

UNIT-5

Introduction to Verilog

Syllabus

5.1 Structural Specification of Logic Circuits

5.2 Behavioral Specification of Logic Circuits

5.3 Hierarchical Verilog Code

5.3.1 Four bit adder from one bit adder

5.4 Verilog Syntax for Combinational Circuits

5.4.1 Conditional Operator

5.4.2 If- else Statement

5.4.3 Case Statement

5.4.4 For loop

5.5 Using Storage Elements with CAD - Tools

5.5.1 Including storage elements in schematics

5.5.2 Using verilog constructs for storage elements

5.5.3 Blocking and Non-Blocking Assignments

5.5.3.1 Blocking Assignments

5.5.3.2 Non-Blocking Assignments

5.5.4 Flip-flops with clear capability

5.5.4.1 Asynchronous Clear

5.5.4.2 Synchronous Clear

5.5.5 Using Verilog constructs for registers and counters

5.5.5.1 A four bit Shift Register

5.5.5.2 Counter

5.6 Verilog code for sequential circuits

5.6.1 Registers

5.6.2 Asynchronous 4-bit Up Counter

5.6.3 Four bit Synchronous Up counter

5.7 Verilog Code for Combinational Circuits

5.7.1 8x3 Encoder

5.7.2 3x8 Decoder

5.7.3 8 x1 Multiplexer

5.7.4 1X8 De-multiplexer

I. Introduction:

- The design and specification of logic circuits form the backbone of digital electronics, influencing everything from simple gadgets to complex systems like microprocessors.
- Verilog, a hardware description language (HDL), plays a significant role in the design process by enabling the modeling and simulation of digital systems.
- In digital circuit design, structural specification refers to the detailed description of the components and their interconnections, while behavioral specification defines the functionality or the desired output behavior based on inputs. By using Verilog, designers can express both structural and behavioral aspects of circuits in a concise and efficient manner. The language supports the design of combinational circuits (e.g., multiplexers, decoders) and sequential circuits (e.g., flip-flops, counters), as well as their implementation using various Verilog constructs.

II. Applications

Applications of Verilog in Logic Circuit Design

- **Combinational Circuit Design:** Verilog enables the design of multiplexers (MUX) and de-multiplexers (DEMUX), essential for data routing and selection in digital systems. Also builds encoders and decoders, which are integral in communication systems, data compression, and error correction.
- **Sequential Circuit Design:** Sequential circuits often rely on storage elements like flip-flops and registers to store and process data. Using Verilog, designers can create flip-flops, registers (for data storage and shifting) and Asynchronous and synchronous counters can be designed with Verilog to keep track of time or events in digital systems. These counters are crucial in applications like timers, frequency division, and state machines.
- **Conditional and Loop Based Logic:** Verilog allows for the use of if-else statements, case statements, and for loops to model complex logic behavior in both combinational and sequential circuits. These constructs help to implement decision-making logic and repetitive tasks such as counting or data shifting.

- Behavioral and Structural Design Approaches: Behavioral Specification in Verilog is used to describe the high-level functionality of a circuit, whereas structural specification focuses on how different components are interconnected. By combining both approaches, Verilog provides flexibility in designing complex digital systems, making it suitable for simulation and verification before hardware implementation.

5.1 Structural Specification of Logic Circuits

- Verilog includes a set of *gate level primitives* that correspond to commonly used logic gates.
- A gate is represented by indicating its functional name, output, and inputs. For example, two-input AND gate, with inputs x_1 and x_2 and output y , is denoted as:

and (y, x_1, x_2);

- A four-input OR gate is specified as **:or** (y, x_1, x_2, x_3, x_4);
- The keywords **nand** and **nor** are used to define the NAND and NOR gates in the same way.
- The NOT gate given by **not** (y, x); implements $y = \sim x$.
- The gate level primitives can be used to specify larger circuits.
- A logic circuit is specified in the form of a *module* that contains the statements that define the circuit, and the module ends with the **endmodule** statement
- The module may have inputs and outputs, which are referred to as its *ports*. The name port is a commonly used term that refers to an input or output connection to an electronic circuit.

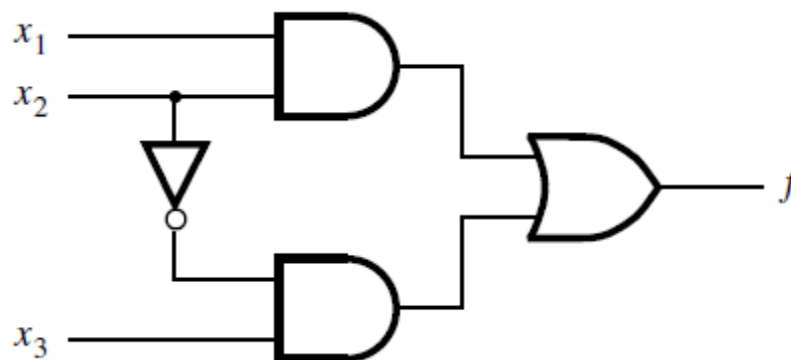


Fig.5.1: A Simple Logic Function

- The circuit in Fig.5.1 can be represented by the following Verilog code:

```

module example1(x1,x2,x3,f);
    input x1,x2,x3;
    output f;
    and (g,x1,x2);
    not(k,x2);
    and (h,k,x3);
    or(f,g,h);
endmodule

```

5.2 Behavioral Specification of Logic Circuits

- Using gate level primitives can be tedious when large circuits have to be designed. An alternative is to use more abstract expressions and programming constructs to describe the behavior of a digital circuit.
- Behavioral specification of a logic circuit defines only its behavior.
- One possibility is to define the circuit using logic expressions.

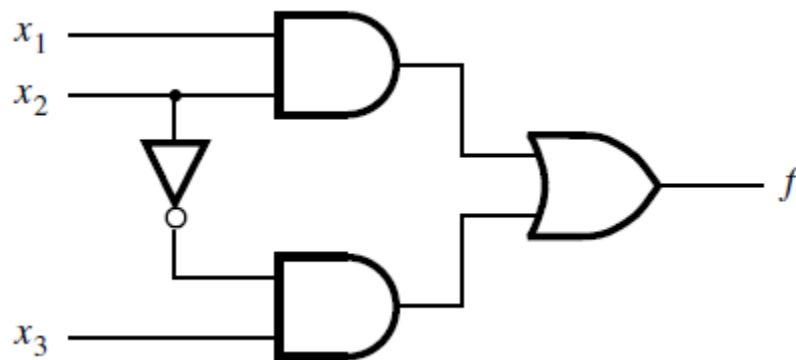


Fig.5.2: A Simple Logic Function

- The circuit in Fig.5.2 can be defined with the expression $f = x_1x_2 + ((\sim x_2)x_3)$. The AND, OR operations and NOT are indicated by the "&", "|", and "~" signs, respectively, and can be expressed in verilog as below.

```

module example1(x1,x2,x3,f);
    input x1,x2,x3;
    output f;
    assign f= (x1&x2)|((~x2)&x3));

```

endmodule

- The assign keyword provides a continuous assignment for the signal f . The word continuous stems from the use of Verilog for simulation; whenever any signal on the right-hand side changes its state, the value of f will be re-evaluated.
- Using logic expressions makes it easier to write Verilog code. But even higher levels of abstraction can often be used to advantage.
- Consider again the circuit in Fig 5.2. This circuit is similar to the 2-to-1 multiplexer circuit discussed in section with x2 being the selection control input and x1 and x3 being the data inputs.
- The circuit can be described in words by saying $f = x1$ if $x2 = 1$ and $f = x3$ if $x2 = 0$.
- This behavior can be defined with the if-else statement if (x2 == 1) f = x1; else f = x3; as the following verilog code.

```
module example1(x1,x2,x3,f);  
  input x1,x2,x3;  
  output reg f;  
  always @(x1,x2,x3)  
  begin  
    if(x2==1)  
      f=x1;  
    else  
      f=x3;  
  end  
endmodule
```

- . The if-else statement is an example of a Verilog procedural statement.
- Procedural statements should be contained inside a construct called an always block,
- An important property of the always block is that the statements it contains are evaluated in the order given in the code. This is in contrast to the continuous assignment statements, which are evaluated concurrently and hence have no meaningful order.
- The part of the always block after the @ symbol, in parentheses, is called the sensitivity list.

- The statements inside an always block are executed by the simulator only when one or more of the signals in the sensitivity list changes value. In this way, the complexity of a simulation process is simplified, because it is not necessary to execute every statement in the code at all times.
- If a signal is assigned a value using procedural statements, then Verilog syntax requires that it be declared as a variable; this is accomplished by using the keyword reg in as described "f" as output reg.

5.3 Hierarchical Verilog Code

- Creation of an n-bit adder requires drawing a hierarchical schematic that contains n full-adders.
- The same approach can also be followed by using Verilog, by first creating a Verilog module for a full-adder and then defining a higher-level module that uses n instances of the full-adder.

5.3.1 Four bit adder from one bit adder

- Consider the full adder circuit shown in Fig.5.3., which has the inputs C_{in} , x , and y , and produces the outputs s and C_{out} .

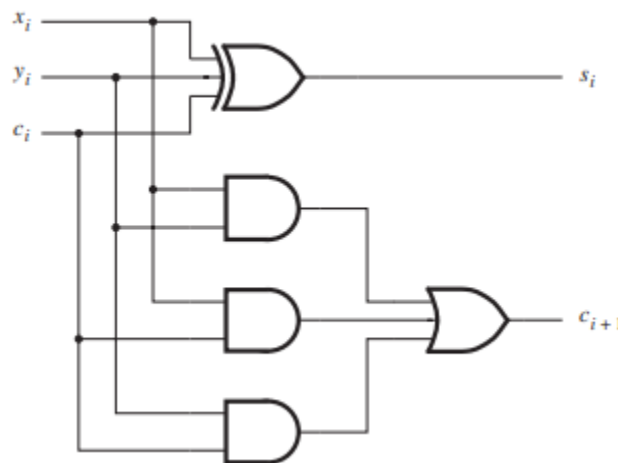


Fig.5.3:Full Adder Circuit

- One way of specifying this circuit in Verilog is to use the gate level primitives as shown below.

```
Module fulladd (Cin, x, y, s, Cout);  
  Input Cin, x, y;  
  output s, Cout;
```

```
xor (s, x, y, Cin);  
and (z1, x, y);  
and (z2, x, Cin);  
and (z3, y, Cin);  
or (Cout, z1, z2, z3);  
endmodule
```

- Each of the three AND gates in the circuit is defined by a separate statement. Verilog allows combining such statements into a single statement ,the recoded program is as shown below.

```
Module fulladd (Cin, x, y, s, Cout);  
input Cin, x, y;  
output s, Cout;  
xor (s, x, y, Cin);  
and (z1, x, y,  
    (z2, x, Cin),  
    (z3, y, Cin);  
or (Cout, z1, z2, z3);  
endmodule
```

- In this case, commas are used to separate the definition of each AND gate.
- Another possibility is to use functional expressions as shown in the following code. The XOR operation is denoted by the \wedge sign.

```
Module fulladd (Cin, x, y, s, Cout);  
Input Cin, x, y;  
output s, Cout;  
assign s=x  $\wedge$  y  $\wedge$  Cin;  
assign Cout = (x & y) | (x & Cin) | (y & Cin);  
endmodule
```

- Again, it is possible to combine the two continuous assignment statements into a single statement as in the following code.

```
Module fulladd (Cin, x, y, s, Cout);  
Input Cin, x, y;
```

```
output s, Cout;
assign s=x ^ y ^Cin,
Cout = (x & y) | (x &Cin) | (y &Cin);
endmodule
```

- We can now create a separate Verilog module for the ripple-carry adder, which instantiates Verilog code for the full-adder using continuous assignment.
- The fulladd module is a sub circuit. Any one of the above four modules can be considered.

```
module adder4 (carryin, x3, x2, x1, x0, y3, y2, y1, y0, s3, s2, s1, s0, carryout);
    inputcarryin, x3, x2, x1, x0, y3, y2, y1, y0;
    output s3, s2, s1, s0, carryout;
    fulladd stage0 (carryin, x0, y0, s0, c1);
    fulladd stage1 (c1, x1, y1, s1, c2);
    fulladd stage2 (c2, x2, y2, s2, c3);
    fulladd stage3 (c3, x3, y3, s3, carryout);
endmodule
```

- The module comprises the code for a four-bit ripple-carry adder, named adder4. One of the four-bit numbers to be added is represented by the four signals x3, x2, x1, x0, and the other number is represented by y3, y2, y1, y0.
- The sum is represented by s3,s2,s1,s0. The circuit incorporates a carryin, into the least-significant bit position and a carry output, from the most-significant bit position.
- Four instantiation statements are used. Each statement begins with the name of the module, fulladd, that is being instantiated. Next comes an instance name, which can be any legal Verilog name.
- The instance names must be unique. The least-significant stage in the adder is named stage0 and the most-significant stage is stage3.
- The signal names in the adder4 module that are to be connected to each input and output port on the fulladd module are then listed.
- These signals are listed in the same order as in the fulladd module, namely the order Cin, x, y, s, Cout.
- The signal names associated with each instance of the fulladd module implicitly specify

how the full-adders are connected together.

5.4 Verilog Syntax for Combinational Circuits

- Verilog provides constructs to define the behavior of a circuit such as conditional statement, If-else statement, case statement, for loop.

5.4.1 Conditional Statement

- In a logic circuit it is often necessary to choose between several possible signals or values based on the state of some condition.
- A typical example is a multiplexer circuit in which the output is equal to the data input signal chosen by the valuation of the select inputs.
- For simple implementation of such choices Verilog provides a conditional operator (?:) which assigns one of two values depending on a conditional expression.
- **syntax:** conditional_expression ? true_expression : false_expression
- If the conditional expression evaluates to 1 (true), then the value of true expression is chosen; otherwise, the value of false expression is chosen.
- For example, the statement `y =(s==1)? w1:w0;`
- In the above statement if "s==1" then "w1" will be assigned to y else "w0" will be assigned.
- Consider the following verilog code for a 2x1 multiplexer using conditional statement

```
module mux2to1 (w0, w1, s, f);  
    input w0, w1, s;  
    output f;  
    assign f = s ? w1 : w0;  
endmodule
```

- The same code can be written in always block without using assign statement as below

```
module mux2to1 (w0, w1, s, f);  
    input w0, w1, s;  
    output f;  
    reg f;  
    always @(w0 or w1 or s)  
    begin  
        f = s ?w1 : w0;  
    end
```

```
end  
endmodule
```

5.4.2 If-else Statement

- If-else statement helps to choose between several possible signals or values based on the state of some condition.
- if-else statement can be used only inside a always block
- **Syntax:** if (conditional_expression)

```
statement;  
else  
statement;
```

- The below verilog program describes 2x1 multiplexer using if-else statement

```
module mux2to1 (w0, w1, s, f);  
input w0, w1, s;  
output f;  
reg f;  
always @(w0 or w1 or s)  
begin  
if (s == 0)  
f = w0;  
else  
f = w1;  
end  
endmodule
```

5.4.3 Case Statement:

- When there are many possible alternatives, the code based on "if-else" statement may become awkward to read.
- In such case case statement becomes helpful with simple syntax.
- **Syntax:** case (expression)
alternative1: statement;
alternative2: statement;
...

alternativej: statement;

[default: statement;]

endcase

- The controlling expression and each alternative are compared bit by bit.
- When there is one or more matching alternative, the statement(s) associated with the first match (only) is executed.
- When the specified alternatives do not cover all possible valuations of the controlling expression, the optional default clause should be included.
- The below verilog code describes 2x1 multiplexer using case statement

```
module mux2to1 (W, S, f);  
    input [0:1] W;  
    input S;  
    output f;  
    reg f;  
    always @(W or S)  
    begin  
        case (S)  
            0: f = W[0];  
            1: f = W[1];  
        endcase  
    end  
endmodule
```

- The below verilog code describes 4x1 multiplexer using case statement

```
module mux4to1 (W, S, f);  
    input [0:3] W;  
    input [1:0] S;  
    output f;  
    reg f;  
    always @(W or S)  
    begin  
        case (S)
```

```
0: f = W[0];
1: f = W[1];
2: f = W[2];
3: f = W[3];
endcase
end
endmodule
```

5.4.4 For loop

- If the structure of a desired circuit exhibits a certain regularity, it may be possible to define the circuit using a for loop
- **syntax :** for (initial_index; terminal_index; increment)
statement;
- A loop control variable, which has to be of type integer, is set to the value given as the initial index.
- . After each iteration, the control variable is changed as defined in the increment.
- The iterations end after the control variable has reached the terminal index.
- During each iteration it specifies a different subcircuit.
- The following verilog module describes 2x4 decoder using for loop

```
module dec2to4 (W, Y, En);
input [1:0] W;
input En;
output [0:3] Y;
reg [0:3] Y;
integer k;
always @(W or En)
begin
for (k = 0; k <= 3; k = k+1)
begin
if ((W == k) && (En == 1))
Y[k] = 1;
else
```

```
Y[k] = 0;  
end  
end  
endmodule
```

5.5 Using Storage Elements with CAD – Tools

- This refers to the study of how circuits with storage elements can be designed using either schematic capture or Verilog code.

5.5.1 Including Storage Elements in Schematics

- One way to create a circuit is to draw a schematic that builds latches and flip-flops from logic gates.
- Because these storage elements are used in many applications, most CAD systems provide them as prebuilt modules.
- Fig 5.4 shows a schematic created with a schematic capture tool, which includes three types of flip-flops that are imported from a library provided as part of the CAD system.
- The top element is a gated D latch, the middle element is a positive-edge-triggered D flip-flop, and the bottom one is a positive-edge-triggered T flip-flop.
- The D and T flip-flops have asynchronous, active-low clear and preset inputs.
- If these inputs are not connected in a schematic, then the CAD tool makes them inactive by assigning the default value of 1 to them.

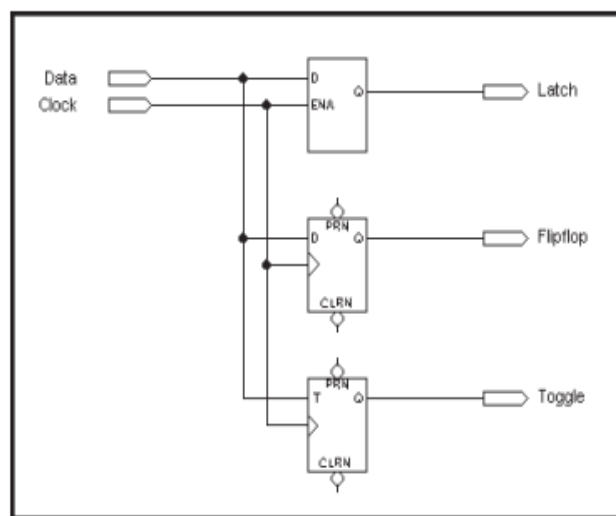


Fig.5.4: Three types of storage elements in a schematic

5.5.2 Using Verilog Constructs for Storage Elements

- A simple way of specifying a storage element is by using the if-else statement to describe the desired behavior responding to changes in the levels of data and clock inputs.
- Consider the always block

```
always @(Control or B)
    if (Control)
        A = B;
```

where A is a variable of reg type.

- This code specifies that the value of A should be made equal to the value of B when Control = 1. But the statement does not indicate an action that should occur when Control = 0.
- In the absence of an assigned value, the Verilog compiler assumes that the value of A caused by the if statement must be maintained until the next time this if statement is evaluated.
- This notion of implied memory is realized by instantiating a latch in the circuit.

```
module D_latch (D, Clk, Q);
    input D, Clk;
    output Q;
    reg Q;
    always @(D or Clk)
        begin
            if (Clk)
                Q = D;
        end
endmodule
```

- The above code defines a module named D_latch, which has the inputs D and Clk and the output Q.
- The “if” clause defines that the Q output must take the value of D when Clk = 1.
- Since no else clause is given, a latch will be synthesized to maintain value of Q when Clk = 0.

- Therefore, the code describes a gated D latch.
- The sensitivity list includes Clk and D because both of these signals can cause a change in the value of the Q output.

```
module flipflop (D, Clock, Q);  
    input D, Clock;  
    output Q;  
    reg Q;  
    always @(posedge Clock)  
    begin  
        Q = D;  
    end  
endmodule
```

- The above code defines a module named flipflop, which is a positive-edge-triggered D flip-flop.
- The sensitivity list contains only the clock signal because it is the only signal that can cause a change in the Q output.
- The keyword “posedge” specifies that a change may occur only on the positive edge of Clock.
- At this time the output Q is set to the value of the input D. Since Q is of reg type it will maintain its value between the positive edges of the clock.

5.5.3 Blocking and Non-Blocking Assignments

5.5.3.1 Blocking Assignments

- The equal sign used for assignments as Q=D is known as blocking assignment.
- A Verilog compiler evaluates the statements in an always block in the order in which they are written.
- If a variable is given a value by a blocking assignment statement, then this new value is used in evaluating all subsequent statements in the block.

```
module exampleblocking (D, Clock, Q1, Q2);  
    input D, Clock;  
    output Q1, Q2;
```

```
reg Q1, Q2;  
always @(posedge Clock)  
begin  
    Q1 = D;  
    Q2 = Q1;  
end  
endmodule
```

- In the above Verilog code always, block is sensitive to the positive clock edge, both Q1 and Q2 will be implemented as the outputs of D flip-flops.
- However, because blocking assignments are involved, these two flip-flops will not be connected in cascade, as the reader might expect.
- The first statement $Q1 = D$; sets Q1 to the value of D.
- This new value is used in evaluating the subsequent statement $Q2 = Q1$; which results in $Q2 = Q1 = D$.
- The synthesized circuit has two parallel flip-flops, as illustrated in Fig 5.5
- A synthesis tool will likely delete one of these redundant flip-flops as an optimization step.

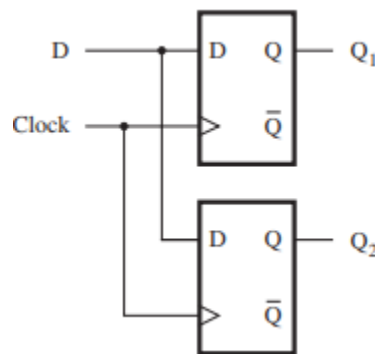


Fig.5.5:Schematic of Verilog module “exampleblocking”

5.5.3.2 Non-Blocking Assignments

- Statements denoted with “ \leq ” for assigning are known as Non –Blocking assignments
- All non-blocking assignment statements in an always block are evaluated using the values that the variables have when the always block is entered.
- Thus, a given variable has the same value for all statements in the block.
- The meaning of non-blocking is that the result of each assignment is not seen until the

end of the always block

```

module examplenonblocking (D, Clock, Q1, Q2);
input D, Clock;
output Q1, Q2;
reg Q1, Q2;
always @(posedge Clock)
begin
Q1 <= D;
Q2 <= Q1;
end
endmodule

```

- The Verilog code for example nonblocking uses non-blocking assignments.
- The two statements $Q1 <= D$; $Q2 <= Q1$;
- The variables $Q1$ and $Q2$ have some value at the start of evaluating the always block, and then they change to a new value concurrently at the end of the always block.
- This code generates a cascaded connection between flip-flops, which implements the shift register depicted in Fig 5.6.

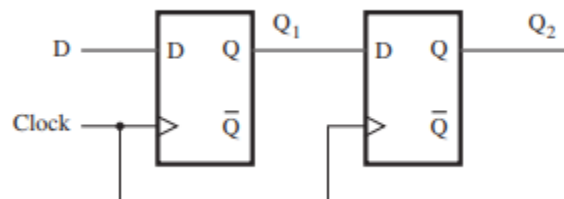


Fig.5.5:Schematic of Verilog module “examplenonblocking” (cascaded connection of flipflops)

5.5.4 Flip-Flops with Clear Capability

- By using a particular sensitivity list and a specific style of if-else statement, it is possible to include clear (or preset) signals on flip-flops.

5.5.4.1 Asynchronous Clear

```

module flipflop (D, Clock, Resetn, Q);
input D, Clock, Resetn;

```

```
output Q;  
reg Q;  
always @(negedge Resetn or posedge Clock)  
begin  
    if (!Resetn)  
        Q <= 0;  
    else Q <= D;  
end  
endmodule
```

- The above Verilog module “flipflop” gives a module that defines a D flip-flop with an asynchronous active-low reset (clear) input.
- When Resetn, the reset input, is equal to 0, the flip-flop’s Q output is set to 0.
- Note that the sensitivity list specifies the negative edge of Resetn as an event trigger along with the positive edge of the clock.
- We cannot omit the keyword negedge because the sensitivity list cannot have both edge-triggered and level-sensitive signals.

5.5.4.2 Synchronous Clear

```
Module flipflopsynchronous (D, Clock, Resetn, Q);  
input D, Clock, Resetn;  
output Q;  
reg Q;  
always @(posedge Clock)  
begin  
    if (!Resetn)  
        Q <= 0;  
    else  
        Q <= D;  
    end  
endmodule
```

- The above Verilog module “flipflopsynchronous” shows how a D flip-flop with a synchronous reset input can be described.
- In this case the reset signal is acted upon only when a positive clock edge arrives.

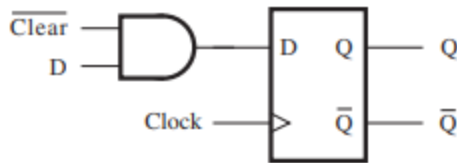


Fig.5.6: Schematic of Verilog module “flipflopsynchronous”

5.5.5. Using Verilog constructs for registers and counters

- Rather than instantiating predefined subcircuits for registers, shift registers, counters, and the like, the circuits can be described in Verilog code.
- One way to describe an n-bit register is to write hierarchical code that includes n instances of the D flip-flop subcircuit.
- A simpler approach is to define the D input and Q output as multibit signals.

5.5.5.1 A Four-Bit Shift Register

- Assume that we wish to write Verilog code that represents the four-bit parallel-access shift register in Fig5.7

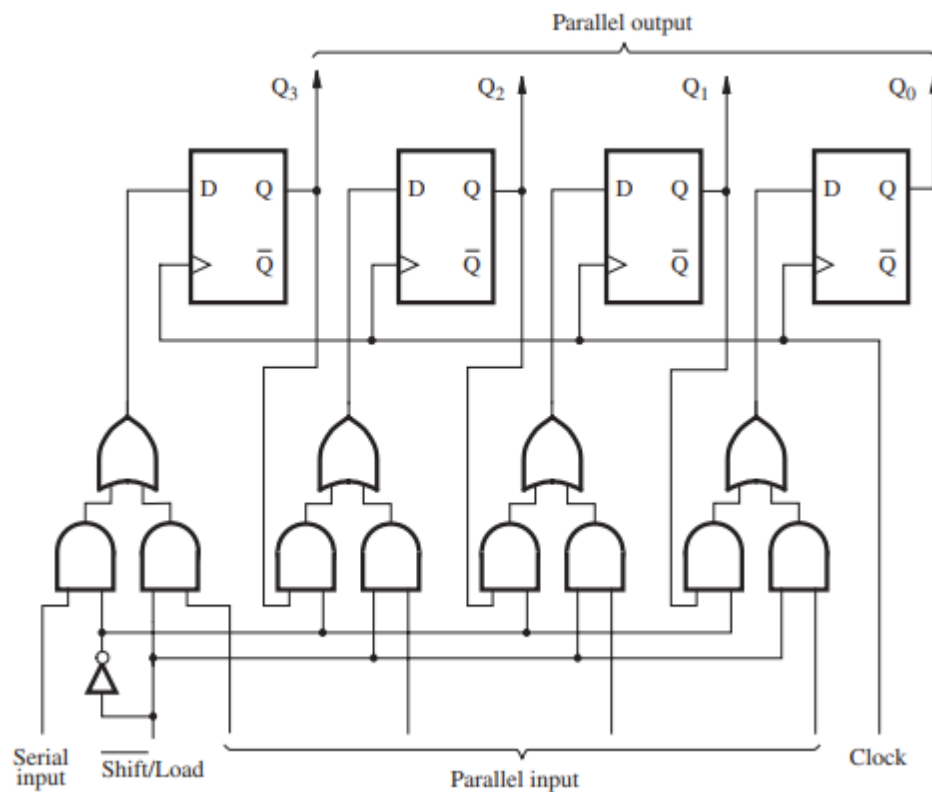


Fig.5.7: Parallel access shift register

- One approach is to write hierarchical code that uses four subcircuits.

- Each subcircuit consists of a D flip-flop with a 2-to-1 multiplexer connected to the D input.

```
module muxdff (D0, D1, Sel, Clock, Q);  
    input D0, D1, Sel, Clock;  
    output Q;  
    reg Q;  
    always @(posedge Clock)  
    begin  
        if (!Sel)  
            Q <= D0;  
        else Q <= D1;  
    end  
endmodule
```

- The above verilog code (Code for a D flip-flop with a 2-to-1 multiplexer on the D input.) defines the module named muxdff which represents the subcircuit.
- The two data inputs are named D0 and D1, and they are selected using the Sel input. The if-else statement specifies that on the positive clock edge if Sel = 0, then Q is assigned the value of D0; otherwise, Q is assigned the value of D1.

```
module shift4 (R, L, w, Clock, Q);  
    input [3:0] R;  
    input L, w, Clock;  
    output [3:0] Q;  
    wire [3:0] Q;  
    muxdff Stage3 (w, R[3], L, Clock, Q[3]);  
    muxdff Stage2 (Q[3], R[2], L, Clock, Q[2]);  
    muxdff Stage1 (Q[2], R[1], L, Clock, Q[1]);  
    muxdff Stage0 (Q[1], R[0], L, Clock, Q[0]);  
endmodule
```

- The above code defines the four-bit shift register in hierarchical way
- The module Stage3 instantiates the leftmost flip-flop, which has the output Q3, and the module Stage0 instantiates the right-most flip-flop, Q0.

- When $L = 1$, the register is loaded in parallel from the R input; and when $L = 0$, shifting takes place in the left to right direction.
- Serial data is shifted into the most-significant bit, Q3, from the w input.

Alternative Code for a Four-Bit Shift Register

- A different style of code for the four-bit shift register is, instead of using sub circuits, the shift register is defined using the following verilog code.

```
module shift4 (R, L, w, Clock, Q);  
    input [3:0] R;  
    input L, w, Clock;  
    output [3:0] Q;  
    reg [3:0] Q;  
    always @(posedge Clock)  
    begin  
        if (L) Q <= R;  
        else  
            begin  
                Q[0] <= Q[1];  
                Q[1] <= Q[2];  
                Q[2] <= Q[3];  
                Q[3] <= w;  
            end  
        end  
    endmodule
```

- All actions take place at the positive edge of the clock.
- If $L = 1$, the register is loaded in parallel with the four bits of input R.
- If $L = 0$, the contents of the register are shifted to the right and the value of the input w is loaded into the most-significant bit Q3.

5.5.5.2 Counter

Up-Counter

```
module upcount (Resetn, Clock, E, Q);  
    input Resetn, Clock, E;
```

```
output [3:0] Q;  
reg [3:0] Q;  
always @(negedge Resetn or posedge Clock)  
begin  
    if (!Resetn)  
        Q <= 0;  
    else if (E)  
        Q <= Q + 1;  
    end  
endmodule
```

- The above verilog code represents a four-bit up-counter with a reset input- Resetn, and an enable input, E.
- The outputs of the flip-flops in the counter are represented by the vector named Q.
- The if statement specifies an asynchronous reset of the counter if Resetn = 0.
- The else if clause specifies that if E = 1 the count is incremented on the positive clock edge.

Up-Counter with Parallel Load

```
module upcount (R, Resetn, Clock, E, L, Q);  
input [3:0] R;  
input Resetn, Clock, E, L;  
output [3:0] Q;  
reg [3:0] Q;  
always @(negedge Resetn or posedge Clock)  
begin  
    if (!Resetn)  
        Q <= 0;  
    else if (L)  
        Q <= R;  
    else if (E)  
        Q <= Q + 1;  
    end  
end
```

```
endmodule
```

- The above verilog code defines an up-counter that has a parallel-load input in addition to a reset input.
- The parallel data is provided as the input vector R.
- The first if statement provides the same asynchronous reset
- The else if clause specifies that if $L = 1$ the flip-flops in the counter are loaded in parallel from the R inputs on the positive clock edge.
- If $L = 0$, the count is incremented, under control of the enable input E.

Down-Counter with Parallel Load

```
module downcount (R, Clock, E, L, Q);  
    parameter n = 8;  
    input [n-1:0] R;  
    input Clock, L, E;  
    output [n-1:0] Q;  
    reg Q;  
    always @(posedge Clock)  
    begin  
        if (L)  
            Q <= R;  
        else if (E)  
            Q <= Q -1;  
        end  
    endmodule
```

- The above code named downcount describes down-counter.
- A down-counter is normally used by loading it with some starting count and then decrementing its contents.
- The starting count is represented in the code by the vector R.
- On the positive clock edge, if $L = 1$ the counter is loaded with the input R, and if $L = 0$ the count is decremented.
- The counter also includes an enable input, E. Setting $E = 0$ prevents the contents of the flip-flops from changing when an active clock edge occurs.

Up/Down Counter

```
module updowncount (R, Clock, L, E, up_down, Q);
    parameter n = 8;
    input [n-1:0] R;
    input Clock, L, E, up_down;
    output [n-1:0] Q;
    reg Q;
    integer direction;
    always @(posedge Clock)
    begin
        if (up_down)
            direction = 1;
        else
            direction = -1;
        if (L)
            Q <= R;
        else if (E)
            Q <= Q + direction;
        end
    endmodule
```

- The above Verilog code defines an up/down counter.
- It includes a control signal up_down that governs the direction of counting.
- It also includes an integer variable named direction, which is equal to 1 for up-count and equal to -1 for down-count

5.6 Verilog Code for Sequential Circuits**5.6.1 Register**

```
module regn (D, Clock, Resetn, Q);
    parameter n=4;
    input [n-1:0] D;
    input Clock, Resetn;
    output [n-1:0] Q;
```



```
reg Q;  
always @(negedge Resetn or posedge Clock)  
begin  
if (!Resetn)  
Q <= 0;  
else  
Q <= D;  
end  
endmodule
```

- The above verilog code defines an n-bit register.
- Since registers of different sizes are often needed in logic circuits, it is advantageous to define a register module for which the number of flip-flops can be easily changed.
- The parameter n specifies the number of flip-flops in the register. By changing this parameter, the code can represent a register of any size.

5.6.2 Asynchronous counter

```
module Asynchronouscounter (Resetn, Clock, Q);  
input Resetn, Clock;  
output [3:0] Q;  
reg [3:0] Q;  
always @(negedge Resetn or posedge Clock)  
begin  
if (!Resetn)  
Q <= 0;  
else  
Q <= Q + 1;  
end  
endmodule
```

- The above verilog code represents a four-bit Asynchronous up-counter with a reset input- Resetn.
- The outputs of the flip-flops in the counter are represented by the vector named Q.
- The if statement specifies an asynchronous reset of the counter if Resetn = 0.

- The else clause specifies that the count is incremented on the positive clock edge.

5.6.3 Synchronous counter

```
module synchronouscounter (Resetn, Clock, Q);  
    input Resetn, Clock;  
    output [3:0] Q;  
    reg [3:0] Q;  
    always @( posedge Clock)  
    begin  
        if (!Resetn)  
            Q <= 0;  
        else  
            Q <= Q + 1;  
        end  
    endmodule
```

- The above verilog code represents a four-bit synchronous up-counter with a reset input-Resetn.
- The outputs of the flip-flops in the counter are represented by the vector named Q.
- The statements inside the always block executes only at positive edge making it synchronous
- The if clause specifies when Resetn is 0 on positive edge of clock then output equals to 0
- The else clause specifies that the count is incremented on the positive clock edge.

5.7 Verilog for combinational circuits

5.7.1 8x3 Encoder

```
module encoder83(o,a,en);  
    input [7:0] a;  
    input en;  
    output reg [2:0] o;  
    always @(a,en)  
    begin  
        if(en==1'b1)
```

```
begin
case(a)
8'b00000001: o=3'b000;
8'b00000010: o=3'b001;
8'b00000100: o=3'b010;
8'b00001000: o=3'b011;
8'b00010000: o=3'b100;
8'b00100000: o=3'b101;
8'b01000000: o=3'b110;
8'b10000000: o=3'b111;
default: o=3'bx;
endcase
end
else
o=3'bz;
end
endmodule
```

- The above verilog code represents an 8x3 encoder with an enable signal
- It has 7 bit input "a", and enable input "en" and 3 bit output "o".
- The if clause when "en=1" then through case statement according to applied input "a" the corresponding output "o" will be obtained
- The else clause represents when enable is disabled then output is high impedance.

5.7.2 3x8 Decoder

```
module decoder3to8(in,out,en);
input [2:0] in;
input en;
output reg [7:0] out;
always @(in,en)
begin
if(en)
out =8'b0;
```

```
else
begin
case(in)
3'b000: y=8'b00000001;
3'b001: y=8'b00000010;
3'b010: y=8'b00000100;
3'b011: y=8'b00001000;
3'b100: y=8'b00010000;
3'b101: y=8'b00100000;
3'b110: y=8'b01000000;
3'b111: y=8'b10000000;
default: y =8'bx;
endcase
end
end
endmodule
```

- The above Verilog code represents an 3x8 Decoder with an enable signal
- It has 3 bit input "in", and enable input "en" and 8 bit output "out".
- The if clause represents when "en=1" output is zero .
- In else clause through "case statement" according to applied input "in" the corresponding output "out" will be obtained.

5.7.3 8x1 Multiplexer

```
module Multiplexer(i,sel,out);
input [7:0] i;
input [2:0] sel;
output reg out;
always@(sel,i)
begin
case(sel)
3'b000:out=i[0];
3'b001:out=i[1];
```

```
3'b010:out=i[2];
3'b011:out=i[3];
3'b100:out=i[4];
3'b101:out=i[5];
3'b110:out=i[6];
3'b111:out=i[7];
endcase
end
endmodule
```

- The above verilog code represents an 8x1 Multiplexer
- It has 8 bit data input "i", 3 bit select input "sel" and 1 bit output "out".
- Through the case statement according to the select input applied one of the input bits get selected and will be assigned to output.

5.7.4 1x8 Demultiplexer

```
module DeMultiplexer(i,sel,out);
input i;
input [2:0] sel;
output reg [7:0] out;
always@(sel,i)
begin
case(sel)
3'b000:out[0]=i;
3'b001:out[1]=i;
3'b010:out[2]=i;
3'b011:out[3]=i;
3'b100:out[4]=i;
3'b101:out[5]=i;
3'b110:out[6]=i;
3'b111:out[7]=i;
endcase
end
```

endmodule

- The above verilog code represents an 1x8 DeMultiplexer
- It has 1 bit data input "i", 3 bit select input "sel" and 8 bit output "out".
- Through the case statement according to the select input applied one of the output bits get selected and will be assigned to input.

Additional Resources:

1. NPTEL course on “System Design Through Verilog” by Prof. Shaik Rafi Ahamed, IIT Guwahati https://onlinecourses.nptel.ac.in/noc21_ee97/preview

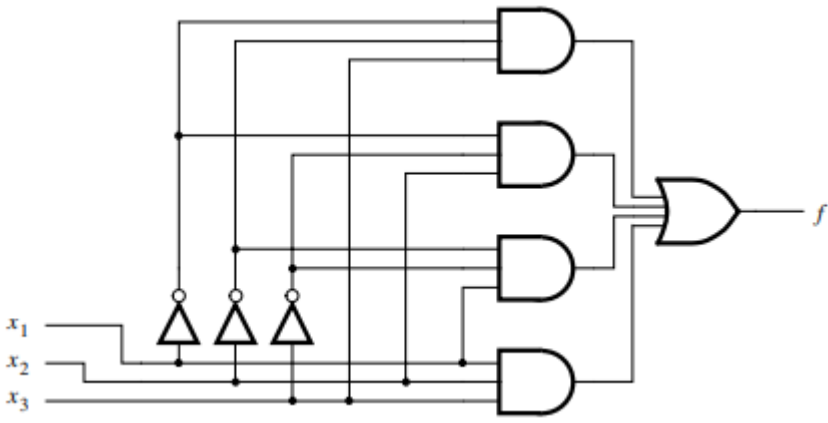
Questions

5.1 Structural Specification of Logic circuits

Objective Questions

- Identify the size, inputs and output of the following gate primitive.
and g1 (z, a,b,c,p);
a) 4,a,b,c,p,z
b) 5,z,a,b,c,p
c) 4,z,a,b,c,p
d) 5,a,b,c,p,z
- What are the gate primitives required to implement half adder
a) xor,and
b) xnor,nand
c) nor,nand
d) xnor,xor
- What is the error in the following Verilog code _____
not g1(z,a,p,d);

Subjective Questions

S.No	Questions	BL
1	<p>Write Verilog code to implement the circuit in below figure using the gate level primitives.</p> 	L3
2.	<p>Write Verilog code to implement the circuit in below figure using the gate level primitives.</p>	L3

3.	Write Verilog code to implement the function $f(x_1, x_2, x_3) = m(1, 2, 3, 4, 5, 6)$ using the gate level primitives. Ensure that the resulting circuit is as simple as possible.	L3
4.	Write Verilog code to implement the function $f(x_1, x_2, x_3) = m(0, 1, 3, 4, 5, 6)$ using the gate level primitives.	L3
5	Write Verilog code to describe the following functions using gate level primitives $F1 = (x1 + x3) \cdot (x1 + x2 + x4) \cdot (x2 + x3 + x4)$	L3
6.	Write Verilog code to describe the following functions using gate level primitives $f1 = x1x3 + x2x3 + x3x4 + x1x2 + x1x4$	L3
7	Explain the syntax of how to use gate level primitives in a Verilog module	L3

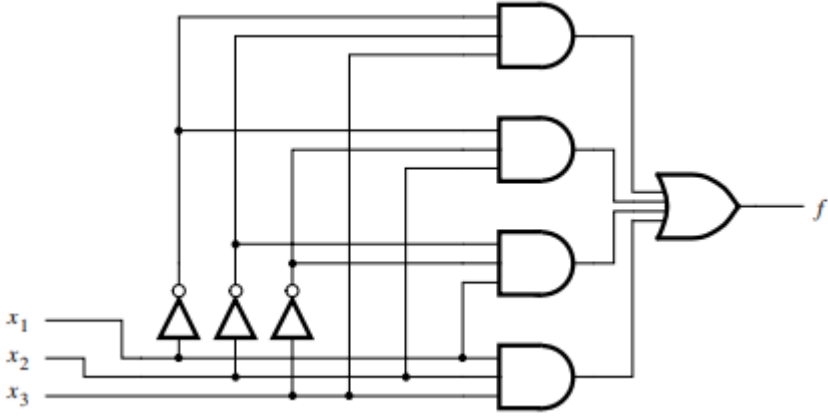
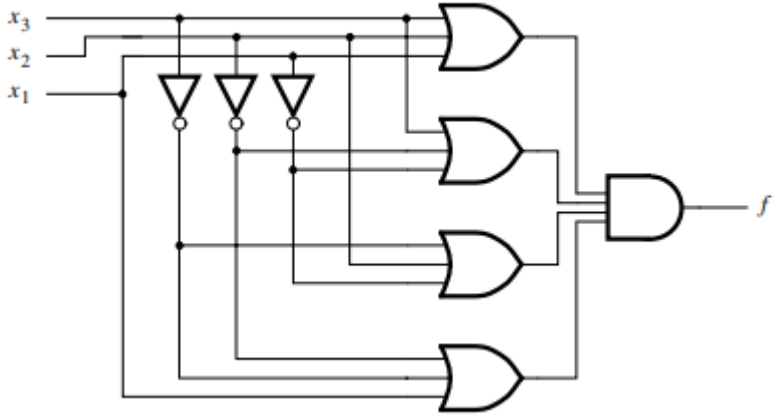
5.2 Behavioral Specification of logic circuits

Objective Questions

- What is the continuous assignment statement required to implement borrow of subtractor circuit
 - assign borrow = ~a &b;
 - assign borrow = !a&b;
 - assign borrow = a^b;
 - assign borrow = xor(a,b);
- How many gates and what gates are required for the following assignment statement.
 assign y= (a1*a2)+(a3+a4)

- a) And =1, or =2
 b) And =2, or =1
 c) And=2,or=2
 d) And=1,or=1
3. If there is any error present in the following Verilog statement; specify it else answer as no error : _____
 assign y =xor(a,b);

Subjective Questions

S.No	Questions	BL
1	<p>Write Verilog code to implement the circuit in below figure using the continuous assignment</p> 	L3
2.	<p>Write Verilog code to implement the circuit in below figure using the continuous assignment</p> 	L3

3.	Write Verilog code to implement the function $f(x_1, x_2, x_3) = m(1, 2, 3, 4, 5, 6)$ using the continuous assignment. Ensure that the resulting circuit is as simple as possible.	L3
4.	Write Verilog code to implement the function $f(x_1, x_2, x_3) = m(0, 1, 3, 4, 5, 6)$ using the continuous assignment.	L3
5.	Write Verilog code to describe the following function using continuous assignment. $F1 = (x_1 + x_3) \cdot (x_1 + x_2 + x_4) \cdot (x_2 + x_3 + x_4)$	L3
6.	Write Verilog code to describe the following function using continuous assignment. $F1 = x_1x_3 + x_2x_3 + x_3x_4 + x_1x_2 + x_1x_4$	L3
7.	Write Verilog code to describe 4x1 multiplexer using if-else statements.	L3
8.	Write Verilog code to describe 8x3 encoder using if-else statements	L3
9.	Write Verilog code to describe 3X8 decoder using if-else statements	L3
10.	Write Verilog code to describe 1x8 de-multiplexer using if-else statements.	L3
11.	Explain the need of if-else statement and its syntax	L3
12.	Briefly discuss about continuous assignment statement along with its syntax	L3

5.3 Hierarchical Verilog Code

Objective Questions

- How many instances of half adder are required to design full adder using half adder.
 - One
 - Two
 - Three
 - Four
- Each instance in hierarchical code will _____
 - Generate a block of hardware
 - Update variables for one instance
 - Update loop variable

- d) Update based on the condition

Subjective Questions

S.No	Questions	BL
1	Write Verilog code to implement the full adder from half adder using hierarchical fashion	L3
2	Write Verilog code to implement the ripple carry adder from full adder using hierarchical fashion	L3
3	Write Verilog code to implement the 8x1 mux from 4x1 mux using hierarchical fashion	L3
4	Briefly discuss about hierarchical Verilog code along with syntax	

5.4 Verilog Syntax for Combinational Circuits

5.4.1 Conditional Operator

Objective Questions

1. For the following Verilog code what will be output when s=0 and s=1 respectively.

Y=s? 1'b0:1'bx;

- a) x,0
 - b) 0,x
 - c) 0,0
 - d) X,x
2. What is the functionality represented by the following code

Y= a? b? 1:0:0

- a) AND gate
- b) OR gate
- c) Xor gate
- d) 2x1 Multiplexer

Subjective Questions

S.No	Questions	BL
1	Write Verilog code to implement the 2x1 multiplexer using conditional operator	L3

2	Write Verilog code to implement the 4x1 multiplexer using conditional operator	L3
3	Explain about conditional operator with syntax	L3

5.4.2 If-else statement

Objective Questions

1. What is the output of the following Verilog code when “in=1” and “in=0” respectively.

```

module ex(in,a,y);
input in, a;
output reg y;
always @(in, a)
begin
if(!in)
y =a;
else
y =in;
end
endmodule

```

- a) in,a
 - b) a,in
 - c) a,a
 - d) in,in
2. What is the functionality being implemented by the following Verilog code

```

module ex(a,β,c,y);
input a,β,χ;
output reg ψ;
αλωαψσ ≡(α,β,χ)
βεγιν
ιφ(α)
ψ =β;
ελσε

```

$\psi = \chi;$

$\varepsilon \vee \delta$

$\varepsilon \vee \delta \text{ modulo } \lambda \varepsilon$

$\alpha) \alpha \text{ xor } \beta$

$\beta) \alpha \text{ and } \beta$

$\chi) 2 \times 1 \text{ multiplexer}$

$\delta) \alpha \text{ or } \beta$

Subjective Questions

S.No	Questions	BL
1	Write Verilog code to implement the 2x1 multiplexer using if else statement	L3
2	Write Verilog code to implement the 4x1 multiplexer using if else statement	L3
3	Explain about if –else condition with syntax	L3
4	Write Verilog code to implement the 2x4 decoder using if else statement	L3
5	Write Verilog code to implement the 4x2 encoder using if else statement	L3

5.4.3 Case statement

Objective Questions

1. In the following Verilog code when will the default statement gets executed

```

module casestmt(a,b,y);
input a,b;
output reg y;
always @(a,b)
begin
case({a,b})
2'b00: y=0;
2'b01: y=1;
2'b11: y=x;
default :y=z;
endcase

```

end

endmodule

a) 00

b) 01

c) 10

d) 11

2. What is the error present in the following lines of verilog code-----

case(sel)

3'b000: y = a+β;

ζ= α &β;

3□β001:

Subjective Questions

S.No	Questions	BL
1	Write Verilog code to implement the 2x1 multiplexer using case statement	L3
2	Write Verilog code to implement the 4x1 multiplexer using case statement	L3
3	Explain about case statement with syntax	L3
4	Write Verilog code to implement the 3x8 decoder using case statement	L3
5	Write Verilog code to implement the 8x3 encoder using case statement	L3
6	Write Verilog code for ALU using case statement	L3

5.4.4 For Loop

Objective Questions

- Which of the following is correct syntax for “for loop”
 - for(initial condition; condition; updation)begin
statements
end
 - for(updation; condition; initialcondition)begin

- ```

statements
end
c) for(update; initialcondition; stepupdate)begin
statements
end
d) for(condition; condition; update)begin
statements
end

```
2. To generate input for 8x1 mux which of the following code is correct
    - a) for (sel =0; sel<7; sel++);
    - b) for (sel =7; sel>=0; sel--);
    - c) for (sel =7; sel>0; sel--);
    - d) for (sel =0; sel <=7; sel+2);

### Subjective Questions

| S.No | Questions                                                          | BL |
|------|--------------------------------------------------------------------|----|
| 1    | Write Verilog code to implement the 2x1 multiplexer using for loop | L3 |
| 2    | Write Verilog code to implement the 4x1 multiplexer using for loop | L3 |
| 3    | Explain about For loop with syntax                                 | L3 |
| 4    | Write Verilog code to implement the 3x8 decoder using for loop     | L3 |

## 5.5 Using Storage Elements with CAD – Tools

### 5.5.1 Including storage elements in schematics

#### Objective Questions

1. Storage elements used in many applications will be provided by most CAD systems as
  - a) prebuilt modules
  - b) logic gates
  - c) assignment statements
  - d) through nand gates

#### Subjective Questions

| S.No | Questions | BL |
|------|-----------|----|
|------|-----------|----|

|   |                                                                       |    |
|---|-----------------------------------------------------------------------|----|
| 1 | Write in detail about how storage elements are included in schematics | L3 |
|---|-----------------------------------------------------------------------|----|

### 5.5.2 Using Verilog constructs for storage elements

#### Objective Questions

- What is the simple way to specify a storage element for describing the desired behavior responding to changes in the levels of data and clock inputs.
  - If-else statement
  - For loop
  - Continuous assignment
  - Hierarchical representation
- construct is used to define a circuit that responds to changes in the signals that appear in the sensitivity list
  - always block
  - initial block
  - generate block
  - for loop

#### Subjective Questions

| S.No | Questions                                                                        | BL |
|------|----------------------------------------------------------------------------------|----|
| 1    | Write a Verilog code for gated D-Latch                                           | L3 |
| 2    | Write a Verilog code for D-Flip Flop                                             | L3 |
| 3    | Write a Verilog code for gated JK-Latch                                          | L3 |
| 4    | Write a Verilog code for JK-Flip Flop                                            | L3 |
| 5    | Write a Verilog code for gated SR-Latch                                          | L3 |
| 6    | Write a Verilog code for SR-Flip Flop                                            | L3 |
| 7    | Write a Verilog code for gated T-Latch                                           | L3 |
| 8    | Write a Verilog code for T-Flip Flop                                             | L3 |
| 9    | Explain in detail about the Verilog constructs used to describe storage elements | L3 |

### 5.5.3 Blocking and Non- Blocking Assignments



### 5.5.3.1 Blocking Assignments

#### Objective Questions

- Which of the following is blocking assignment
  - ==
  - =
  - <=
  - =<
- Consider  $Q1=5$ ,  $Q2=7$  then what will be the output after executing following code  
 $Q1 = Q2$ ;  
 $Q2 = Q1$ ;
  - 5,5
  - 5,7
  - 7,7
  - 7,5

#### Subjective Questions

| S.No | Questions                                                           | BL |
|------|---------------------------------------------------------------------|----|
| 1    | Explain the differences between blocking and nonblocking statements | L3 |
| 2    | Write the code for D-flipflop using blocking statements             | L3 |
| 3    | Write the code for D-Latch using blocking statements                | L3 |

### 5.5.3.2 Non-Blocking statements

#### Objective Questions

- Which of the following is Non-blocking assignment
  - ==
  - =
  - <=
  - =<
- Consider  $Q1=5$ ,  $Q2=7$  then what will be the output after executing following code  
 $Q1 <= Q2$ ;  
 $Q2 <= Q1$ ;
  - 5,5

- b) 5,7
- c) 7,7
- d) 7,5

### Subjective Questions

| S.No | Questions                                                             | BL |
|------|-----------------------------------------------------------------------|----|
| 1    | Explain the differences between blocking and nonblocking statements   | L3 |
| 2    | Write the code for D-flipflop using non-blocking statements           | L3 |
| 3    | Write the code for D-Latch using non-blocking statements              | L3 |
| 4    | Write a verilog code for shift register using non-blocking statements | L3 |

### 5.5.4 Flip-flops with clear capability

#### 5.5.4.1 Asynchronous Clear

#### Objective questions

- What does the following code represents  
always@(posedge clk, negedge reset)
  - a) Asynchronous reset
  - b) Synchronous reset
  - c) Asynchronous set
  - d) Synchronous clear

### Subjective Questions

| S.No | Questions                                                          | BL |
|------|--------------------------------------------------------------------|----|
| 1    | Explain the differences between Synchronous and Asynchronous clear | L3 |
| 2    | Write a verilog code for D-flip flop using asynchronous clear      | L3 |
| 3    | Write a verilog code for Shift register using asynchronous clear   | L3 |

#### 5.5.4.2 Synchronous Clear

#### Objective questions

- What does the following code represents  
always@(posedge Clk, negedge reset)
  - a) Asynchronous reset
  - b) Synchronous reset

- c) Asynchronous set
- d) Synchronous clear

### Subjective Questions

| S.No | Questions                                                          | BL |
|------|--------------------------------------------------------------------|----|
| 1    | Explain the differences between Synchronous and Asynchronous clear | L3 |
| 2    | Write a verilog code for D-flip flop using synchronous clear       | L3 |
| 3    | Write a verilog code for Shift register using synchronous clear    | L3 |

### 5.5.5 Using Verilog Constructs for registers and counters

#### 5.5.5.1 A Four bit shift register

### Objective Questions

- For 4 bit shift register what should be the value of “x” in the following code  

```
parameter n=x;
output [n-1:0] q;
```

  - a) 4
  - b) 3
  - c) 5
  - d) n
- When coding a four bit shift register in behavioral style the statements inside the always block should be
  - a) Blocking statements
  - b) Non-blocking statements
  - c) Combination of blocking and non-blocking statements
  - d) Continuous assignment statements

### Subjective Questions

| S.No | Questions                                                            | BL |
|------|----------------------------------------------------------------------|----|
| 1    | Write a verilog code for 4 bit shift register using behavioral style | L3 |
| 2    | Write a verilog code for 4 bit shift register using structural style | L3 |

### 5.5.5.2 Counters

### Objective Questions

- For active low reset up counter which of the following statements is correct

- a) When reset =0 then  $Q=1$   
 b) When reset =0 then  $Q= Q+1$   
 c) When reset =1 then  $Q = Q+1$   
 d) When reset =0 then  $Q=0$
2. For the following updown counter code when does the design works as up counter  
 if (updown)  
   direction =1;  
 else  
   direction = -1;  
 if(en)  
   q = q+direction  
 a) updown =1 and direction =1  
 b) updown =0 and direction =1  
 c) updown =1 and direction =-1  
 d) updown =0 and direction =-1
3. For the following updown counter code when does the design works as down counter  
 if (updown)  
   direction =1;  
 else  
   direction = -1;  
 if(en)  
   q = q+direction  
 a) updown =1 and direction =1  
 b) updown =0 and direction =1  
 c) updown =1 and direction =-1  
 d) updown =0 and direction =-1

### Subjective Questions

| S.No | Questions                             | BL |
|------|---------------------------------------|----|
| 1    | Write a verilog code for up counter   | L3 |
| 2    | Write a verilog code for down counter | L3 |

|   |                                                          |    |
|---|----------------------------------------------------------|----|
| 3 | Write a verilog code for down counter with parallel load | L3 |
| 4 | Write a verilog code for up counter with parallel load   | L3 |
| 5 | Write a verilog code for up down counter                 | L3 |

## 5.6 Verilog code for sequential circuits

### 5.6.1 Registers

#### Objective Questions

- Choose the correct output after executing following lines of code

Initially q1=1; q2=0; d=1;

q1 <=d;

q2<=q1;

- q1=0,q2=0
- q1=0,q2=1
- q1=1,q2=0
- q1=1,q2=1

#### Subjective Questions

| S.No | Questions                                                  | BL |
|------|------------------------------------------------------------|----|
| 1    | Write a verilog code for synchronous 4 bit shift register  | L3 |
| 2    | Write a verilog code for Asynchronous 4 bit shift register | L3 |

### 5.6.2 Asynchronous 4 bit counter

#### Objective Questions

- Which of the following lines represent asynchronous design
  - always @(posedge clk)
  - always @ (negedge clk)
  - always @(posedge clk or negedge reset)
  - always @(clk)

#### Subjective Questions

| S.No | Questions                                           | BL |
|------|-----------------------------------------------------|----|
| 1    | Write a verilog code for Asynchronous 4 bit counter | L3 |

|   |                                                                |    |
|---|----------------------------------------------------------------|----|
| 2 | Explain how an asynchronous circuit can be designed in verilog | L3 |
|---|----------------------------------------------------------------|----|

### 5.6.3 Synchronous 4 bit counter

#### Objective Questions

- Which of the following lines represent synchronous design
  - always @(posedge clk)
  - always @ (negedge clk or posedge reset)
  - always @(posedge clk or negedge reset)
  - always @(clk or reset)

#### Subjective Questions

| S.No | Questions                                                    | BL |
|------|--------------------------------------------------------------|----|
| 1    | Write a verilog code for synchronous 4 bit counter           | L3 |
| 2    | Explain how a synchronous circuit can be designed in verilog | L3 |

### 5.7 Verilog code for Combinational Circuits

#### 5.7.1 8x3 Encoder

#### Objective Questions

- For an 8x3 encoder with default statement what are the minimum number of different cases that need to be written to define the truth table?
  - 256
  - 257
  - 8
  - 9

#### Subjective Questions

| S.No | Questions                            | BL |
|------|--------------------------------------|----|
| 1    | Write a verilog code for 8*3 encoder | L3 |

#### 5.7.2 3x8 Decoder

#### Objective Questions

- For an 3x8 decoder with active low enable which of the following statements is correct
  - When en=1 and in=3'b001 then y[1]=1

- b) When  $en=0$  and  $in=3'b001$  then  $y[1] = 1$
- c) When  $en=1$  and  $in=3'b011$  then  $y[1] = 1$
- d) When  $en=0$  and  $in=3'b001$  then  $y[1] = 0$

**Subjective Questions**

| S.No | Questions                            | BL |
|------|--------------------------------------|----|
| 1    | Write a verilog code for 3*8 decoder | L3 |

**5.7.3 8\*1 Multiplexer****Objective Questions**

- To write a verilog code for an 8\*1 Multiplexer using assign statements minimum number of lines required to write functionality or logic
  - a) 1
  - b) 9
  - c) 3
  - d) 8

**Subjective Questions**

| S.No | Questions                                | BL |
|------|------------------------------------------|----|
| 1    | Write a verilog code for 8*1 Multiplexer | L3 |

**5.7.4 1\*8 De-Multiplexer****Objective Questions**

- For an 8\*1 De-Multiplexer which of the following statements is correct  
 initial begin  $s=3'b001$   
 $in = 1$ 
  - a)  $Y[1] = 1;$
  - b)  $Y[1] = 0;$
  - c)  $Y[2] = 1;$
  - d)  $Y[4] = 1;$

**Subjective Questions**

| S.No | Questions                                   | BL |
|------|---------------------------------------------|----|
| 1    | Write a verilog code for 1*8 De-Multiplexer | L3 |

