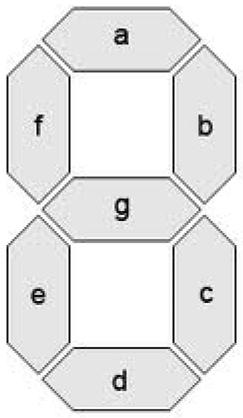
CS225- Lab 10
Design using HDL(Verilog)

The goal of this lab10 is to familiarize the students with describing computer architectural blocks in verilog Hardware Description Language (HDL). Further we will simulate various hardware building blocks with appropriate stimulus. In this section we describe various some more logic circuits.

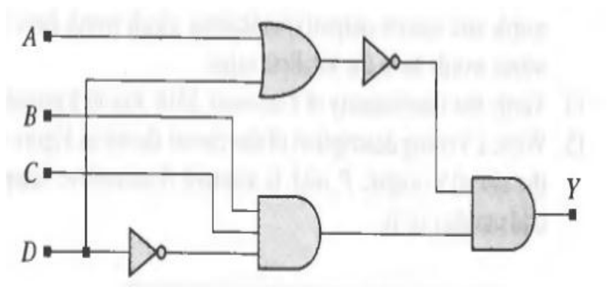
Combinational Logic

Task1:The seven-segment light-emitting diode (LED) display depicted in Figure is a useful circuit in many applications using prototyping boards. Module Seven_Seg_Display accepts 4-bit words representing binary coded decimal (BCD) digits and displays their decimal value. The display has active-low illumination outputs, and can be implemented with combinational logic. The description synthesizes into a combinational circuit. Several of the input codes are unused and should not occur under ordinary operation. One possibility is to assign don't-cares to those codes. However, this would display an output if such an input code occurred. Instead, the default assignment blanks the display for all unused codes. This prevents a bogus display condition. If the default assignment is omitted, an event of an input that is not decoded will be detected by the event control expression of the cyclic behavior, but will not cause Display to be an assigned value. Simulate the following and study the behavior.



```
module Seven_Seg_Display (Display, BCD);
output [6: 0] Display;
input [3: 0] BCD;
reg [6: 0] Display;
            // abc_detg
parameter BLANK =7'b111_1111;
parameter ZERO = 7'b000_0001;      // h01
parameter ONE =  7'b100_1111;      // h4f
parameter TWO =  7'b001_0010;      // h12
parameter THREE =7'b000_0110;      // h06
parameter FOUR = 7'b100_1100;  // h4c
parameter FIVE = 7'b010_0100;  // h24
parameter SIX =  7'b010_0000;      // h20
parameter SEVEN= 7'b000_1111;      // hOf
parameter EIGHT =7'b000_0000;      // hOO
parameter NINE = 7'b000_0100;      // h04
always @ (BCD)
case (BCD)
0: Display = ZERO;
1: Display = ONE;
2: Display = TWO;
3: Display = THREE;
4: Display = FOUR;
5: Display = FIVE;
6: Display = SIX;
7: Display = SEVEN;
8: Display = EIGHT;
9: Display = NINE;   default:Display = BLANK;
endcase
 endmodule
```

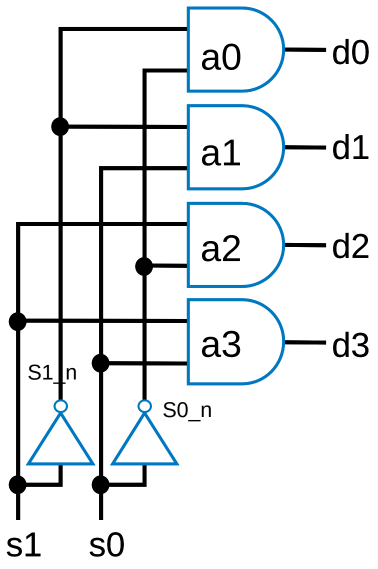
*Task2:**Using a single continuous assignment. develop and verify a behavioral model implementing a Boolean equation describing the logic of the circuit below. Use the following names for the testbench, the model, and its ports: tb_Combo_CA(), and Combo _ CA (Y, A, B, C, D), respectively. Note: The test bench will have no ports. Exhaustively simulate the circuit and provide graphical and text output demonstrating that the model is correct.***



```
module Combo_CA (Y,A, B, C, D);
output Y;
input A, B, C, D;
assign Y = (~(A | D)) & (B & C & ~D);
endmodule
```

Task 3: Decoders

Decoder is a combinational circuit that converts binary information from an n input lines to a maximum of 2^n output lines. If the n-bit coded information has unused combinations, the decoder may have fewer than 2^n outputs. Consider decoder_2_to_4 below and verilog description.



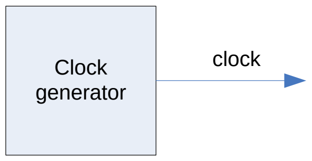
```
module decoder_2_to_4(d3, d2, d1, d0,s1, s0);
input s1, s0;
output d3, d2, d1, d0;
wire d3, d2, d1, d0; // wires for outputs
wire s1_n, s0_n; // interconnection wires
wire a3, a2, a1, a0; // interconnection wires
// Structural model of decoder
not n1 (s1_n, s1),
not n2 (s0_n, s0);
and a0 (d0,s1_n,so_n);
and a1 (d0,s1_n,so);
and a2 (d0,s1,so_n);
and a3 (d0,s1,so);
endmodule
```

Simulate the following decoder design

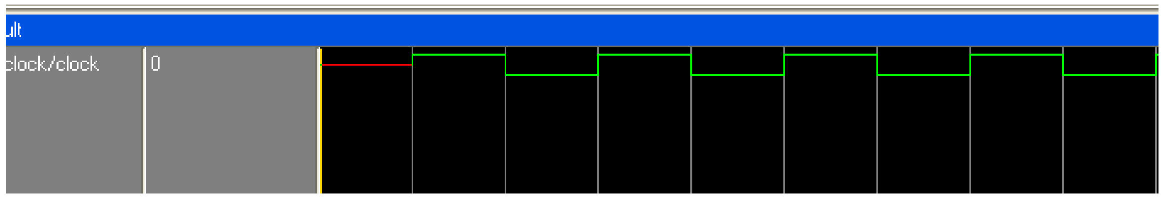
```
module decoder_case (binary_in,decoder_out,enable);
  input [3:0] binary_in  ; //  4 bit binary input
  input  enable ;   //  Enable for the decoder
  output [15:0] decoder_out ;  //  16-bit  out

  reg [15:0] decoder_out ;

  always @ (enable or binary_in)
  begin
    decoder_out = 0;
    if (enable) begin
      case (binary_in)
        4'h0 : decoder_out = 16'h0001;
        4'h1 : decoder_out = 16'h0002;
        4'h2 : decoder_out = 16'h0004;
        4'h3 : decoder_out = 16'h0008;
        4'h4 : decoder_out = 16'h0010;
        4'h5 : decoder_out = 16'h0020;
        4'h6 : decoder_out = 16'h0040;
        4'h7 : decoder_out = 16'h0080;
        4'h8 : decoder_out = 16'h0100;
        4'h9 : decoder_out = 16'h0200;
        4'hA : decoder_out = 16'h0400;
        4'hB : decoder_out = 16'h0800;
        4'hC : decoder_out = 16'h1000;
        4'hD : decoder_out = 16'h2000;
        4'hE : decoder_out = 16'h4000;
        4'hF : decoder_out = 16'h8000;
      endcase
    end
```

Task 4 : Clock generators

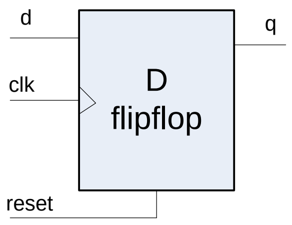
Clock generators are used in testbenches to provide a clock signal for testing the model of a synchronous circuit. A flexible clock generator will be parameterized for a variety of applications. The forever loop causes unconditional repetitive execution of statements, subject to the disable statement, and is a convenient construct for describing clocks.



```
module test_clock();
parameter half_cycle = 50;
reg clock;
initial
begin: clock_label // Note: clock_loop is a named block of statements
forever
begin
#half_cycle clock = 1;
#half_cycle clock = 0;
end
end
// always #100 clock = ~clock;  another way of declaring clock
endmodule
```



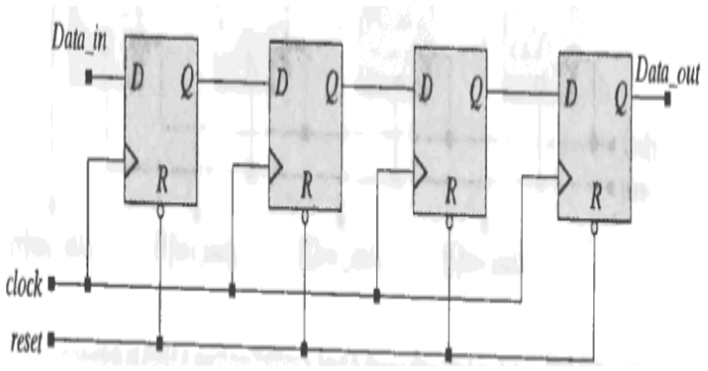
Task5: D Flipt flop



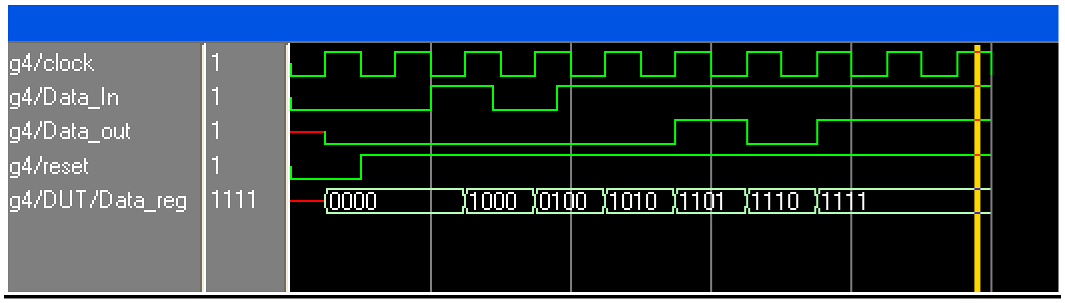
```
module dff (q,d,clk,reset);
input d, clk, reset ;

output q;
reg q;
 always @ ( posedge clk)
 if (~reset)
   q <= 1'b0;
 else
   q <= d;
  endmodule
```

Task 6: Shift register

Here,shift register declares an internal 4-bit register, Data_reg, which creates Data_out by a continuous assignment to the least significant bit (LSB) of the register and forms the register contents synchronously from a concatenation of the scalar Data_in with the three leftmost bits of the register. Notice that the register variable, Data_reg, is referenced by concatenation in a nonblocking assignment before it is assigned value in a synchronous behavior. This implies the need for memory, and synthesizes to the flip-flop structure shown . Also, recall that the values on the RHS of the non blocking assignments are the values of the variables immediately before the active edge of the clock, and the values on the LHS are the values formed after the edge.



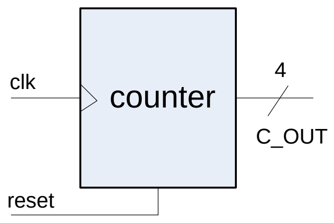
```
module Shif_reg4 (Data_out, Data_In,
clock, reset);
output Data_out;
input Data_In, clock, reset;
reg [3: 0] Data_reg;
assign Data_out = Data_reg[0];
always @ (posedge clock)
begin
if (reset == 1'b0) Data_reg <= 4'b0;
else Data_reg <= {Data_In,
Data_reg[3:1]};
end
endmodule
```



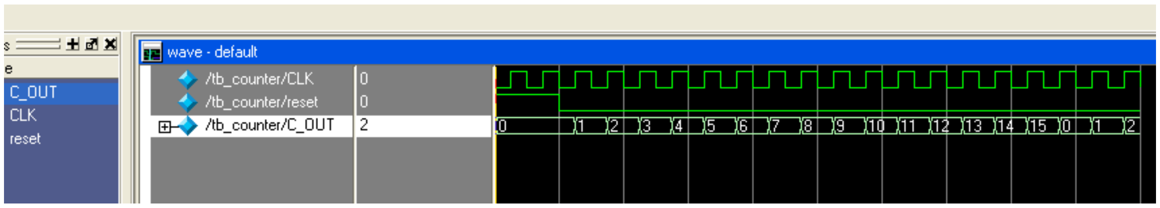
4-bit universal shift register

A 4-bit universal shift register is an important unit of digital machines that employ a bit-slice architecture, with multiple identical slices of a 4-bit shift register chained together with additional logic to form a wider and more versatile datapath. Its features include synchronous reset, parallel inputs, parallel outputs, bidirectional serial input from either the LSB or the most significant bit (MSB), and bidirectional serial output to either the LSB or the MSB. In the serial-in, serial-out mode the machine can delay an input signal for 4 clock ticks, and act as a uni-directional shift register. In parallel-in, serial-out mode it operates as a parallel-to-serial converter, and in the serial-in, parallel-out mode it operates as a serial-to-parallel converter. Its parallel-in, parallel-out mode, combined with shift operations, allows it to perform any of the operations of less versatile unidirectional shift registers.

Task 7: Counters

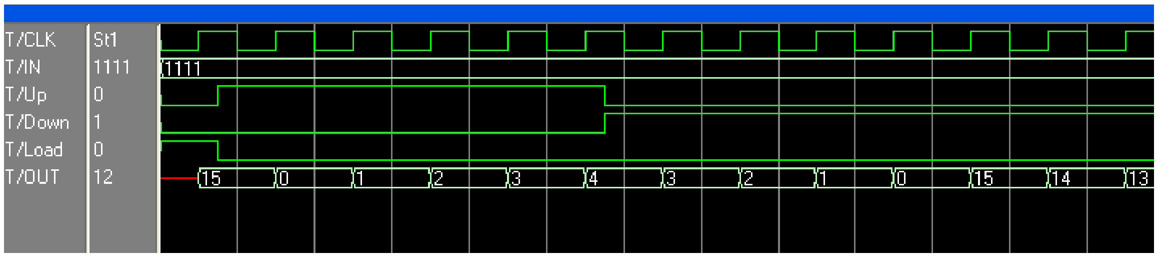


```
module counter (C_OUT,CLK,reset);
  output   [3: 0]      C_OUT;
  input    CLK,reset;
  reg      [3:0]       C_OUT;

  always @ (posedge CLK)
  begin
  if (reset) C_OUT <= 4'b0000;
  else
  C_OUT <= C_OUT + 4'b0001;
  end
  endmodule
```



Task: Write and verify the HDL beharioral description of a four-bit updown counter with parallel load using the following control inputs: (a) The counter has three control inputs for the three operations Load, Up, and Down. The order of precedence is Load, Up, and Down. (b) The counter has two selection inputs to specify four operations: Up, Down , Load, and no change.

```
module updown (OUT, Up, Down, Load, IN, CLK);
output     [3: 0]     OUT;
input      [3: 0]     IN;
input                            Up, Down, Load, CLK;
reg        [3:0]      OUT;

always @ (posedge CLK)
if (Load) OUT <= IN;
else if (Up)          OUT <= OUT + 4'b0001;
else if (Down)        OUT <= OUT - 4'b0001;
else                  OUT <= OUT;
endmodule
```



Your Lab Assignment 10 (75 points)

Submission Your submission must contain:

The source code of your design/testbench if any (for each of the problem) and answer document (no page limit). Code should reasonably well commented. Please submit document as a pdf (single) with file name rollNo_Lab10.pdf.
Lab submission through cs225.iitp@gmail.com with subject: YourrollNo_Lab10

Q1.
Design and test a verilog model for 4 bit Binary to Gray code Converter.

(5 points)

Q2:
Develop a sequential circuit that has a single data input signal, S, and produces an output Y. The output is 1 whenever S has the same value over three successive clock cycles, and 0 otherwise. Assume that the value of S for a given clock cycle is defi ned at the time of the rising clock edge at the end of the clock cycle.

(5 points)

Q3:

Design and verify an 16 bit ALU which performs the following operation: Add, sub, xor, and, or, increment, left shift, right shift. The ALU should indicate a zero flag if the result operation is zero.

(10 points)

Q4:
Design an encoder for use in a domestic burglar alarm that has sensors for each of eight zones. Each sensor signal is 1 when an intrusion is detected in that zone, and 0 otherwise. The encoder has three bits of output, encoding the zone as follows:
Zone 1: 000 ; Zone 2: 001; Zone 3: 010; Zone 4: 011
Zone 5: 100; Zone 6: 101; Zone 7: 110; Zone 8: 111

(5 points)

Q5: Write a test bench for the following Verilog models
(a)
```
module vat_buzzer_behavior  ( output  buzzer,
    input  above_25_0, above_30_0, low_level_0,
    input  above_25_1, above_30_1, low_level_1,
    input  select_vat_1 );

  assign buzzer =
    select_vat_1 ? low_level_1 | (above_30_1 | ~above_25_1)
                 : low_level_0 | (above_30_0 | ~above_25_0);
Endmodule
```

(b)
```
module alarm_priority ( output [2:0] intruder_zone,
                        output       valid,
                        input  [1:8] zone );

  wire [1:8] winner;

  assign winner[1] = zone[1];
  assign winner[2] = zone[2] & ~zone[1];
  assign winner[3] = zone[3] & ~(zone[2] | zone[1]);
  assign winner[4] = zone[4] & ~(zone[3] | zone[2] | zone[1]);
  assign winner[5] = zone[5] & ~(zone[4] | zone[3] | zone[2] |
                               zone[1]);
  assign winner[6] = zone[6] & ~(zone[5] | zone[4] | zone[3] |
                               zone[2] | zone[1]);
  assign winner[7] = zone[7] & ~(zone[6] | zone[5] | zone[4] |
                               zone[3] | zone[2] | zone[1]);
  assign winner[8] = zone[8] & ~(zone[7] | zone[6] | zone[5] |
                               zone[4] | zone[3] | zone[2] |
                               zone[1]);
  assign intruder_zone[2] = winner[5] | winner[6] |
                          winner[7] | winner[8];
  assign intruder_zone[1] = winner[3] | winner[4] |
                          winner[7] | winner[8];
  assign intruder_zone[0] = winner[2] | winner[4] |
                          winner[6] | winner[8];

  assign valid = zone[1] | zone[2] | zone[3] | zone[4] |
                 zone[5] | zone[6] | zone[7] | zone[8];

endmodule
```

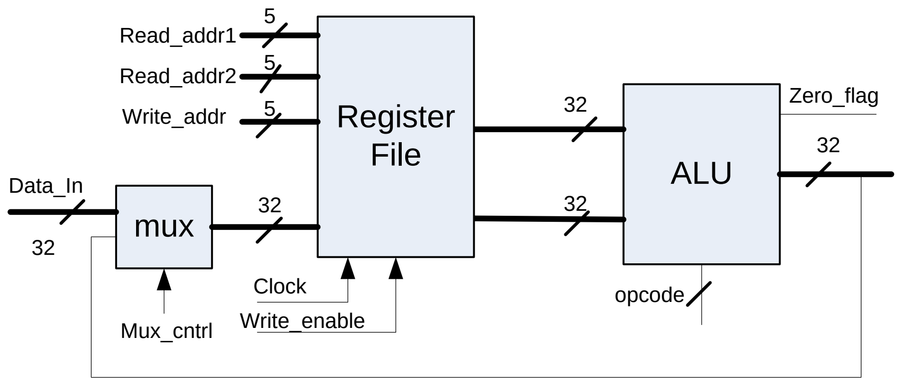
: **(5 points)**

Q6: Simulate the following models by writing appropriate test bench.
```
module decade_counter ( output reg [3:0] q,
                        input            clk );

  always @(posedge clk)
    q <= q == 9 ? 0 : q + 1;

endmodule


 module decoded_counter ( output ctrl,
                          input  clk );
  reg [3:0] count_value;

  always @(posedge clk)
    count_value <= count_value + 1;

  assign ctrl = count_value == 4'b0111 ||
                count_value == 4'b1011;

endmodule
```

(5 points)

Q7:

Combine the ALU from problem 3 and a register file to form an architecture like the one that shown in Figure below . Develop a testbench to verify each functional unit and the overall structure. Write random data to registers 0 to 24 and find sum those data and store the result in register 31. Write test to find the sum of the contents of the register 0 to 24 and store the result in register 31. (40 points)


Read_addr1
5
Read_addr2
5
Write_addr
5
Register File
32
32
Zero_flag
32
ALU
Data_In
32
mux
32
Clock
Write_enable
Mux_cntrl
opcode