1. Write a Verilog code for 2X4 decoder.

DATA FLOW MODELLING:

```
module decoder(d0,d1,d2,d3,e,a,b);
  input e,a,b;
  output d0,d1,d2,d3;
  assign d0=e&~a&~b;
  assign d1=e&~a&b;
  assign d2=e&a&~b;
  assign d3=e&a&b;
endmodule
```

//testbench//

```
module testbench();

  reg t_e, t_a, t_b;
  wire t_d0, t_d1, t_d2, t_d3;

  decoder dut(.e(t_e), .a(t_a), .b(t_b), .d0(t_d0), .d1(t_d1),
.d2(t_d2), .d3(t_d3));

  initial begin
    t_e = 0; t_a = 1; t_b = 1;
    #100;
    t_e = 1; t_a = 0; t_b = 0;
    #100;
    t_e = 1; t_a = 0; t_b = 1;
    #100;
    t_e = 1; t_a = 1; t_b = 0;
    #100;
```

```
   t_e = 1; t_a = 1; t_b = 1;
   $finish();
  end
endmodule
```

BEHAVIORAL MODELLING:

```
module decoder(d0,d1,d2,d3,e,a,b);
  input e,a,b;
  output d0,d1,d2,d3;
  reg d0,d1,d2,d3;
    always @(e,a,b)
      begin
        d0 = e & ~ a & ~ b;
        d1 = e & ~ a & b;
        d2 = e & a & ~ b;
        d3 = e & a & b;
        end
    endmodule

//testbench//
//verilog code
module decoder(d0,d1,d2,d3,e,a,b);
  input e, a, b;
  output d0, d1, d2, d3;
  assign d0 = e & ~a & ~b;
  assign d1 = e & ~a & b;
  assign d2 = e & a & ~b;
  assign d3 = e & a & b;
endmodule
```

//testbench//

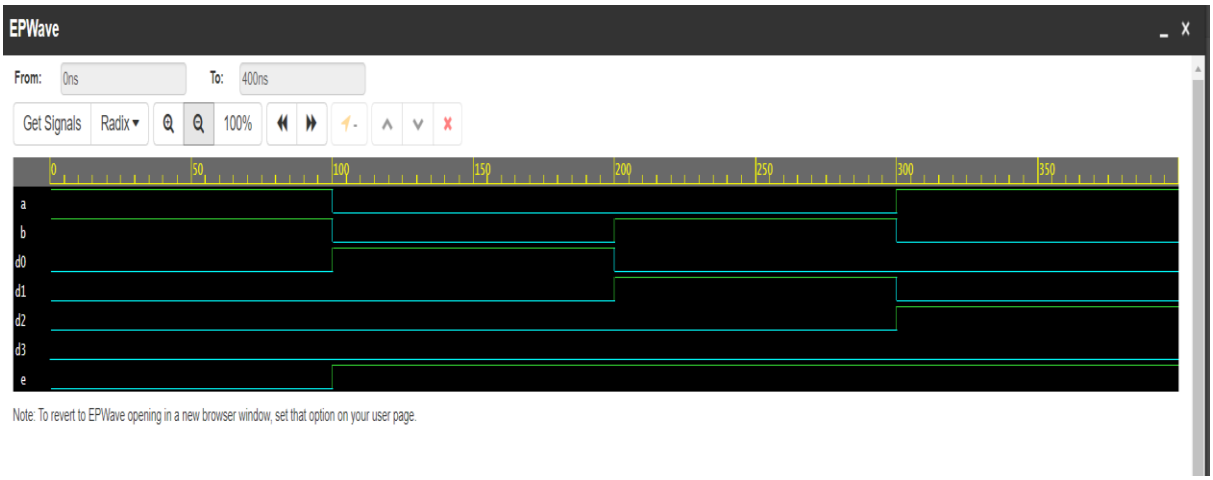
```
module testbench();
  reg t_e, t_a, t_b;
  wire t_d0, t_d1, t_d2, t_d3;

  decoder dut(.e(t_e), .a(t_a), .b(t_b), .d0(t_d0), .d1(t_d1),
.d2(t_d2), .d3(t_d3));

  initial begin
    t_e = 0; t_a = 1; t_b = 1;
    #100;
    t_e = 1; t_a = 0; t_b = 0;
    #100;
    t_e = 1; t_a = 0; t_b = 1;
    #100;
    t_e = 1; t_a = 1; t_b = 0;
    #100;
    t_e = 1; t_a = 1; t_b = 1;
    $finish();
  end
endmodule
```

GATE LEVEL MODELLING:

```
module decoder(d0,d1,d2,d3,a,b);
  input a,b;
  output d0,d1,d2,d3;
  wire w1,w2;
  not g1(w1,a);
  not g2(w2,b);
  and g3(d0,w1,w2);
```

```
  and g4(d1,w1,b);
  and g5(d2,a,w2);
  and g6(d3,a,b);
endmodule

//testbench//
module testbench();
  reg a, b;
  wire d0, d1, d2, d3;

  decoder dut(.a(a), .b(b), .d0(d0), .d1(d1), .d2(d2), .d3(d3));

  initial begin
    $dumpfile("testbench.vcd");
    $dumpvars(0, testbench);

    // Test case 1
    a = 0; b = 0;
    #10;

    // Test case 2
    a = 0; b = 1;
    #10;

    // Test case 3
    a = 1; b = 0;
    #10;

    // Test case 4
    a = 1; b = 1;
```

```
    #10;

    $finish;
  end
endmodule
```



2.Write a Verilog code for Full subtractor.

Dataflow modelling:

```
module full_subtractor(
  bout, d, a, b, c
);
  input a, b, c;
  output bout, d;

  assign d = a ^ b ^ c;
  assign bout = (~a & b) || (c & ~(a ^ b));
endmodule

                                    //Testbench//
module testbench();
  reg t_a, t_b, t_c;
  wire t_bout, t_d;
  full_subtractor dut(.a(t_a), .b(t_b), .c(t_c), .bout(t_bout),
.d(t_d));
  initial begin
    $dumpfile("testbench.vcd");
    $dumpvars(0, testbench);
  end


  initial begin
    t_a = 0; t_b = 0; t_c = 0; // Test case 1
    #10;
    t_a = 0; t_b = 0; t_c = 1; // Test case 2
```

```
    #10;
    t_a = 0; t_b = 1; t_c = 0; // Test case 3
    #10;
    t_a = 0; t_b = 1; t_c = 1; // Test case 4
    #10;
    t_a = 1; t_b = 0; t_c = 0; // Test case 5
    #10;
    t_a = 1; t_b = 0; t_c = 1; // Test case 6
    #10;
    t_a = 1; t_b = 1; t_c = 0; // Test case 7
    #10;
    t_a = 1; t_b = 1; t_c = 1; // Test case 8
    #10;
    $finish();
  end
endmodule
```

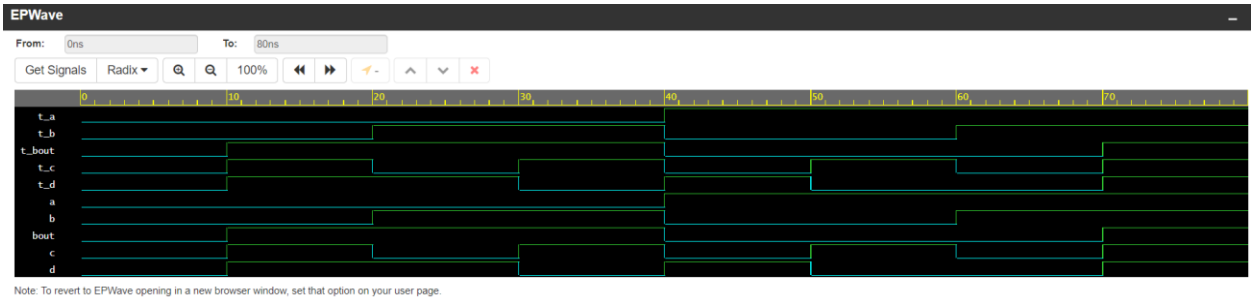
Behavioral modelling:

```
module full_subtractor(bout,d,a,b,c);
  input a,b,c;
  output bout,d;
  wire w1,w2,w3,w4,w5;
  xor g3(w1,a,b);
  not g1(w2,a);
  not g2(w3,w1);
  and g4(w4,w2,b);
  and g5(w5,w3,c);
  xor g6(d,w1,c);
  or g7(bout,w4,w5);
endmodule
```

//Testbench//

```
module testbench;
 reg a, b, c;
 wire bout, d;

 full_subtractor dut(
  .a(a),
  .b(b),
  .c(c),
  .bout(bout),
  .d(d)
 );

 initial begin
  $dumpfile("testbench.vcd");
  $dumpvars(0, testbench);

  // Test case 1
  a = 0; b = 0; c = 0;
  #10;

  // Test case 2
  a = 0; b = 0; c = 1;
  #10;

  // Test case 3
  a = 0; b = 1; c = 0;
  #10;

  // Test case 4
```

```
    a = 0; b = 1; c = 1;
    #10;

    // Test case 5
    a = 1; b = 0; c = 0;
    #10;

    // Test case 6
    a = 1; b = 0; c = 1;
    #10;

    // Test case 7
    a = 1; b = 1; c = 0;
    #10;

    // Test case 8
    a = 1; b = 1; c = 1;
    #10;

    $finish;
  end
endmodule
```



3.Write a Verilog code for 2-bit comparator.

DATAFLOW MODELLING:

```
module comparator_2bit(
   input [1:0] A,
   input [1:0] B,
   output reg EQ,
   output reg GT,
   output reg LT
);

   assign EQ = (A[1] & ~B[1] & A[0] & ~B[0]) | (~A[1] & B[1] & ~A[0] & B[0]);
   assign GT = (A[1] & ~B[1]) | ((A[1] ^ B[1]) & ~A[0]);
   assign LT = (B[1] & ~A[1]) | ((A[1] ^ B[1]) & ~B[0]);

endmodule
```

```
                                        //Testbench//
module testbench;

  reg [1:0] A;
  reg [1:0] B;
  wire EQ, GT, LT;

  // Instantiate the comparator
  comparator_2bit uut (
     .A(A),
     .B(B),
     .EQ(EQ),
     .GT(GT),
     .LT(LT)
  );
 initial begin
  $dumpfile("testbench.vcd");
  $dumpvars(0, testbench);
 end

  // Initialize inputs and monitor outputs
  initial begin
```

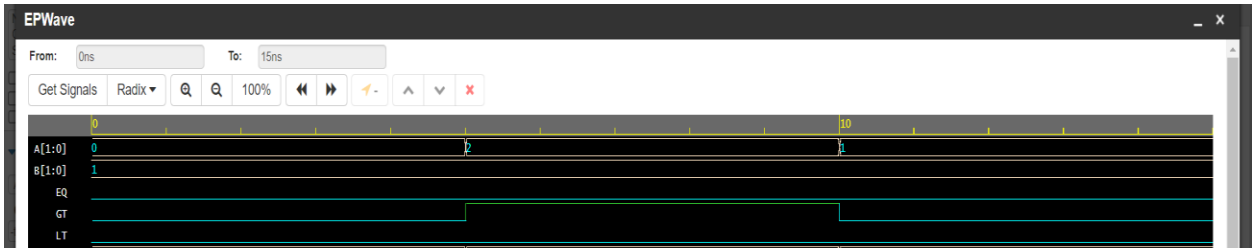
```
        $monitor("Time = %0t: A = %b, B = %b, EQ = %b, GT = %b, LT =
%b", $time, A, B, EQ, GT, LT);

        A = 2'b00;
        B = 2'b01;
        #5;

        A = 2'b10;
        B = 2'b01;
        #5;

        A = 2'b01;
        B = 2'b01;
        #5;

        // Add more test cases here

        $finish;
    end

endmodule
```

Behavioral modelling:

```
module comparator_2bit(
   input [1:0] A,
   input [1:0] B,
   output reg EQ,
   output reg GT,
   output reg LT
);

   always @* begin
      EQ = (A == B);
      GT = (A > B);
      LT = (A < B);
   end

endmodule
                                        //Testbench//
    module testbench;

       reg [1:0] A;
       reg [1:0] B;
       wire EQ, GT, LT;

       // Instantiate the comparator
```

```
  comparator_2bit uut (
      .A(A),
      .B(B),
      .EQ(EQ),
      .GT(GT),
      .LT(LT)
  );
 initial begin
  $dumpfile("testbench.vcd");
  $dumpvars(0,testbench);
 end

  // Initialize inputs and monitor outputs
  initial begin
      $monitor("Time = %0t: A = %b, B = %b, EQ = %b, GT = %b, LT =
%b", $time, A, B, EQ, GT, LT);

      A = 2'b00;
      B = 2'b01;
      #5;

      A = 2'b10;
      B = 2'b01;
      #5;

      A = 2'b01;
      B = 2'b01;
      #5;

      // Add more test cases here
```

```
        $finish;
    end

endmodule
```



4. Write a Verilog code for 3 bit binary to gray convertor.

Dataflow modelling:

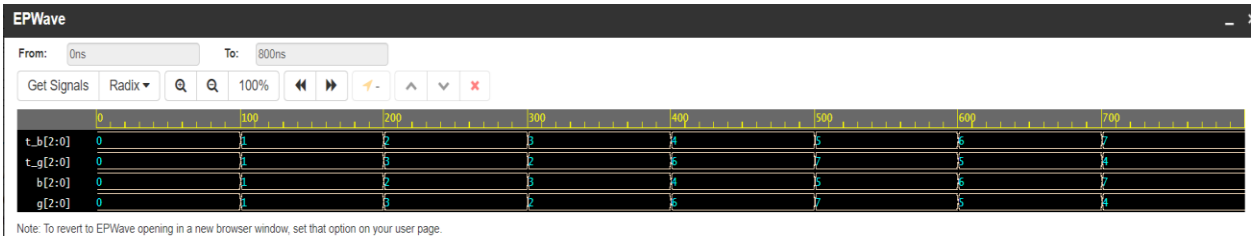
```
module binary_to_gray(g,b);
  input [2:0]b;
```

```
  output [2:0]g;
  assign g[2]=b[2];
  assign g[1]=b[2]^b[1];
  assign g[0]=b[1]^b[0];
endmodule

                                    //Testbench//
module testbench();
  reg [2:0]t_b;
  wire [2:0]t_g;
  binary_to_gray dut(.g(t_g), .b(t_b));
  initial begin
    $dumpfile("testbench.vcd");
    $dumpvars(0,testbench);
  end
  initial begin
    t_b[2]=0;t_b[1]=0;t_b[0]=0;
    #100
    t_b[2]=0;t_b[1]=0;t_b[0]=1;
    #100
    t_b[2]=0;t_b[1]=1;t_b[0]=0;
    #100
    t_b[2]=0;t_b[1]=1;t_b[0]=1;
    #100
    t_b[2]=1;t_b[1]=0;t_b[0]=0;
    #100
    t_b[2]=1;t_b[1]=0;t_b[0]=1;
    #100
    t_b[2]=1;t_b[1]=1;t_b[0]=0;
```

```
        #100
        t_b[2]=1;t_b[1]=1;t_b[0]=1;
        #100
        $finish();
      end
    endmodule
```

Behavioral modelling:

```
module binary_to_gray(g,b);
  input [2:0]b;
  output [2:0]g;
  reg [2:0]g;
  always @(b[2],b[1],b[0]);
  initial begin
  g[2]=b[2];
  g[1]=b[2]^b[1];
  g[0]=b[1]^b[0];
  end
endmodule
```



5. Write a Verilog code for BCD to excess 3 convertors.

Dataflow modelling:

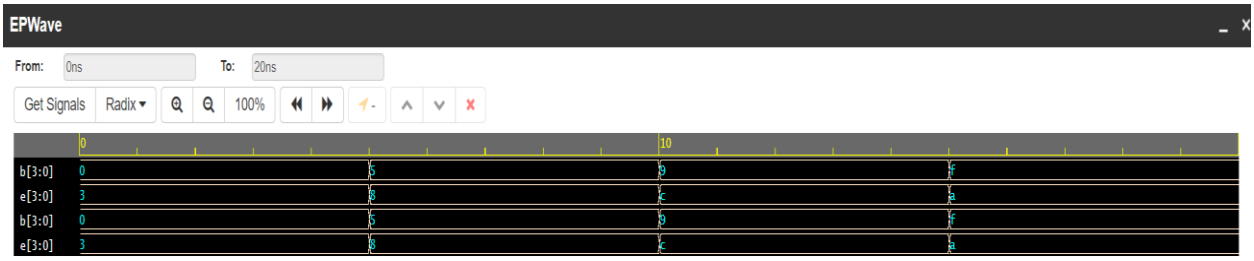
```
module bcd_to_excess3(e,b);
  input [3:0]b;
  output [3:0]e;
  assign e[3]=((b[3])||(b[2]&b[0]||(b[2]&b[1])));
  assign e[2]=((b[2]&~b[1]&~b[0])||(~b[2]&b[0])||(~b[2]&b[1]));
  assign e[1]=(~b[1]&~b[0]||(b[1]&b[0]));
  assign e[0]=~b[0];
endmodule
                                        //Testbench//
module testbench;

    reg [3:0] b;
    wire [3:0] e;

    // Instantiate the bcd_to_excess3 module
    bcd_to_excess3 uut (
        .b(b),
```

```
    .e(e)
  );
initial begin
  $dumpfile("testbench.vcd");
  $dumpvars(0,testbench);
end

  // Initialize inputs and monitor outputs
  initial begin
    $monitor("Time = %0t: BCD = %b, Excess-3 = %b", $time, b, e);

    b = 4'b0000;
    #5;

    b = 4'b0101;
    #5;

    b = 4'b1001;
    #5;

    b = 4'b1111;
    #5;
```

```
        // Add more test cases here

        $finish;
    end

endmodule
```

Behavioral modelling:

```
module bcd_to_excess3(
    input [3:0] b,
    output reg [3:0] e
);
    always @* begin
        e[3] = (b[3]) || (b[2] & b[0]) || (b[2] & b[1]);
        e[2] = (b[2] & ~b[1] & ~b[0]) || (~b[2] & b[0]) || (~b[2] & b[1]);
        e[1] = (~b[1] & ~b[0]) || (b[1] & b[0]);
        e[0] = ~b[0];
    end
endmodule
```



Note: To revert to EPWave opening in a new browser window, set that option on your user page.

ASSIGNMENT 2

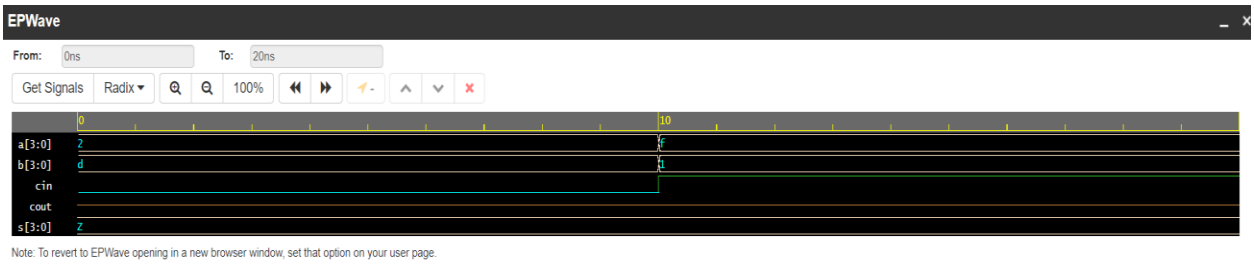
[1] Design 4-bit Ripple Carry Adder with the help of 1-bit adder.

```
module full_adder(
   input a,
   input b,
   input cin,
   output sum,
   output carry
);
   assign sum = a ^ b ^ cin;
   assign carry = (a & b) | (cin & (a ^ b));
endmodule



module rca_4bit(
   output [3:0] s,
   output cout,
   input [3:0] a,
   input [3:0] b,
   input cin
);
```

```
    wire c0, c1, c2, c3;
    full_adder fa0(s[0], c0, a[0], b[0], cin);
    full_adder fa1(s[1], c1, a[1], b[1], c0);
    full_adder fa2(s[2], c2, a[2], b[2], c1);
    full_adder fa3(s[3], c3, a[3], b[3], c2);
    assign cout = c3;
endmodule
                                        //Testbench//
module testbench;
    reg [3:0] a;
    reg [3:0] b;
    reg cin;
    wire [3:0] s;
    wire cout;

    rca_4bit rca_inst(
        .s(s),
        .cout(cout),
        .a(a),
        .b(b),
        .cin(cin)
    );
```

```
initial begin
  $dumpfile("testbench.vcd");
  $dumpvars(0,testbench);
end


  initial begin
     $display("a   b   cin | s   cout");
     $monitor("%b %b %b | %b %b", a, b, cin, s, cout);

     a = 4'b0010;
     b = 4'b1101;
     cin = 0;
     #10;

     a = 4'b1111;
     b = 4'b0001;
     cin = 1;
     #10;

     $finish;
  end
```

endmodule



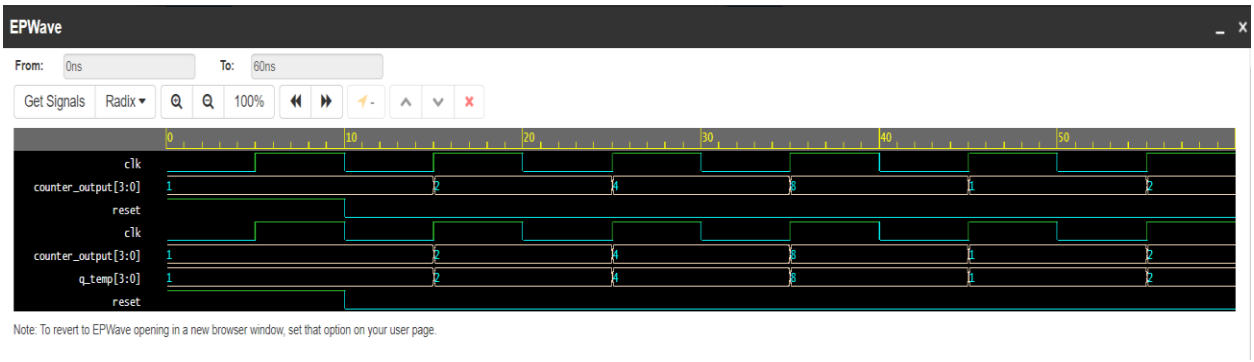
2] Design D-flipflop and reuse it to implement 4- bit Johnson Counter.

Dataflow modelling:

```
module Johnson_Counter (
   input wire clk,
   input wire reset,
   output reg [3:0] counter_output
);

   reg [3:0] q_temp;

   always @(posedge clk or posedge reset) begin
      if (reset)
         q_temp <= 4'b0001;  // Initial value
      else
         q_temp <= {q_temp[2:0], q_temp[3]};
```

```
    end

    assign counter_output = q_temp;

endmodule
                                        //Testbench//
module testbench;

    reg clk;
    reg reset;
    wire [3:0] counter_output;

    Johnson_Counter dut (
        .clk(clk),
        .reset(reset),
        .counter_output(counter_output)
    );
  initial begin
    $dumpfile("testbench.vcd");
    $dumpvars(0,testbench);
  end
```

```
    always begin
        #5 clk = ~clk;  // Toggle the clock every 5 time units
    end

    initial begin
        clk = 0;
        reset = 1;  // Start with reset active
        #10 reset = 0;  // Deactivate reset after 10 time units

        // Wait for a few clock cycles to observe the counter sequence
        #50;

        $finish;  // End simulation
    end

endmodule
```



[3] Reuse 2:1 Mux code to implement 8:1 Mux.

```
module Mux2to1 (
    input wire sel,
    input wire d0,
    input wire d1,
    output wire y
);

    assign y = (sel == 1'b0) ? d0 : d1;

endmodule

module Mux8to1 (
    input wire [2:0] sel,
    input wire [7:0] d,
    output wire y
);

    wire [1:0] sel0, sel1;
    wire [3:0] d0, d1;
    wire [1:0] d2;
```

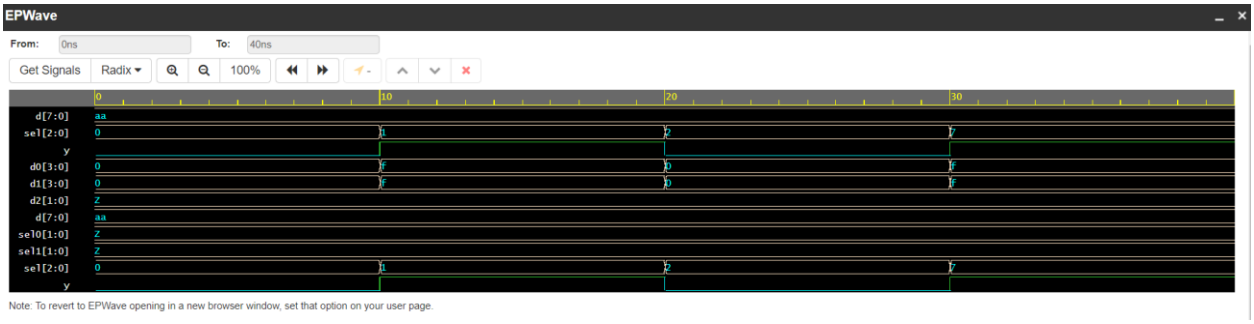
```
// First stage of 2:1 muxes
Mux2to1 mux0 (
    .sel(sel[0]),
    .d0(d[0]),
    .d1(d[1]),
    .y(d0[0])
);

Mux2to1 mux1 (
    .sel(sel[0]),
    .d0(d[2]),
    .d1(d[3]),
    .y(d0[1])
);

Mux2to1 mux2 (
    .sel(sel[0]),
    .d0(d[4]),
    .d1(d[5]),
    .y(d0[2])
);
```

```
Mux2to1 mux3 (
    .sel(sel[0]),
    .d0(d[6]),
    .d1(d[7]),
    .y(d0[3])
);

// Second stage of 2:1 muxes
Mux2to1 mux4 (
    .sel(sel[1]),
    .d0(d0[0]),
    .d1(d0[1]),
    .y(d1[0])
);

Mux2to1 mux5 (
    .sel(sel[1]),
    .d0(d0[2]),
    .d1(d0[3]),
    .y(d1[1])
);
```

```
Mux2to1 mux6 (
    .sel(sel[1]),
    .d0(d0[0]),
    .d1(d0[1]),
    .y(d1[2])
);

Mux2to1 mux7 (
    .sel(sel[1]),
    .d0(d0[2]),
    .d1(d0[3]),
    .y(d1[3])
);

// Third stage of 2:1 muxes
Mux2to1 mux8 (
    .sel(sel[2]),
    .d0(d1[0]),
    .d1(d1[1]),
    .y(y)
);
```

```
endmodule
                                        //Testbench//
module testbench();

   reg [2:0] sel;
   reg [7:0] d;
   wire y;

   Mux8to1 dut (
      .sel(sel),
      .d(d),
      .y(y)
   );
   initial begin
   $dumpfile("testbench.vcd");
   $dumpvars(0,testbench);
  end


   initial begin
      // Initialize inputs
      sel = 3'b000; // Select input 0 (000 in binary)
```

```
    d = 8'b10101010; // Arbitrary data input

    // Wait for a few time units before changing inputs
    #10;

    // Test with different selection values
    sel = 3'b001; // Select input 1 (001 in binary)
    #10;

    sel = 3'b010; // Select input 2 (010 in binary)
    #10;

    sel = 3'b111; // Select input 7 (111 in binary)
    #10;

    $finish; // End simulation
  end

endmodule
```



[4] Design a Full Subtractor with Gate Level Modeling Style.(use primitive gates)

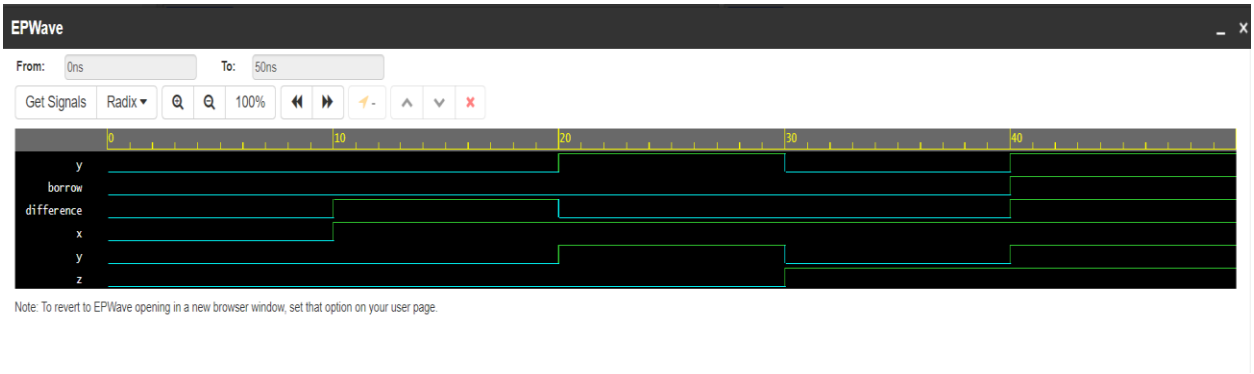
```
module full_subtractor_gate_level(
    input x,
    input y,
    input z,
    output difference,
    output borrow
    );

    xor_21 x1(x,y,k1);
    xor_21 x2(k1,z,difference);

    and_21 a1(~x,y,k2);
    and_21 a2(~x,z,k3);
    and_21 a3(y,z,k4);
```

```
   or_21 o1(k2,k3,k5);
   or_21 o2(k5,k4,borrow);

endmodule
                                        //Testbench//
module xor_21 (
   input wire a,
   input wire b,
   output wire y
);
   assign y = a ^ b;
endmodule

module and_21 (
   input wire a,
   input wire b,
   output wire y
);
   assign y = a & b;
endmodule
```

```
module or_21 (
    input wire a,
    input wire b,
    output wire y
);
    assign y = a | b;
endmodule

module testbench;
    reg x, y, z;
    wire difference, borrow;

    full_subtractor_gate_level uut(
        .x(x),
        .y(y),
        .z(z),
        .difference(difference),
        .borrow(borrow)
    );

    initial begin
        $dumpfile("testbench.vcd");
```

```
$dumpvars(0, testbench);

// Initialize inputs
x = 0; y = 0; z = 0;

// Wait for a few time units before changing inputs
#10;

// Change inputs and observe outputs
x = 1; y = 0; z = 0;
#10;

x = 1; y = 1; z = 0;
#10;

x = 1; y = 0; z = 1;
#10;

x = 1; y = 1; z = 1;
#10;

// End simulation
```

```
        $finish;
    end
endmodule
```



[5] Design a 2X4 decoder using gate level modelling.

```
module decoder_24_gate_level(
    input x1,
    input x2,
    output y1,
    output y2,
    output y3,
    output y4
    );

    and a1(y1,~x1,~x2);
    and a2(y2,~x1,x2);
    and a3(y3,x1,~x2);
```

```
   and a4(y4,x1,x2);


endmodule
                                        //Testbench//
module decoder_24_tb;
  reg x1,x2;
  wire y1,y2,y3,y4;


  initial begin


    $monitor(" %t | x1 = %b | x2 = %b | y1 = %b | y2 = %b | y3 = %b | y4
= %b ",$time,x1,x2,y1,y2,y3,y4);


  end
   decoder_24_gate_level uut(x1,x2,y1,y2,y3,y4);
    initial begin


    #000 x1=0; x2=0;
    #100 x1=0; x2=1;
    #100 x1=1; x2=0;
    #100 x1=1; x2=1;
    #100 $finish;
```

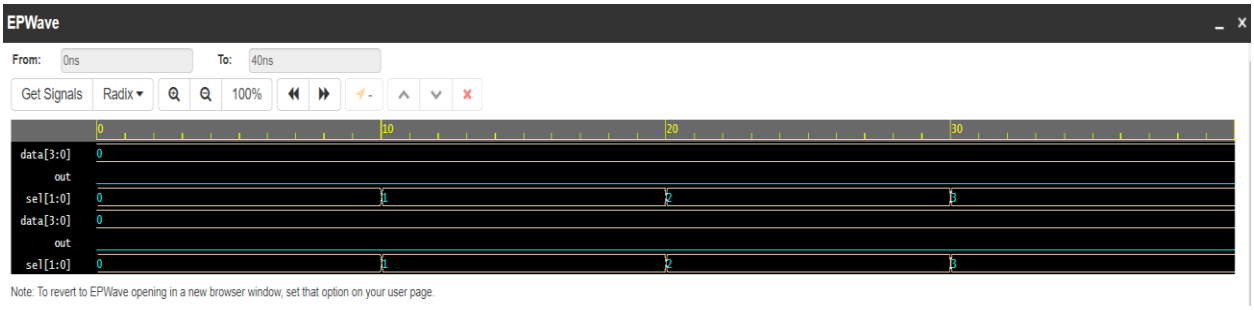
```
 end

  initial begin

    $dumpfile("dump.vcd");
    $dumpvars(0);

  end

endmodule
```

[6] Design a 4x1 mux using operators. (use data flow)

```
module Mux4x1 (
    input wire [3:0] data,
    input wire [1:0] sel,
    output wire out
);

    assign out = (sel == 2'b00) ? data[0] :
                 (sel == 2'b01) ? data[1] :
```

```
            (sel == 2'b10) ? data[2] :
            (sel == 2'b11) ? data[3] : 1'bx;

Endmodule
                                        //Testbench//
module testbench_mux4x1;

   reg [3:0] data;
   reg [1:0] sel;
   wire out;

   Mux4x1 uut (
      .data(data),
      .sel(sel),
      .out(out)
   );

   initial begin
      $dumpfile("testbench_mux4x1.vcd");
      $dumpvars(0, testbench_mux4x1);

      // Initialize inputs
```

```
        data = 4'b0000;
        sel = 2'b00;

        // Wait for a few time units before changing inputs
        #10;

        // Test with different selection values
        sel = 2'b01;
        #10;

        sel = 2'b10;
        #10;

        sel = 2'b11;
        #10;

        // End simulation
        $finish;
    end
endmodule
```



[7] Design a Full adder using half adder.

```
module Half_Adder (
    input wire A,
    input wire B,
    output wire Sum,
    output wire Carry
);

    assign Sum = A ^ B;
    assign Carry = A & B;

endmodule

module Full_Adder (
    input wire A,
    input wire B,
    input wire Cin,
```

```
    output wire Sum,
    output wire Cout
);

    wire HA_Sum1, HA_Carry1, HA_Carry2;

    Half_Adder HA1 (.A(A), .B(B), .Sum(HA_Sum1), .Carry(HA_Carry1));
    Half_Adder HA2 (.A(HA_Sum1), .B(Cin), .Sum(Sum),
.Carry(HA_Carry2));

    assign Cout = HA_Carry1 | HA_Carry2;

endmodule
                                        //Testbench//
module testbench;

    reg A, B, Cin;
    wire Sum, Cout;

    Full_Adder uut (
        .A(A),
        .B(B),
```

```
    .Cin(Cin),
    .Sum(Sum),
    .Cout(Cout)
);

initial begin
    $dumpfile("testbench.vcd");
    $dumpvars(0, testbench);

    // Test cases
    A = 0; B = 0; Cin = 0;
    #10;

    A = 0; B = 0; Cin = 1;
    #10;

    A = 0; B = 1; Cin = 0;
    #10;

    A = 0; B = 1; Cin = 1;
    #10;
```

```
        A = 1; B = 0; Cin = 0;
        #10;

        A = 1; B = 0; Cin = 1;
        #10;

        A = 1; B = 1; Cin = 0;
        #10;

        A = 1; B = 1; Cin = 1;
        #10;

        $finish;
    end

endmodule
```