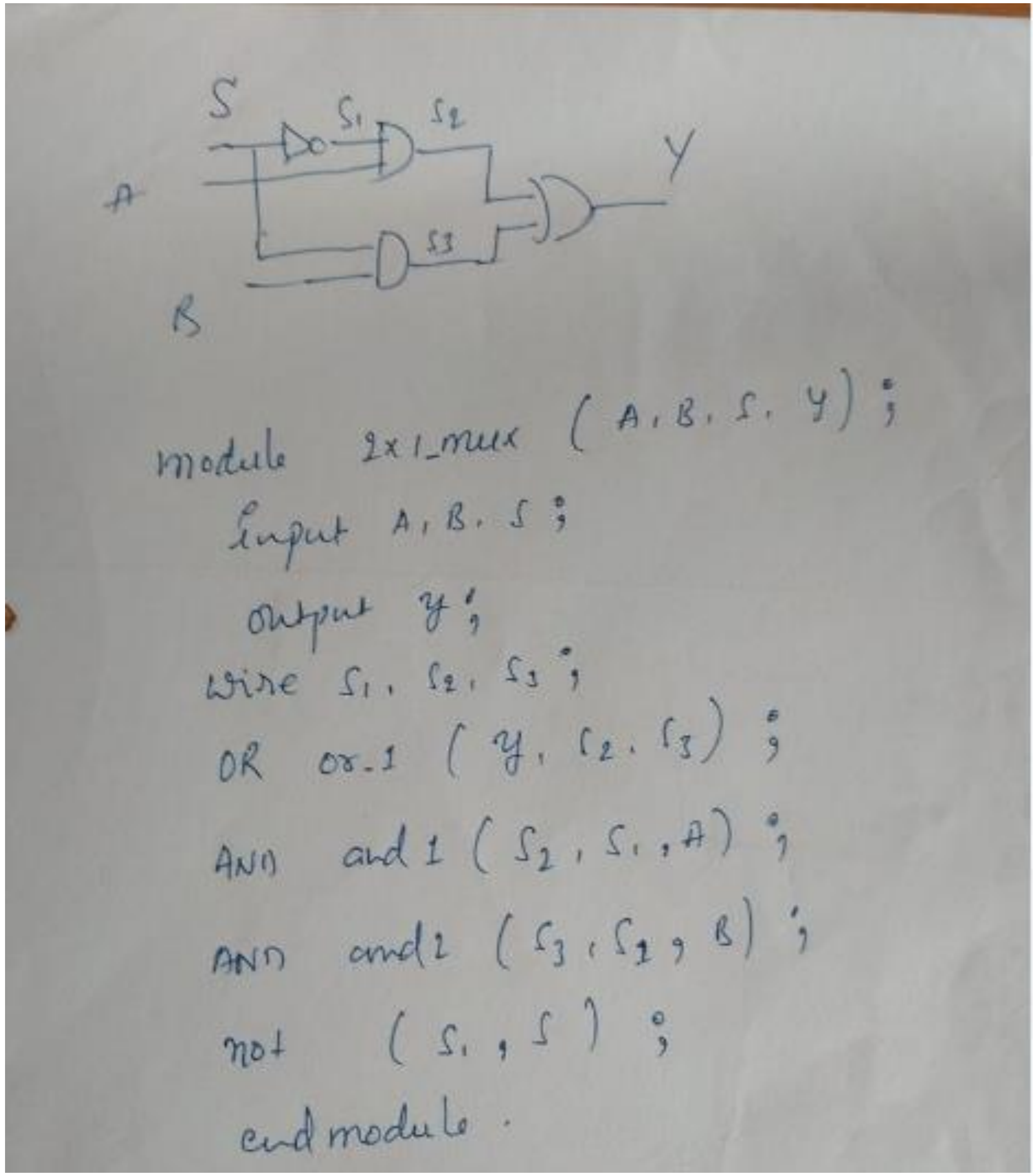
DIGITAL SYSTEM DESIGN USING VERILOG

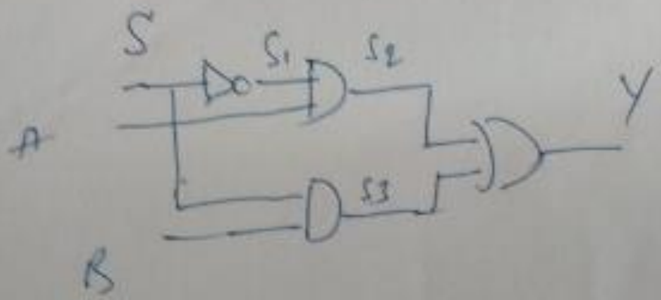
Course Code: 19EC5DCDSV
(3 Credits)

Module 2A

Faculty In-Charge: Dr. Dinesha P

drdinesh-ece@dayanandasagar.edu





```
module 2x1_mux (A, B, S, Y);
    input A, B, S;
    output y;
   wire S1, S2, S3;
   OR or_1 (y, S2, S3);
   AND and1 (S2, S1, A);
   AND and2 (S3, S1, B);
   not (S1, S);
    endmodule.
```

Delays in Verilog:

```
assign #5 D = A && B;
```

to model an AND gate with a propagation delay of 5ns (assuming its time unit is ns).

Basically, delays in Verilog can be categorized into two models: inertial delay and transport delay. The inertial delay for combinational blocks can be expressed in the following three ways:

```
// explicit continuous assignment
wire D;
assign #5 D = A && B;
// implicit continuous assignment
wire #5 D = A && B;
// net declaration
wire #5 D;
assign D = A && B;
```

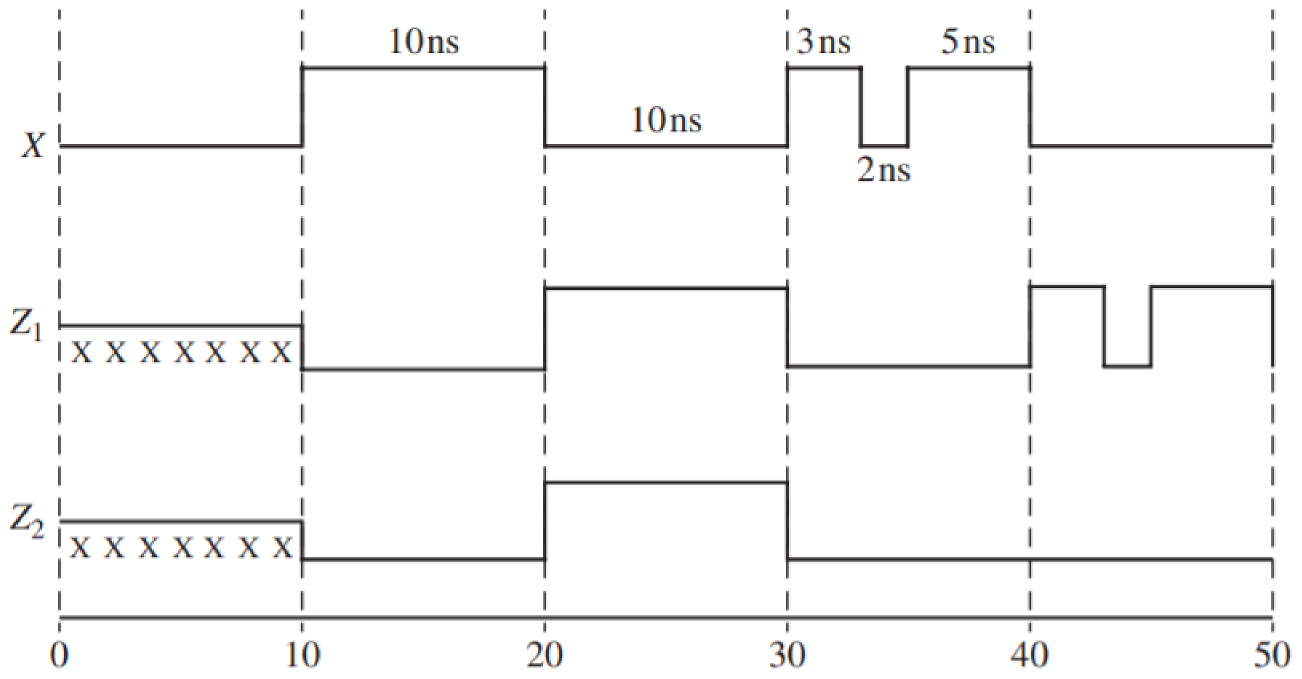
Any changes in A and B will result in a delay of 5ns before the change in output is visible. If values in A or B are changed 5ns before the evaluation of D output, the change in values will be propagated. However, an input pulse that is shorter than the delay of the assignment does not propagate to the output. This feature is called inertial delay.

Inertial delay is used to model gates and other devices that do not propagate short pulses from the input to the output. If a gate has an ideal inertial delay T, in addition to delaying the input signals by time T, any pulse with a width less than T is rejected. For example, if a gate has an inertial delay of 5ns, a pulse of width 5ns would pass through, but a pulse of width 4.999ns would be rejected.

Transport delay is intended to model the delay introduced by wiring; it simply delays an input signal by the specified delay time. In order to model this delay, a delay value must be specified on the right-hand side of the statement.

The following example illustrates transport delay and inertial delay in Verilog

```
always @ (X)
begin Z1 <= #10 (X);       // transport delay
end
assign #10 Z2 = X;          // inertial delay
```



- The first statement has transport delay while the second one has inertial delay. As shown in Figure, if the delay is shorter than 10ns, the input signal will not be propagated to the output in the second statement.
- Only one pulse (between 10ns and 20ns) on input X is propagated to the output Z2 , since it has 10ns pulse width. All other pulses are not propagated to the output Z2 .
- But Z1 has transport delay and hence propagates all pulses.
- It is assumed that the output Z1 and Z2 are initialized to 0 at 0ns.
- The delay in the statement Z1 <= #10 X; is called intra-assignment delay.
- The expression on the right hand side is evaluated but not assigned to Z1 until the delay has elapsed (also called delayed assignment). However, in a statement like #10 Z1 <= X; the delay of #10 elapses first and then the expression is evaluated and assigned to Z1 (also called delayed evaluation).

```
            assign a = #10 b;
```

Verilog also has a type of delay called net delay.

Consider the code

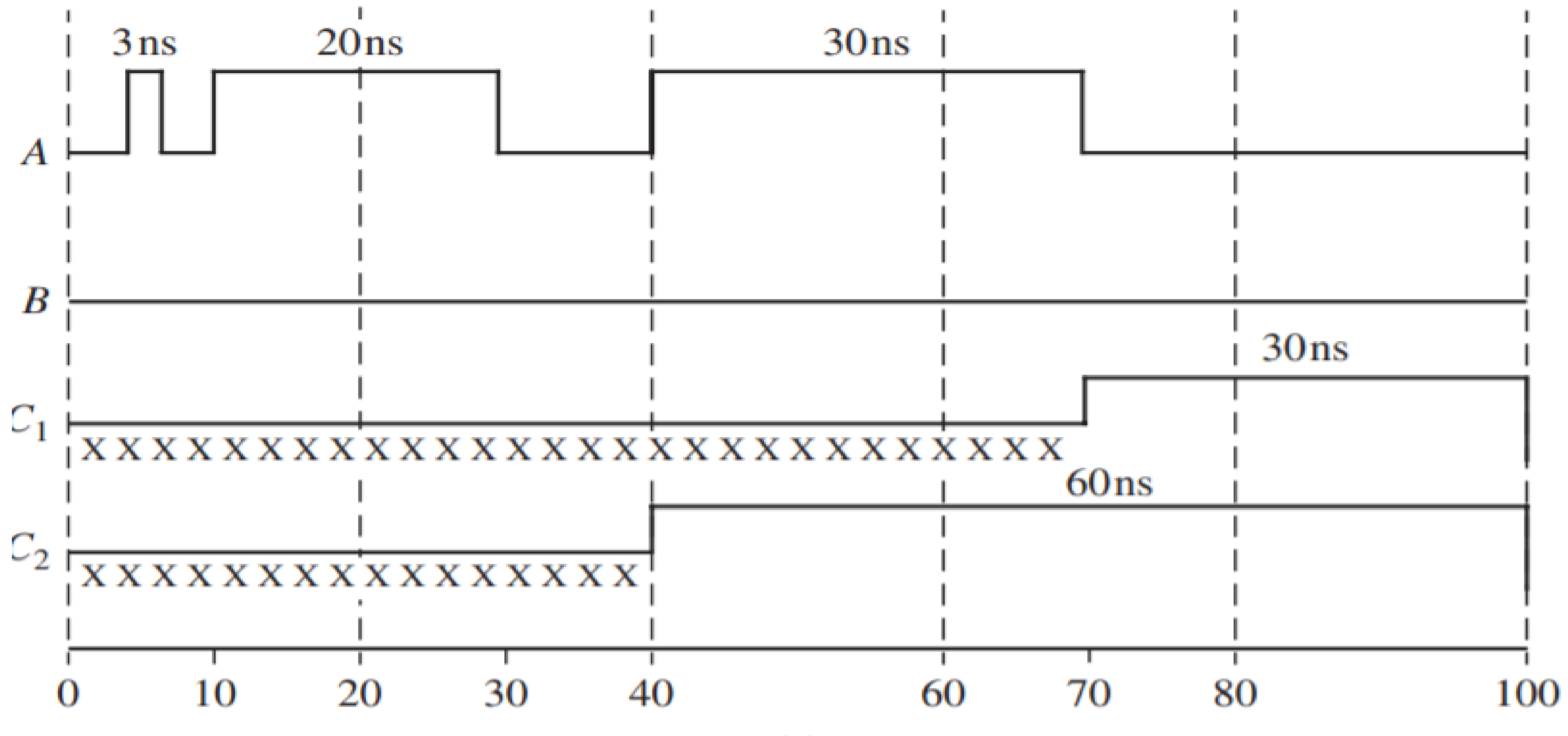
```
  wire C1;
  wire #10 C2; // net delay on wire C2
  assign #30 C1 = A || B;    // statement 1 – inertial delay
  assign #20 C2 = A || B;    // statement 2 – inertial delay will                                 be
                                  // added to net delay before being
                                  // assigned to wire C2
```

The wire C2 has a net delay of 10ns, whereas C1 has no such net delay.

Net delay refers to the time it takes from any driver on the net to change value to the time when the net value is updated and propagated further.

There are inertial delays of 30ns for C1 in statement 1 and 20ns for C2 in statement 2, typically representative of gate delays. After statement 2 processes its delay of 20ns, the net delay of 10ns is added to it.

- Figure indicates the difference between C1 and C2 . C1 rejects all narrow pulses less than 30ns, whereas C2 rejects only pulses less than 20 units.



(a)

consider the following two statement pairs with the Y waveform as shown in Figure.

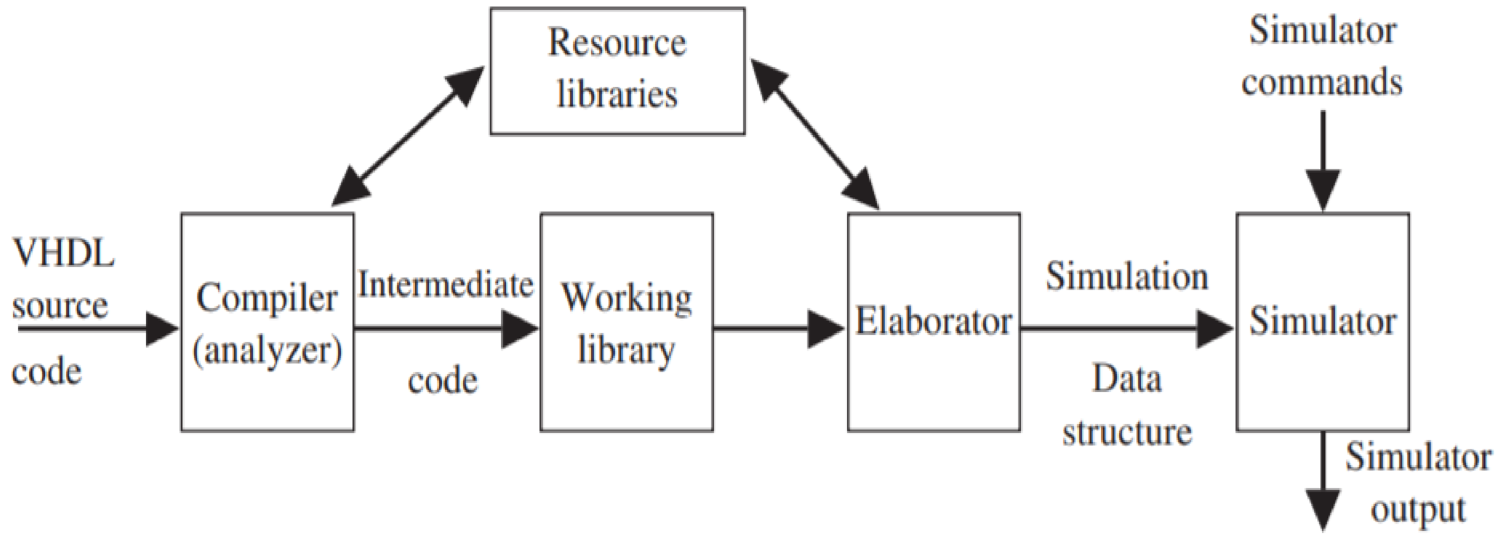
```
wire #3 D; // net delay on wire D
assign #7 D = Y; // statement 1 – inertial delay
wire #7 E; // net delay on wire E
assign #3 E = Y; // statement 1 – inertial delay
```

The assign statement for D works with a 7ns inertial delay and rejects any pulse below 7ns. Hence D rejects the 3ns, 2ns and 5ns pulses in Y. The 3ns net delay from the wire statement is added to the signal that comes out from the assign statement.

In the case of E, pulses below 3ns are rejected. Hence the 3ns pulse in Y passes through the assign statement for E, the 2ns pulse is rejected and the 5ns pulse is accepted.

Hence the 3ns and 5ns pulses get combined in the absence of the 2ns pulse to yield output on E appears as a big 10ns pulse. The 7ns net delay from the wire statement is added to the signal that comes out from the assign statement. If any pulses less than 7ns are encountered at the net delay phase, they will be rejected.

Compilation, Simulation, and Synthesis of Verilog Code

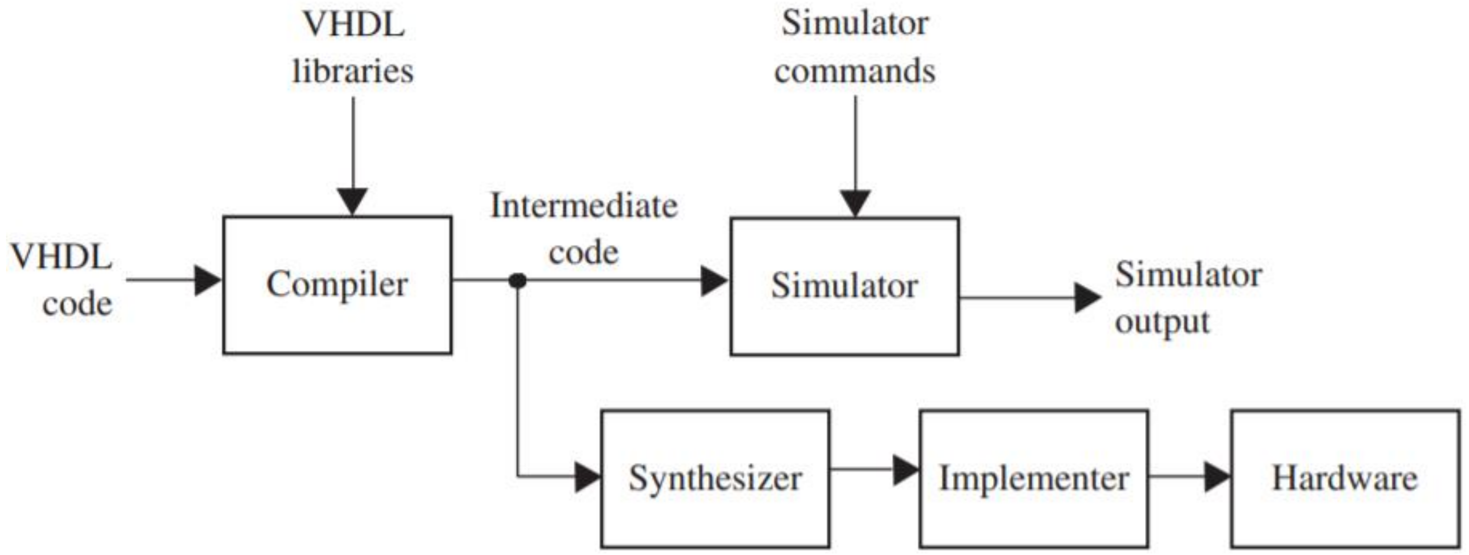


Simulation of the Verilog code is important for two reasons. First, we need to verify that the Verilog code correctly implements the intended design, and second, we need to verify that the design meets its specifications.
We first simulate the design and then synthesize it to the target technology (FPGA or custom ASIC).

There are three phases in the simulation of Verilog code: analysis (compilation), elaboration, and simulation.

- Before simulation, the Verilog code must first be compiled. The Verilog compiler, also called an analyzer, first checks the Verilog source code to see that it conforms to the syntax and semantic rules of Verilog. If there is a syntax error, such as a missing semicolon, or if there is a semantic error, such as trying to add two signals of incompatible types, the compiler will output an error message. The compiler also checks to see that references to libraries are correct. If the Verilog code conforms to all of the rules, the compiler generates intermediate code, which can be used by a simulator or by a synthesizer.
- Next, the design must have the modules being instantiated linked to the modules being defined, the parameters propagated among the various modules, and hierarchical references resolved. This phase in understanding a Verilog description is referred to as **elaboration**. During elaboration, a driver is created for each signal. Each driver holds the current value of a signal and a queue of future signal values.
- The simulation process consists of an initialization phase and actual simulation. The simulator accepts simulation commands, which control the simulation of the digital system and which specify the desired simulator output. Verilog simulation uses what is known as discrete event simulation.

One of the most important uses of Verilog is to synthesize or automatically create hardware from a Verilog description. The synthesis software for Verilog translates the Verilog code to a circuit description that specifies the needed components and the connections between the components. The initial steps (analysis and elaboration) in Figure are common whether Verilog is used for simulation or synthesis. The simulation and synthesis processes are shown in following Figurer.



- After the Verilog code for a digital system has been simulated to verify that it works correctly, the Verilog code can be synthesized to produce a list of required components and their interconnections, typically called the **netlist.**
- The synthesizer output can then be used to implement the digital system using specific hardware, such as a CPLD or an FPGA or as an ASIC. The CAD software used for implementation generates the necessary information to program the CPLD or FPGA hardware.

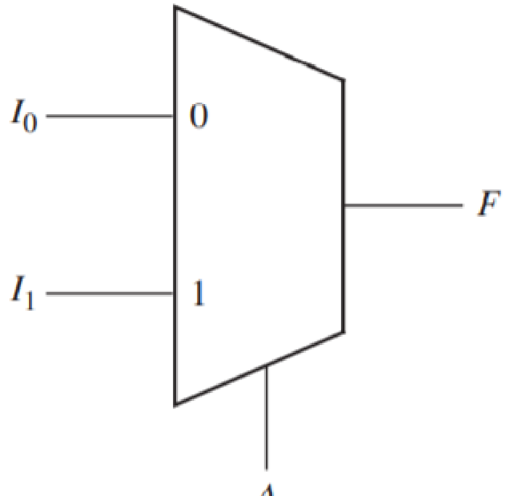
Verilog Models for Multiplexers:

A multiplexer is a combinational circuit and can be modeled using concurrent statements only or using always statements. A conditional operator with assign statement can be used to model a multiplexer without always statements. A case statement or if-else statement can also be used to make a model for a multiplexer within an always statement.

Using Conditional Operator:

Figure shows a 2-to-1 multiplexer (MUX) with 2 data inputs and one control input. The MUX output is F = A`.I_0 + A. I_1. The corresponding Verilog statement is

```
assign F = (~A && I0) || (A && I1);
```



```
// conditional signal assignment
statement
    assign F = (A) ? I1 : I0;
```

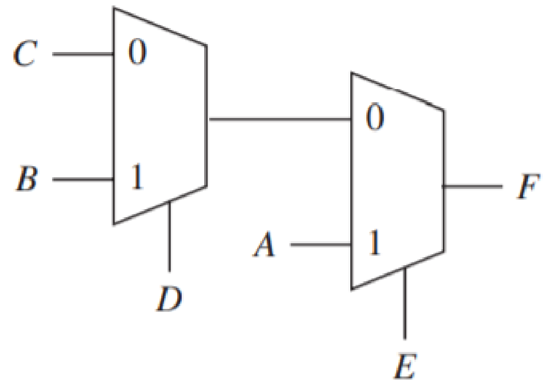
- Alternatively, we can represent the MUX by a conditional signal assignment statement as shown in Figure. This statement executes whenever A, I_0 , or I_1 changes. The MUX output is I_0 when A = 0; otherwise it is I_1 . In the conditional statement, I_0 , I_1 , and F can be one or more bits.

- The general form of a conditional signal assignment statement.

assign signal_name = condition ? expression_T : expression_F;

This concurrent statement is executed whenever a change occurs in a signal used in one of the expressions or conditions. If condition is true, signal_name is set equal to the value of expression_T. Otherwise if condition is false, signal_name is set equal to the value of expression_F.

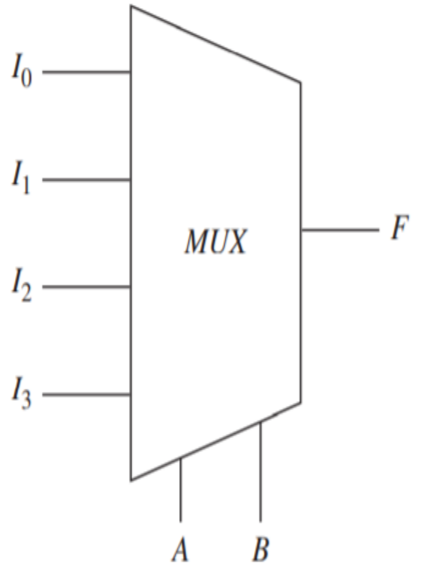
Figure shows how two cascaded MUXes can be represented by a conditional signal assignment statement. The output MUX selects A when the condition E is true; otherwise, it selects the output of the first MUX, which is B when the condition D is true, or it is C.



```
assign F = E ? A : ( D ? B : C );
// nested conditional assignment
```

Figure 2-38 shows a 4-to-1 multiplexer (MUX) with four data inputs and two control inputs, A and B. The control inputs select which one of the data inputs is transmitted to the output. The logic equation for the 4-to-1 MUX is

$$F = A'B'I_0 + A'B\,I_1 + A\,B'I_2 + A\,B\,I_3$$



One way to model the MUX is with the Verilog statement

```
assign F = (~A && ~B && I0) || (~A && B && I1) ||
           A && ~B && I2) || (A && B && I3);
```

Another way to model the 4-to-1 MUX is to use a conditional assignment statement:

```
assign F = (A) ? (B ? I3 : I2) : ( B ? I1 : I0 );
```

Using If-else or Case Statement in an Always Block:

If a MUX model is used inside an always statement, a concurrent statement cannot be used.

The MUX can be modeled using a case statement within an always block:

```
always @ (Sel or I0 or I1 or I2 or I3)
        case Sel
        2'b00 : F= I0;
        2'b01 : F = I1;
         2'b10 : F 5 I2;
        2'b11 : F 5 I3;
        endcase
```

MUX has four input signals,

the selection signal, Sel should be a 2-bit signal.

The selection signals are represented as 2'b00, 2'b01, 2'b10, and 2'b11

The b represents that the base is binary here

The case statement has the general form:

```
Case
expression choice1 : sequential statements1
choice2 : sequential statements2
. . .
[default : sequential statements]
endcase;
```

The expression is evaluated first. If it is equal to choice1, then sequential statements1 are executed; if it is equal to choice2, then sequential statements2 are executed; and so forth. If all values are not explicitly given, a default clause is required in the case statement.

the MUX can also be modeled using an if-else statement within an always block:

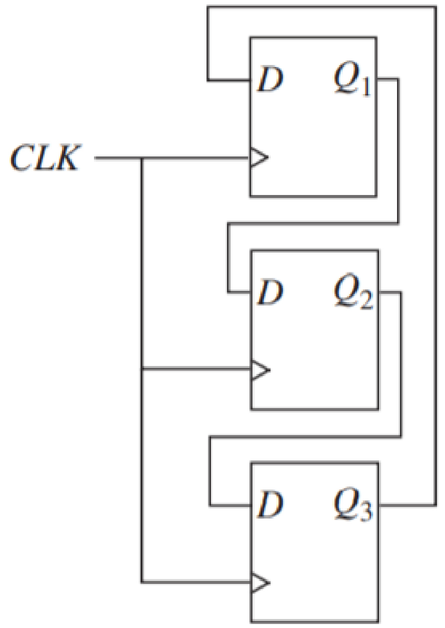
```
always @ (Sel or I0 or I1 or I2 or I3)
begin
 if (Sel == 2'b00)              F = I0;
else if (Sel == 2'b01)          F = I1;
else if (Sel == 2'b10)          F = I2;
else if (Sel == 2'b11)          F = I3;
end
```

Modeling Registers and Counters Using Verilog Always Statements:

Figure shows three flip-flops connected as a cyclic shift register (or a rotating shift register). These flip-flops all change state following the rising edge of the clock.

assumed a 5ns propagation delay between the clock edge and the output change

Immediately following the clock edge, the three statements in the always statement execute in sequence with no delay. Then the sequential statements are Q1 <= Q3; Q2 <= Q1; Q3 <= Q2;



```
always @ (posedge CLK)
begin
   Q1 <= #5 Q3;
   Q2 <= #5 Q1;
   Q3 <= #5 Q2;
end
```

- The order of the statements is not important when the non-blocking assignment operator “<= ” is used. The same result is obtained even if the statements are in reverse order as shown here.

```
always @ (posedge CLK)
begin
Q3 <= #5 Q2;
Q2 <= #5 Q1;
Q1 <= #5 Q3;
end
```

Example 1: What is the hardware obtained if the following code is synthesized?

```
module reg3 (Q1,Q2,Q3,A,CLK);
input A;
input CLK;
output Q1,Q2,Q3;
reg Q1,Q2,Q3;
always @(posedge CLK)
begin
Q3 = Q2; // statement 1
 Q2 = Q1; // statement 2
Q1 = A; // statement 3
end
endmodule
```

In the above example, the list of statements executes from top to bottom in order. Note that the blocking operator is used. Therefore, the first statement finishes update before the second statement is executed. Synthesis results in a 3-bit shift register with serial input A, and outputs Q1 , Q2 , and Q3 .

Answer: A 3-bit shift register

Example2: What is the hardware obtained if the following code is synthesized?

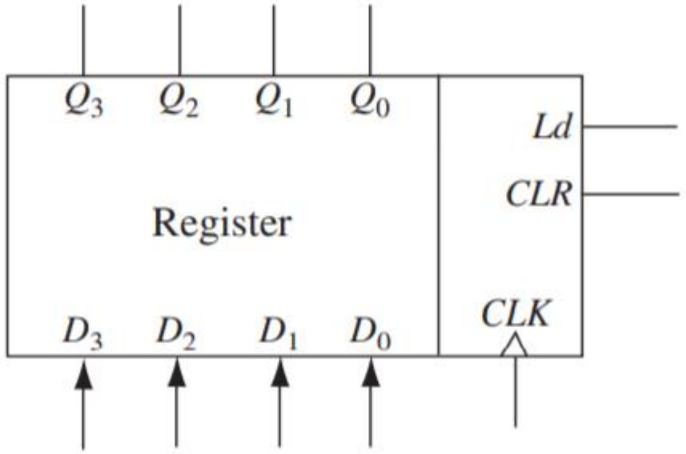
```
module reg31 (Q1,Q2,Q3,A,CLK);
input A;
 input CLK;
output Q1,Q2,Q3;
reg Q1,Q2,Q3;
always @(posedge CLK)
begin
Q1 = A; // statement 1
Q2 = Q1; // statement 2
Q3 = Q2; // statement 3
end
endmodule
```

Explanation: In this example, blocking operator is used, so the list of statements executes from top to bottom in order. So the first statement finishes update before the second statement is executed. Q1 gets the value of the serial input A when statement 1 finishes. In statement 2, the same value propagates to Q2 . In statement 3, the same value propagates to Q3 . In effect, the input A has reached Q3 . Modern synthesis tools will generate a single flip-flop with input A when this code is synthesized. The outputs Q1 , Q2 , and Q3 can all be connected to the output of the same flip-flop. If the synthesizer does not have good optimization algorithms, it might generate three parallel flip-flops, each with the same input A but with outputs Q1 , Q2 , and Q3 , respectively.

Ans: A single flip-flop

Register with Synchronous Clear and Load:

Figure shows a simple register that can be loaded or cleared on the rising edge of the clock. If CLR is set to 1, the register is cleared, and if Ld = 1, the D inputs are loaded into the register. This register is fully synchronous so that the Q outputs change only in response to the clock edge and not in response to a change in Ld or CLR.



```
always @ (posedge CLK)
begin
 if (CLR)          Q <= 4'b0000;
 else if (Ld)      Q <= D;
end
```

- In the Verilog code for the register, Q and D are 4-bit vectors dimensioned [3:0]. Since the register outputs can only change on the rising edge of the clock, CLR and Ld are not on the sensitivity list. The CLR and Ld signals are tested after the rising edge of the clock.

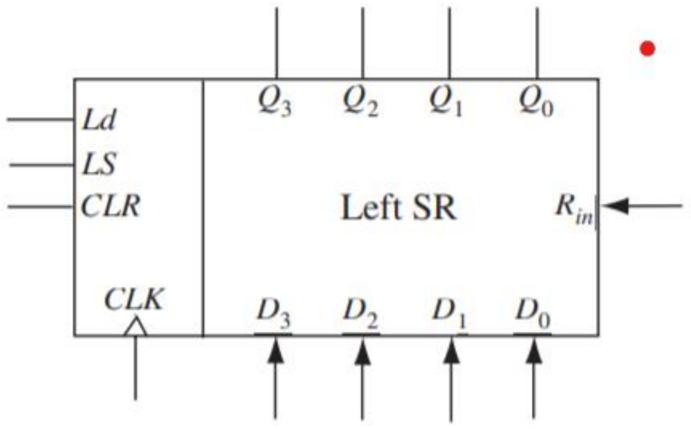
Left Shift Register with Synchronous Clear and load:

Model a left shift register using a Verilog always statement. A left shift control input (LS) is used in addition to Ld, CLR control input.

When LS = 1, the contents of the register are shifted left and the right-most bit is set equal to Rin. The shifting is accomplished by taking the rightmost 3 bits of Q, Q[2:0], and concatenating them with Rin.

For example, if Q = 1101 and Rin = 0, then {Q[2:0], Rin} = 1010, and this value is loaded back into the Q register on the rising edge of CLK.

If CLR = Ld =LS = 0, then Q remains unchanged.

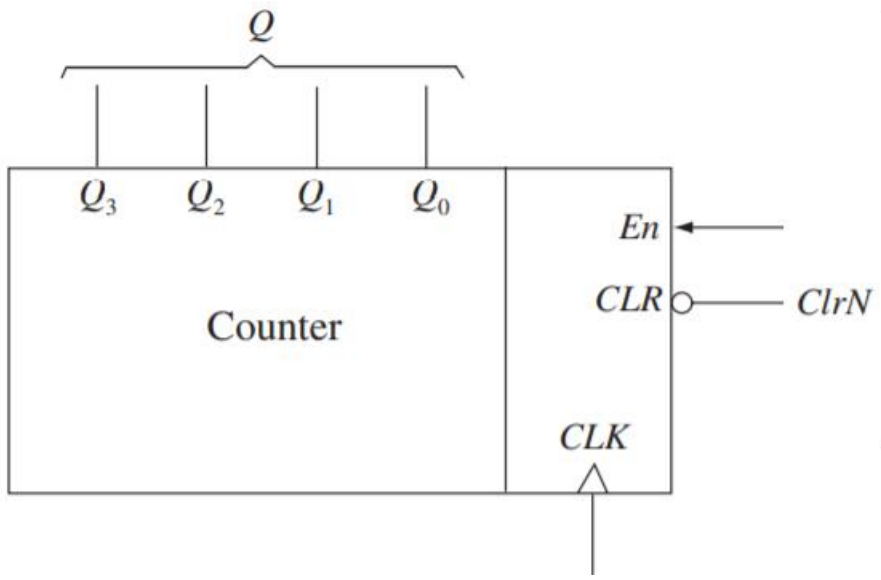


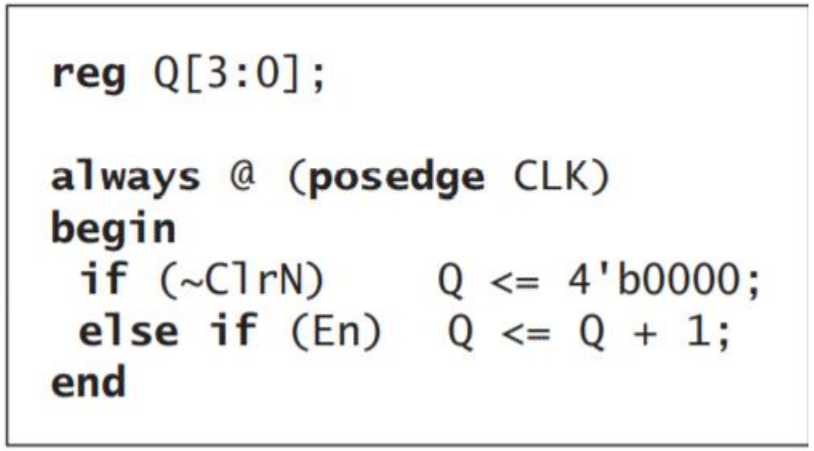
```
always @ (posedge CLK)
begin
 if (CLR)          Q <= 4'b0000;
 else if (Ld)      Q <= D;
 else if (LS)      Q <= {Q[2:0], Rin};
end
```

Synchronous counter:

Figure shows a simple synchronous counter. On the rising edge of the clock, the counter is cleared when ClrN =0, and it is incremented when ClrN = En = 1. In this example, the signal Q represents the 4-bit value stored in the counter.

The signal Q is declared to be of type **reg** with length of 4 bits. Then Q <= Q+1; increments the counter. When the counter is in state 1111, the next increment takes it back to state 0000.





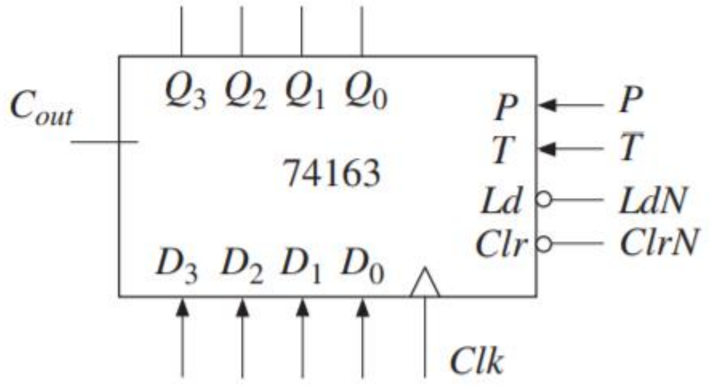
```
reg Q[3:0];

always @ (posedge CLK)
begin
 if (~ClrN)      Q <= 4'b0000;
 else if (En)    Q <= Q + 1;
end
```

Verilog model for a standard MSI counter (74163):

It is a 4-bit fully synchronous binary counter, which is available in both TTL and CMOS logic families.

It can be cleared or loaded in parallel.

All operations are synchronized by the clock, and all state changes take place following the rising edge of the clock input.



Control Signals			Next State				
ClrN	*LdN*	*PT*	Q_3^+	Q_2^+	Q_1^+	Q_0^+	
0	X	X	0	0	0	0	(clear)
1	0	X	D_3	D_2	D_1	D_0	(parallel load)
1	1	0	Q_3	Q_2	Q_1	Q_0	(no change)
1	1	1	present state + 1				(increment count)

This counter has four control inputs—ClrN, LdN, P, and T. Both P and T are used to enable the counting function.

While P is an actual enable signal to the 4-bit generic counter, T is used for a carry connection signal when cascading multiple counters.

Operation of the counter is as follows:

1. If ClrN = 0, all flip-flops are set to 0 following the rising clock edge.
2. If ClrN = 1 and LdN = 0, the D inputs are transferred (loaded) in parallel to the flip-flops following the rising clock edge.
3. If ClrN = LdN = 1 and P = T = 1, the count is enabled and the counter state will be incremented by 1 following the rising clock edge.

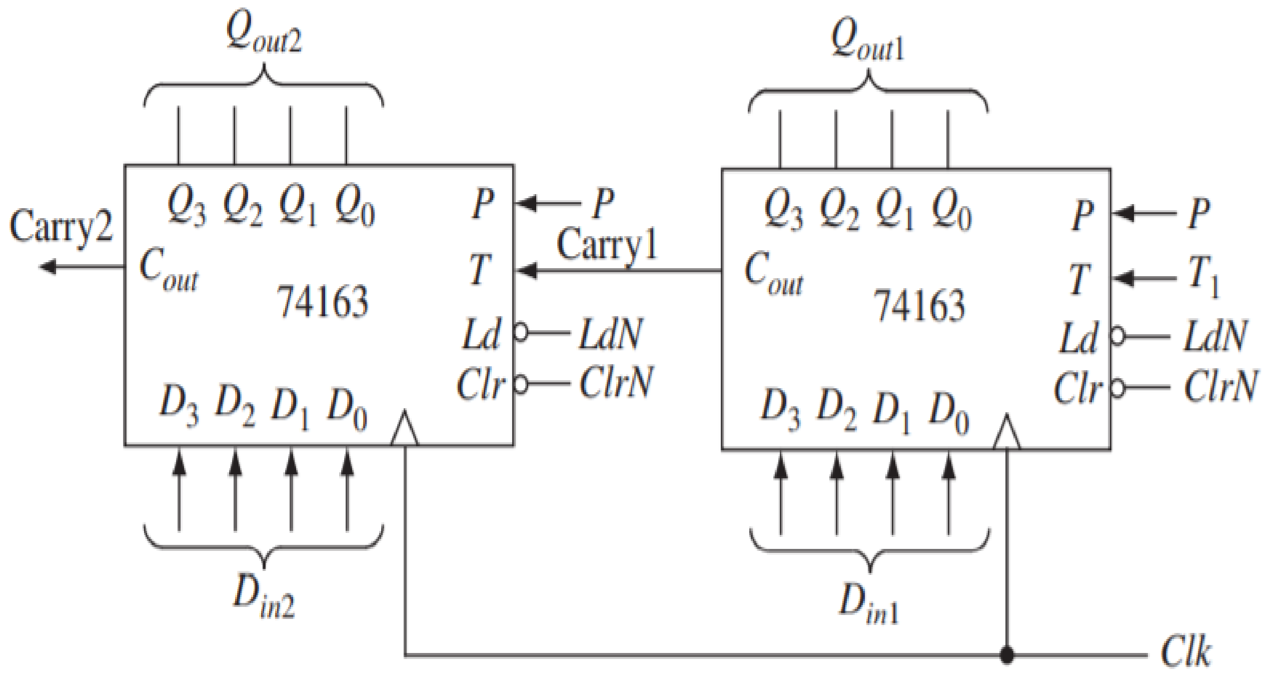
If T = 1, the counter generates a carry (Cout) in state 15; consequently

$$Cout = Q3\ Q2\ Q1\ Q0\ T$$

```
// 74163 FULLY SYNCHRONOUS COUNTER
        module c74163 (LdN, ClrN, P, T, Clk, D, Cout, Qout);
        input        LdN;
        input        ClrN;
         input        P;
        input        T;
        input        Clk;
        input        [3:0] D;
        output       Cout;
        output [3:0]  Qout;
        reg [3:0]     Q;
        assign Qout = Q;
        assign Cout = Q[3] & Q[2] & Q[1] & Q[0] & T;
        always @ (posedge Clk)
        begin
        if (~ClrN)              Q <= 4'b0000;
        else if (~LdN)           Q <= D;
        else if (P & T)          Q <= Q + 1;
        end
        endmodule
```

Two 74163 Counters Cascaded to Form an 8-Bit Counter:

To test the counter, we have cascaded two 74163s to form an 8-bit counter. When the counter on the right is in state 1111 and T1 = 1, Carry1 = 1. Then for the left counter, PT = 1 if P = 1. If PT = 1, on the next clock the right counter is incremented to 0000 at the same time the left counter is incremented.



Verilog for 8-Bit Counter Using 4-Bit Counter Modules

```
// 8-Bit counter using two 74163 counters using the model in Fig 2-44
module eight_bit_counter(ClrN, LdN, P, T1, Clk, Din1, Din2, Count, Carry2);
input ClrN;
input LdN;
 input P;
input T1;
input Clk;
input [3:0]      Din1;
 input [3:0]     Din2;
output [7:0]     Count;
 output Carry2;
 wire Carry1;
 wire [3:0] Qout1;
wire [3:0] Qout2;
 c74163 ct1 (LdN, ClrN, P, T1, Clk, Din1, Carry1, Qout1);
 // instance 1 (right)
c74163 ct2 (LdN, ClrN, P, Carry1, Clk, Din2, Carry2, Qout2);
 //instance 2 (left)
assign Count = {Qout2, Qout1};
 endmodule
```

In this code, used the c74163 model as a component and have instantiated two copies of it. The instantiation for the lower four bits is done using the statement

```
c74163 ct1(LdN, ClrN, P, T1, Clk, Din1, Carry1, Qout1);
```

Loops in Verilog:

A loop statement is a sequential statement. Verilog has several kinds of loop statements including for loops and while loops. There is also a repeat loop.

Forever Loop (Infinite Loop)

Infinite loops are useful in hardware modeling where a device works continuously and continues to work until the power is off. Below is an example for a forever loop:

```
begin
 clk = 1'b0;
forever #10 clk = ~clk;
end
```

For Loop:

The general form of a for loop is as follows:

```
for (initial_statement; expression; incremental_statement)
begin
sequential statement(s);
end
```

The following example is of initializing an array variable, eight_bit_register_ array, using a for loop.

```
reg [7:0] eight_bit_register_array [15:0];
for (i=0; i<16; i=i+1)
begin
register A[i] = 8'b00000000;
 end
```

This type of loop is generally used in behavioral models. The following is for a 4-bit adder.

```
for (i=0; i<4; i=i+1)
begin
Cout = (A[i] && B[i]) || (A[i] && Cin) || (B[i] && Cin);
 sum[i] = A[i] ^ B[i] ^ Cin;
Cin = Cout;
end
```

While Loop:

In while loops a condition is tested before each iteration. The loop is terminated if the condition is false. The general form of a while loop is

```
while (condition)
 begin
sequential statements;
 end
```

Example:

```
While (i<x)
Begin
        i=i+1;
        z=i*z;
end
```

Repeat Loop:

The repeat loop repeats the sequential statement(s) for specified times. The number of repetitions is set by a constant value or a logical expression.

Ex1:
```
repeat( 8 )
begin
x = x + 1;
 y = y + 2;
 end
```

Ex2:
```
Repeat (32)
begin
        #100 i=i+1;
end
```

Constants:

Verilog provides constants in addition to variables and nets. Verilog provides three different ways to define constant values, one of which is using `define as in

` define constant_name constant_value

The `define is one of the compiler directives in Verilog and is used to define a number or an expression for a meaningful string.

The `define compiler directive replaces 'constant_name with constant_ value.

For example:

```
`define wordsize 16
 reg [1:`wordsize] data;
```

causes the string wordsize to be replaced by 16. It then shows how data is declared to be a reg of width wordsize.

Another method to create constants is to use the parameter keyword as follows:

parameter constant_name = constant_value;

For example,

parameter msb = 15; // defines msb as a constant value 15

parameter [31:0] decim = 1'b1; // value converted to 32 bits

Another method to make constants is using localparam.

localparam constant_name = constant_value;

The localparam is similar to the parameter, but it cannot be directly changed. The localparam can be used to define constants that should not be changed.

Arrays:

Arrays in Verilog can be used while modeling the repetition. Digital systems often use memory arrays. Verilog arrays can be used to create memory arrays and specify the values to be stored in these arrays.

In order to use an array in Verilog, we must declare the array upper and lower bound. There are two positions to declare the array bounds:

In one option, the array bounds are declared between the variable type (reg or net) and the variable name, and the array bound means the number of bits for the declared variable. If the array bound is defined as [7:0] as shown in the following example,

```
reg [7:0] eight_bit_register;
```

the register variable eight_bit_register can store one byte (eight bits) of information.

The 8-bit register can be initialized to hold the value 00000001 using the following statement:

```
eight_bit_register = 8'b00000001;
```

array bounds can be declared after the name of the array.

```
reg rega [1:n]; // This is an array of n 1-bit registers
reg [1:n] regb; // This is an n-bit register
```

We can define multiple 8-bit registers in one array declaration. In this, 16 registers are declared; each register can store onebyte (8-bit) vector information.

```
reg [7:0] eight_bit_register_array [15:0];
```

This array can be initialized as follows:

```
eight_bit_register_array[15] = 8'b00001100;
eight_bit_register_array[14] = 8'b00000000;
 . . . . . . . .
eight_bit_register_array[1] = 8'b11001100;
eight_bit_register_array[0] 5 8'b00010001;
```

Arrays can be created of various data types. Arrays of wires and integers can be declared as follows:

```
wire wire_array[5:0]; // declares an array of 6 wires
 integer inta[1:64]; // declares an array of 64 integer values
```

Matrices:

Multidimensional array types may also be defined with two or more dimensions.

The following example, defines a 2-dimensional array variable in an initial statement, which is a matrix of integers with four rows and three columns with 8-bit elements:

```
reg [7:0] matrixA [0:3][0:2] = { { 1, 2, 3},
                                 { 4, 5, 6},
                                  { 7, 8, 9},
                                 {10, 11, 12}};
```

Look-Up Table Method Using Arrays and Parameters:

The array construct together with parameter can be used to create look-up tables which can be used to create combinational circuits using the ROM or Look-up Table (LUT) method.

Example:

Parity bits are often used in digital communication for error detection and correction. The simplest of these involve transmitting one additional bit with the data, a parity bit. Use Verilog arrays to represent a parity generator that generates a 5-bit-odd-parity generation for a 4-bit input number using the look-up table (LUT) method

Answer: The input word is a 4-bit binary number. A 5-bit odd-parity representation will contain exactly an odd number of 1s in the output word. This can be accomplished by the ROM or LUT method using a look-up table of size 16 entries × 5 bits. The look-up table is indicated in following table.

LUT Contents for a Parity Code Generator

Input (LUT Address)				Output (LUT Data)				
A	B	C	D	P	Q	R	S	T
0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	1	0
0	0	1	0	0	0	1	0	0
0	0	1	1	0	0	1	1	1
0	1	0	0	0	1	0	0	0
0	1	0	1	0	1	0	1	1
0	1	1	0	0	1	1	0	1
0	1	1	1	0	1	1	1	0
1	0	0	0	1	0	0	0	0
1	0	0	1	1	0	0	1	1
1	0	1	0	1	0	1	0	1
1	0	1	1	1	0	1	1	0
1	1	0	0	1	1	0	0	1
1	1	0	1	1	1	0	1	0
1	1	1	0	1	1	1	0	0
1	1	1	1	1	1	1	1	1

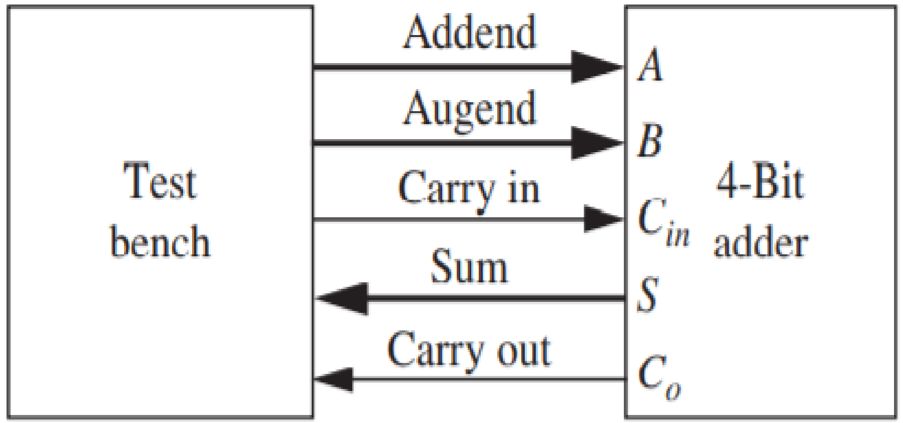
```
module parity_gen (X, Y);
 input [3:0]  X;
output [4:0]  Y;
wire     ParityBit;
 parameter [0:15] OT = {1'b1, 1'b0, 1'b0, 1'b1,
1'b0, 1'b1, 1'b1, 1'b0, 1'b0, 1'b1, 1'b1, 1'b0, 1'b1,
1'b0, 1'b0, 1'b1};
assign ParityBit = OT[X];
 assign Y = {X, ParityBit};
 endmodule
```

Testing a Verilog Model:

Once a Verilog model for a system has been developed, the next step is to test it. A model has to be tested and validated before it can be successfully used.

A test bench is a piece of Verilog code that can provide input combinations to test a Verilog model for the system under test. It provides stimuli to the system or circuit under test. Test benches are frequently used during simulation to provide sequences of inputs to the circuit or Verilog model under test.

Test bench for testing the 4-bit binary adder



Test Bench for 4-Bit Adder:

```
module TestAdder_v2; //Statement 1
parameter N = 11; //Statement 2
reg [3:0] addend;
reg [3:0] augend;
reg cin;
wire [3:0] sum;
wire cout;
 reg [3:0] addend_array[1:N];
 reg [1:N] cin_array;
reg [3:0] augend_array[1:N];
reg [3:0] sum_array[1:N];
reg [1:N] cout_array;
```

Test Bench for 4-Bit Adder:

```
module TestAdder_v2; //Statement 1
parameter N = 11; //Statement 2
reg [3:0] addend;
reg [3:0] augend;
reg cin;
wire [3:0] sum;
wire cout;
 reg [3:0] addend_array[1:N];
 reg [1:N] cin_array;
reg [3:0] augend_array[1:N];
reg [3:0] sum_array[1:N];
reg [1:N] cout_array;
```

```
//initialization of cin_array
 cin_array[1] = 1’b0;
cin_array[2] = 1’b0;
cin_array[3] = 1’b0;
cin_array[4] = 1’b0;
cin_array[5] = 1’b1;
cin_array[6] = 1’b0;
cin_array[7] = 1’b0;
cin_array[8] = 1’b0;
 cin_array[9] = 1’b1;
cin_array[10] = 1’b1;
 cin_array[11] = 1’b0;
```

```
//initialization of augend_array
augend_array[1] = 4’b0101;
augend_array[2] = 4’b0101;
 augend_array[3] = 4’b1101;
augend_array[4] = 4’b1101;
augend_array[5] = 4’b0111;
 augend_array[6] = 4’b0111;
augend_array[7] = 4’b1000;
augend_array[8] = 4’b1000;
augend_array[9] = 4’b1101;
augend_array[10] = 4’b1111;
augend_array[11] = 4’b0000;
```

```
//initialization of sum_array (expected sum outputs)
sum_array[1] = 4’b1100;
 sum_array[2] = 4’b0010;
 sum_array[3] = 4’b0010;
sum_array[4] = 4’b1010;
 sum_array[5] = 4’b1111;
sum_array[6] = 4’b1111;
sum_array[7] = 4’b1111;
sum_array[8] = 4’b0000;
sum_array[9] = 4’b1110;
sum_array[10] = 4’b1111;
 sum_array[11] = 4’b0000;
```

```
//initialization of cout_array (expected carry output)
cout_array[1] = 1'b0;
cout_array[2] = 1'b1;
cout_array[3] = 1'b1;
cout_array[4] = 1'b1;
 cout_array[5] = 1'b0;
cout_array[6] = 1'b0;
cout_array[7] = 1'b0;
 cout_array[8] = 1'b1;
cout_array[9] = 1'b0;
cout_array[10] = 1'b1;
cout_array[11] = 1'b0;
end
```

```
integer i;
 always
 begin
for(i = 1 ; i <= N ; i = i + 1)
 begin
$display(i);
addend <= addend_array[i];//apply an addend test vector
augend <= augend_array[i];//apply an augend test vector
cin <= cin_array[i];//apply a carry in #(40);//adder expected to take 40 time units
if(!(sum == sum_array[i] & cout == cout_array[i]))
Begin
 $write("ERROR: ");
 $display("Wrong Answer ");
 end
```

```
else
 begin
$display("Correct!!");
 end
end
 $display(“Test Finished”);
end
Adder4 add1(addend, augend, cin, sum, cout); //module under test instantiated endmodule
```

Input and output vectors are hard-coded into the test bench. An exhaustive test of the adder module requires 512 tests, since there are 9 input bits (4 addend; 4 augend and 1 carryin). We have a random set of 11 tests.

The $display and $write statements are used to print test results. In Verilog, a name following the $ is interpreted as a system task or a system function. The dollar sign ($) essentially introduces a language construct that enables the development of user-defined system tasks and functions.