	one
	^~ or ~^	reduction xnor	one

3.7.1 Arithmetic Operators

There are two types of arithmetic operators: binary and unary

Binary operators

Binary arithmetic operators are multiply (*), divide (/), add (+), subtract (-), power (**) and modulus (%). Binary operators take two operands.

Examples:

A = 4'b00111; B = 4'b01001 // A and B are register vectors

D = 6, E = 4, F = 2// D and E are integers

A*B // Multiply A and B. Evaluates to 4'b1100

D/E // Divide D by R. Evaluates to 1. Truncates any fractional part

A+B// Add A and B. Evaluates to 4'b0111

B-A // Subtract A from B. Evaluates to 4'b0001

F = E ** F //E to the power F, yields 16

If any operand value x, then the result of the entire expression is x. This seems intuitive because if an operand value is not known precisely, the result should be an unknown.

Example:

in1 = 4'b101x;

in2 = 4'b1010;

sum = in1 + in2; // sum will be evaluated to the value 4'bx

Modulus operators produce the remainder from the division of two numbers. They operate similarly to the modulus operator in the C programming language.

Examples:

13 % 3// Evaluates to 1

16 % 4 Evaluates to 0

-7 % 2 Evaluates to -1, takes sign of the first operand

7 % -2 // Evaluates to 1, takes sign of the first operand

Unary operators

The operators + and - can also work as unary operators. They are used to specify the positive or negative sign of the operand. Unary + or - operator have higher precedence than the binary + or - operators.

Examples:

-4/ Negative 4

+5 // Positive 5

Negative numbers are represented as 2's complement internally in Verilog. It is advisable to use negative numbers only of the type integer or real in expressions. Designers should avoid negative numbers of the type <sss>'<base> <nnn> in expressions because they are converted to unsigned 2's complement numbers and hence yield unexpected results.

Examples:

//Advisable to use integer or real numbers

-10 / 5// Evaluates to 2

//Do not use numbers of type <sss> '<base> <nnn>

-'d10 / 5 // Is equivalent (2's complement of 10)/5 - (2^{32} - 10)/5 // where 32 is the default //machine word width. This evaluates to an incorrect and unexpected result.

3.7.2 Logical Operators

Logical operators are logical and (&&), logical-or (||) and logical-not (!). Operators && and || are binary operators. Operator ! is a unary operator. Logical operators follow these conditions:

1. Logical operators always evaluate to 1-bit value, 0 (false), 1 (true), or x(ambiguous).
2. If an operand is not equal to zero, it is equivalent to a logical 1 (true condition). If it is equal to zero, it is equivalent to a logical 0 (false condition).If any operand bit is x or z, it is equivalent to x (ambiguous condition) and as is normally treated by simulators as a false condition.
3. Logical operators take variables or expressions as operands.

Use of parentheses to group logical operation is highly recommended to improve readability. Also, the user does not have to remember the precedence of operators.

Examples:

1) A = 3, B=0;

A && B / Evaluates to 0. Equivalent to (logical -1 && logical-0)

A || B // Evaluates to 1. Equivalent to (logical -1 || logical-0)

!A // Evaluates to 0. Equivalent to not (logical -1)

!B // Evaluates to 1. Equivalent to not (logical -0)

2) A = 2'b0x, B = 2'b10;

A && B / Evaluates to x. Equivalent to (x && logical-1)

3) (a==2) && (b==3) // Evaluates to 1 if both a==2 and b==3 are true.

// Evaluates to 0 if either is false.

3.7.3 Relational Operators

Relational operators are greater than (>), less than (<), greater than or equal (>=), and less than or equal to (<=). If relational operators are used in an expression, the expression returns a logical value of 1 if the expression is true and 0 if the expression is false. If there are any unknown or z bits in the operands, the espresso takes a value x. These operators function exactly as the corresponding operators in the C programming language.

Examples:

```
//A=4, B=3
// X = 4'b1010, Y = 4'b1101, Z = 4'b1xxx;
A <= B // Evaluates to a logical 0
A > B // Evaluate to a logical 1
Y >= X // Evaluates to a logical 1
Y < Z // Evaluates to an x
```

3.7.4 Equality Operators

Equality operators are logical equality (==), logical inequality (!=), case equality (===) and raw inequality (!==). When used in an expression, equality operators return logical value 1 if true, 0 if false. These operators compare the two operands bit by bit, with zero filling if the operands are of unequal length. Table 3.5 lists the operators.

Table 3.5

Expression	Description	Possible logical value
a == b	a equal to b, result unknown if x or z in a or b	0, 1, x
a != b	a not equal to b, result unknown if x or z in a or b	0, 1, x
a === b	a equal to b, including x and z	0, 1
a !== b	a not equal to b, including x and z	0, 1

It is important to note the difference between the logical equality operators (==, !=) and case equality operators (===, !==). The logical equality operators (==, !=) will yield an x if either operand has x or z in its bits. However, the case equality operators (===, !==) compare both operands bit by bit and compare all bits, including x and z. The result is 1 if the operands match exactly, including x and z bits. The result is 0 if the operands do not match exactly. Case equality operators never result in an x.

Examples:

```
// A = 4, B = 3
// X=4'b1010, Y = 4'bl101
```

// Z = 4 b1xxx, M = 4'blxxz, N= 4'blxxx

A == B // Results in logical 0

X != Y // Results in logical 1

X == Z // Results in x

Z === M // Results in logical 1 (all bits match, including x and z)

Z === N // Results in logical 0 (least significant bit does not match)

M !== N // Results in logical 1

3.7.5 Bitwise Operators

Bitwise operators are negation (~), and (&), or (|), xor (^), xnor (^~, ~^), Bitwise operators perform a bit-by-bit operation on two operands. They take each bit in one operand and perform the operation with the corresponding bit in the other operand. If one operand is shorter than the other, it will be bit-extended with zeros to match the length of the longer operand. Logic tables for the bit-by-bit computation are shown in Table 3.6. A z is treated as an x in a bitwise operation. The exception is the unary negation operator (-), which takes only one operand and operates on the bits of the single operand.

Table 3.6: Truth Tables for Bitwise Operator

bitwise and	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

bitwise or	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

bitwise xor	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

bitwise xnor	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

bitwise negation	Result
0	1
1	0
x	x

Examples:

// X = 4'b1010, Y = 4'b1101

// Z= 4'b10x1

~X // Negation. Result is 4'b0101

X&Y // Bitwise and. Result is 4'b1000

X | Y // Bitwise or Result is 4'b1111

X ^ Y // Bitwise xor. Result is 4'b0111

X ^~ Y // Bitwise xnor. Result is 4'b1000

X & Z // Result is 4'b10x0

It is important to distinguish bitwise operators ~, &, and | from logical operators !, &&, ||. Logical operators always yield s logical value 0, 1, x, whereas bitwise operators yield a bit-by-bit value. Logical operators perform a logical operation, not a bit by bit operation.

// X = 4b1010, Y = 4'b0000;

X | Y / bitwise operation. Result is 4'b1010

X || Y / logical operation. Equivalent to 1 ||0. Result is 1.

3.7.6 Reduction Operators

Reduction operators are and (&), nand (~&), or (|), nor(~|), xor(^), and xnor(~^, ^~). Reduction operators take only one operand. Reduction operators perform a bitwise operation on a single vector operand and yield a 1-bit result. The difference is that bitwise operations are on bits from two different operands, whereas reduction operations are on the bits of the same operand. Reduction Operators work bit by bit from right to left. Reduction nand, reduction nor, and reduction xnor are computed by inverting the result of the reduction and, reduction or, and reduction xor, respectively.

Example:

// x = 4'b1010

&X //Equivalent to 1& 0 & 1 & 0. Results in 1'b0

|X//Equivalent to 1| 0 | 1 | 0. Results in 1'b1

^X //Equivalent to 1 ^ 0 ^ 1 ^0. Result in 1'b0

//A reduction xor or xnor can be used for even or odd parity generation of a vector.

The use of a similar set of symbols for logical (!, &&, ||), bitwise (~, &, |, ^), and reduction operators (&, |, ^) is somewhat confusing initially. The difference lies in the number of operands each operator takes and also the value of results computed.

3.7.7 Shift Operators

Shift operators are right shift (>>), left shift (<<), arithmetic right shift (>>>) and arithmetic left shift (<<<). Regular shift operators shift a vector operand to the right or the left by a specified number of bits. The operands are the vector and the number of bits to shift.

When the bits are shifted, the vacant bit positions are filled with zeros. Shift operations does not wrap around. Arithmetic shift operators use the context of the expression to determine the value with which to fill the vacated bits.

Example:

```
// x = 4'b1100
Y = X >> 1: Y is 4'b0110. Shift right 1 bit. 0 filled in MSB position
Y = X << 1; //Y is 4 b1000. Shift left 1 bit. 0 filled in LSB position
Y = X << 2; //Y is 4 b0000. Shift Left 2 bits.
integer a, b, c//Signed data types
a = 0;
b = -10;  // 00111...10110 binary
c = a+ (b>>> 3): //Results in -2 decimal, due to arithmetic shift.
```

Shift operators are useful because they allow the designer to model shift operations, shift-and-add algorithms for multiplication, and other useful operations.

3.7.8 Concatenation Operator

The concatenation operator ({ , }) provides a mechanism to append multiple operands. The operands must be sized. Unsized operands are not allowed because the size of each operand must be known for computation of the size of the result.

Concatenations are expressed as operands within braces, with commas separating the operands. Operands can be scalar nets or registers, vector nets or registers, bit-select, part-select, or sized constants.

Example:

```
// A = 1'b1, B = 2'b00, C = 2'b10, D = 3'bl10;
Y = {B, C} // Result Y is 4'b0010
Y = {A, B, C, D, 3'b001} // Result Y is 11'b10010110001
Y = {A, B[0], C[1]) // Result Y is 3'b101
```

3.7.9 Replication Operator

Repetitive concatenation of the same number can be expressed by using a replication constant. A replication constant specifies how many times to replicate the number inside the brackets ({ })

Example:

reg A;

reg [1:0] B, C;

reg [2:0] D;

A= 1'b1, B = 2'b00, C= 2'b10, D = 3'b110;

Y= {4 {A} } // Result Y is 4'b1111

Y = {4{A}, 2(B} } Result is 8'b11110000

Y = {4{A}, 2{B}, C} // Result Y is 10'b1111000010

3.7.10 Conditional Operator:

The conditional operator (?:) takes three operands.

Usage: condition_expression ? true_expression : false_expression

The condition expression (condition_expression) is first evaluated. If the result is true (logical 1), then the true_expression is evaluated. If the result is false (logical 0), then the false_expression is evaluated. If the result is x (ambiguous), then both true_expression and false_expression are evaluated and their results are compared, bit by bit, to return for each bit position an x if the bits are different and the value of the bits if they are the same.

The action of a conditional operator is similar to a multiplexer. Alternately, it can be compared to the if-else expression.

Conditional operators are frequently used in dataflow modeling to model conditional assignments. The conditional expression acts as a switching control.

```
// model functionality of a tristate buffer
assign addr_bus = drive_enable ? addr_out : 36'bz;
//model functionality of a 2-to-1 mux
assign out = control ? in1 : in0;
```

Conditional operations can be nested. Each true_expression or false_expression can itself be a conditional operation. In the example that follows, convince yourself that (A==3) and control are the two select signals of 4-to-1 multiplexer with n, m, y, x as the inputs and out as the output signal.

```
assign out = (A == 3) ? (control ? x : y) : ( control ? m : n);
```

3.7.11 Operator Precedence

Having discussed the operators, it is now important to discuss operator precedence. If no parentheses are used to separate parts of expressions, Verilog enforces the following precedence. Operators listed in Table 3.7 are in order from highest precedence to lowest precedence. It is recommended that parentheses be used to separate expressions except in the case of unary operators or when there is no ambiguity.

Table 3.4: Operator precedence

Operators	Operator symbols	Precedence
Unary	**+ - ! ~**	Highest precedence
Multiply, Divide, Modulus	*** / %**	
Add, Subtract	**+ -**	
Shift	**<< >>**	
Relational	**< <= > >=**	
Equality	**== != === !==**	
Reduction	**&, ~&** **^, ^~** **| ~|**	
Logical	**&&** **||**	
Conditional	**?:**	Lowest precedence

3.8 Example

A design can be represented in terms of gates, data flow, or a behavioral description. Consider the 4-to-1 multiplexer and 4-bit full adder described earlier. Previously, these designs were directly translated from the logic diagram into a gate-level Verilog description. Here, we describe the same designs in terms of data flow. We also discuss two additional examples: a 4-bit full adder using carry look ahead and a 4-bit counter using negative edge-triggered D-flip-flops.

3.8.1 4-to-1 Multiplexer

Gate-level modeling of a 4-to-1 multiplexer, Example. The logic diagram for the multiplexer is given in Figure 3.4 and the gate-level Verilog description is shown in Example. We describe the multiplexer, using dataflow statements. Compare it with the gate-level description. We show two methods to model the multiplexer by using dataflow statements.

Method 1: logic equation

We can use assignment statements instead of gates to model the logic equations of the multiplexer. Notice that everything is same as the gate-level Verilog description except that computation of out is done by specifying one logic equation by using operators instead of individual gate instantiations. I/O ports remain the same. This important so that the interface with the environment does not change. Only the internals of the module change.

Example 3.15: 4-to-1 Multiplexer, Using Logic Equations

```
// Module 4-to-1 multiplexer using data flow. logic equation
// Compare to gate-level model
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram output out;
input i0, i1, i2, i3; input s1, s0;
//Logic equation for out
assign out = (~s1 & ~s0 & i0)| (~s1 & s0 & i1) | (s1 & ~s0 & i2) | (s1 & s0 & i3) ;
endmodule
```

Method 2: conditional operator

There is a more concise way to specify the 4-to-1 multiplexers. Conditional operators can be used to implement 4-to-1 multiplexers.

Example3.16: 4-to-1 Multiplexer, Using Conditional Operators

```
// Module 4-to-1 multiplexer using data flow. Conditional operator.
// Compare to gate-level model
module multiplexer4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram output out;
input i0, i1, i2, i3;
input s1, s0;
// Use nested conditional operator
assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ;
endmodule
```

In the simulation of the multiplexer, the gate-level module can be substituted with the dataflow multiplexer modules described above. The stimulus module will not change. The simulation results will be identical. By encapsulating functionality inside a module, we can replace the gate-level module with a dataflow module without affecting the other modules in the simulation. This is a very powerful feature of Verilog.

3.8.2: 4 bit Full Adder

The 4-bit full adder in, Examples, was designed by using gates; the logic diagram is shown in Figure 3.7. In this section, we write the dataflow description for the 4-bit adder. In gates, we had to first describe a 1-bit full adder. Then we built a 4-bit full ripple carry adder. We again illustrate two methods to describe a 4-bit full adder by means of dataflow statements.

Method 1: dataflow operators

A concise description of the adder is defined with the + and { } operators.

Example 3.16: 4-bit Full Adder, Using Dataflow Operators

```
// Define a 4-bit full adder by using dataflow statements.
module fulladd4(sum, c_out, a, b, c_in);
// I/O port declarations output [3:0] sum;
output c_out;
input[3:0] a, b;
input c_in;
// Specify the function of a full adder
assign {c_out, sum} = a + b + c_in;
endmodule
```

Method 2: full adder with carry lookahead

In ripple carry adders, the carry must propagate through the gate levels before the sum is available at the output terminals. An n-bit ripple carry adder will have 2n gate levels. The propagation time can be a limiting factor on the speed of the circuit. One of the most popular methods to reduce delay is to use a carry lookahead mechanism. Logic equations for implementing the carry lookahead mechanism can be found in any logic design book. The propagation delay is reduced to four gate levels, irrespective of the number of bits in the adder. The Verilog description for a carry lookahead adder. This module can be substituted in place of the full adder modules described before without changing any other component of the simulation. The simulation results will be unchanged.

Example 3.17: 4-bit Full Adder with Carry Lookahead

```
module fulladd4(sum, c_out, a, b, c_in);
// Inputs and outputs
output [3:0] sum;
output c_out;
```

```
input [3:0] a,b;
input c_in;
// Internal wires
wire p0,g0, p1,g1, p2,g2, p3,g3; wire c4, c3, c2, c1;
// compute the p for each stage assign
p0 = a[0] ^ b[0],
p1 = a[1] ^ b[1],
p2 = a[2] ^ b[2],
p3 = a[3] ^ b[3];
// compute the g for each stage assign
g0 = a[0] & b[0],
g1 = a[1] & b[1],
g2 = a[2] & b[2],
g3 = a[3] & b[3];
// compute the carry for each stage
assign c1 = g0 | (p0 & c_in),
        c2 = g1 | (p1 & g0) | (p1 & p0 & c_in),
c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & c_in),
c4 = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0) | (p3 & p2 & p1 & p0 & c_in);
// Compute Sum
assign sum[0] = p0 ^ c_in, sum[1] = p1 ^ c1, sum[2] = p2 ^ c2, sum[3] = p3 ^ c3;
// Assign carry output
assign c_out = c4;
endmodule
```

3.8.3 Ripple Counter

Consider the design of a 4-bit ripple counter by using negative edge-triggered flipflops. This example was discussed at a very abstract level, Hierarchical Modeling Concepts. We design it using Verilog dataflow statements and test it with a stimulus module. The diagrams for the 4-bit ripple carry counter modules are shown below. Figure 3.10 shows the counter being built with four T-flipflops. Figure 3.11 shows that T flip flop is built with one D flip flop and an inverter and figure 3.12 shows D flip flop constructed using basic gates.

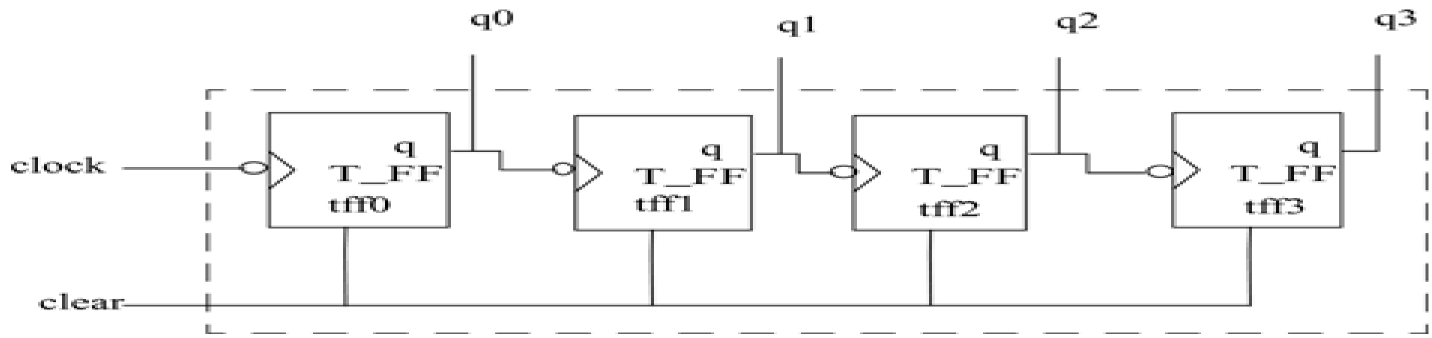
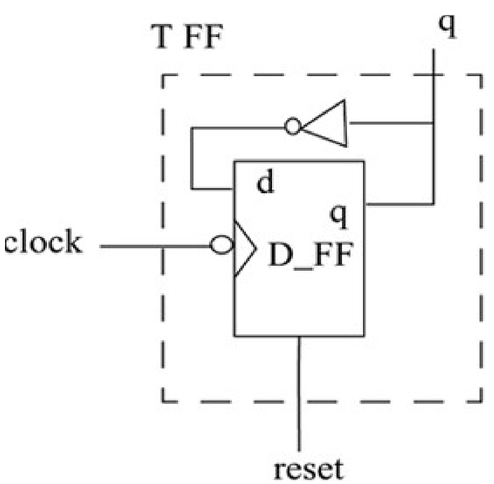


Figure 3.10: 4-bit ripple carry counter



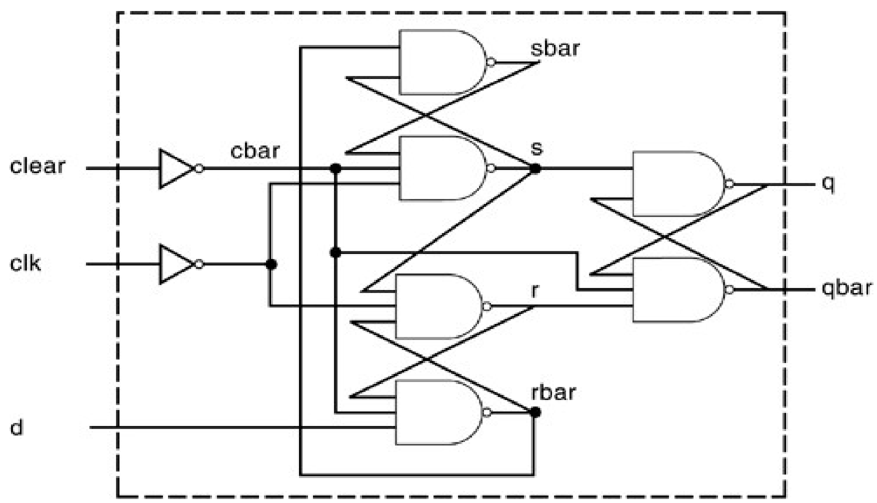


Figure 3.11: T- flip flop **Figure 3.12:** Negative edge triggered D flip flop

Given the above diagrams, we write the corresponding Verilog, using dataflow statements in a top-down fashion. First we design the module counter. The code is shown in. The code contains instantiation of four T_FF modules.

Example 3.18: Verilog code for ripple counter

```
module counter(Q , clock, clear);
// I/O ports
output [3:0] Q;
input clock, clear;
// Instantiate the T flipflops
T_FF tff0(Q[0], clock, clear);
T_FF tff1(Q[1], Q[0], clear);
T_FF tff2(Q[2], Q[1], clear);
T_FF tff3(Q[3], Q[2], clear);
endmodule
```

Example 3.19: Verilog Code for T-flipflop

```
// Edge-triggered T-flipflop. Toggles every clock cycle.
module T_FF(q, clk, clear);
// I/O ports
output q;
input clk, clear;
// Instantiate the edge-triggered DFF
// Complement of output q is fed back.
// Notice qbar not needed. Unconnected port.
edge_dff ff1(q, ,~q, clk, clear);
endmodule
```

Example 3.19: Verilog Code for Edge-Triggered D-flipflop

```
// Edge-triggered D flipflop
module edge_dff(q, qbar, d, clk, clear);
// Inputs and outputs
output q,qbar;
input d, clk, clear;
// Internal variables
Wire s, sbar, r, rbar, cbar;
// dataflow statements
//Create a complement of signal clear
assign cbar = ~clear;
// Input latches; A latch is level sensitive. An edge-sensitive flip-flop is implemented by using
// 3 SR latches.
assign sbar = ~(rbar & s), s = ~(sbar & cbar & ~clk), r = ~(rbar & ~clk & s), rbar = ~(r & cbar
& d);
// Output latch
assign q = ~(s & qbar), qbar = ~(q & r & cbar);
endmodule
```

Example 3.20: Stimulus Module for Ripple Counter

```
// Top level stimulus module
module stimulus;
```

```
// Declare variables for stimulating input
reg CLOCK, CLEAR;
wire [3:0] Q;
initial
$monitor($time, " Count Q = %b Clear= %b", Q[3:0],CLEAR);
// Instantiate the design block counter
counter c1(Q, CLOCK, CLEAR);
// Stimulate the Clear Signal
initial
begin
CLEAR = 1'b1;
#34 CLEAR = 1'b0;
#200 CLEAR = 1'b1;
#50 CLEAR = 1'b0;
end
// Set up the clock to toggle every 10 time units
initial
begin
CLOCK = 1'b0;
forever #10 CLOCK = ~CLOCK;
end
// Finish the simulation at time 400
initial
begin
#400 $finish;
end
endmodule
```

The output of the simulation is shown below. Note that the clear signal resets the count to zero.

```
0 Count Q = 0000 Clear= 1
34 Count Q = 0000 Clear= 0
40 Count Q = 0001 Clear= 0
```

60 Count Q = 0010 Clear= 0

80 Count Q = 0011 Clear= 0

100 Count Q = 0100 Clear= 0

120 Count Q = 0101 Clear= 0

140 Count Q = 0110 Clear= 0

160 Count Q = 0111 Clear= 0

180 Count Q = 1000 Clear= 0

200 Count Q = 1001 Clear= 0

220 Count Q = 1010 Clear= 0

234 Count Q = 0000 Clear= 1

284 Count Q = 0000 Clear= 0

300 Count Q = 0001 Clear= 0

320 Count Q = 0010 Clear=0

340 Count Q = 0011 Clear= 0

360 Count Q = 0100 Clear= 0

380 Count Q = 0101 Clear= 0