VLSI DESIGN AND AUTOMATION SUMMER INTERNSHIP

WEEK 4 SUBMISSION

RESOURCE: WEEK 1(ENC,NETLIST and DATA files are here)

VERILOG CODE AND WAVE FORM

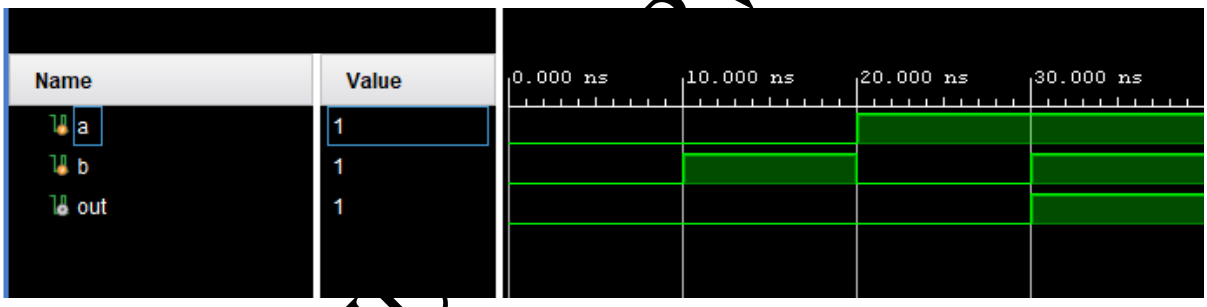
1. AND Gate

```
module and_gate(
input a, b,
output reg out);
always @(*) begin
if (a==1'b1 && b==1'b1)
out = 1'b1;
else
out = 1'b0;
end
endmodule
```

TB

```
module gate_and_TB();
reg a,b;
wire out;
and_gate dut(a, b, out);
initial begin
a = 0;
b = 0;
#10;

a = 0;
b = 1;
```

```
#10;

a = 1;
b = 0;
#10;

a = 1;
b = 1;
#10;

//$finish;
end
endmodule
```



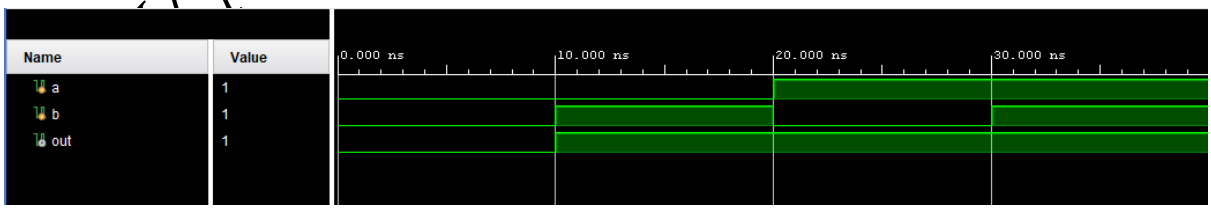
2. OR gate

```
module gate_or
input a, b;
output reg out);

always @(*) begin
if (a==1'b0 && b==1'b0)
out = 1'b0;
else
out = 1'b1;
```

```
end
endmodule
```

TB

```
module gate_or_tb;
reg a, b;
wire out;

gate_or uut (
    .a(a),
    .b(b),
    .out(out)
);

initial begin
    a = 0; b = 0; #10;
    a = 0; b = 1; #10;
    a = 1; b = 0; #10;
    a = 1; b = 1; #10;
    $finish;
end
endmodule
```

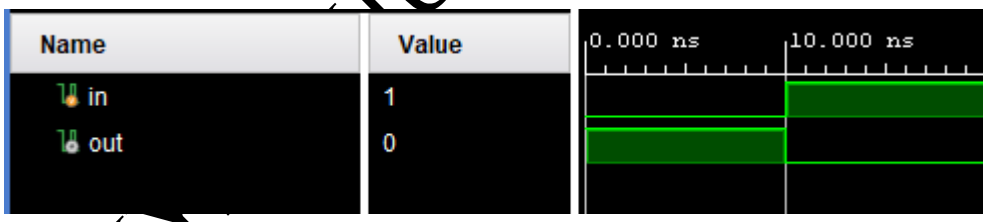
3. NOT gate

```
module gate_not(
input in,
output out);
```

```
assign out = ~in;
endmodule
```

TB

```
module gate_not_tb;
reg in;
wire out;

gate_not uut (
    .in(in),
    .out(out)
);

initial begin
    in = 0; #10;
    in = 1; #10;
    $finish;
end
endmodule
```



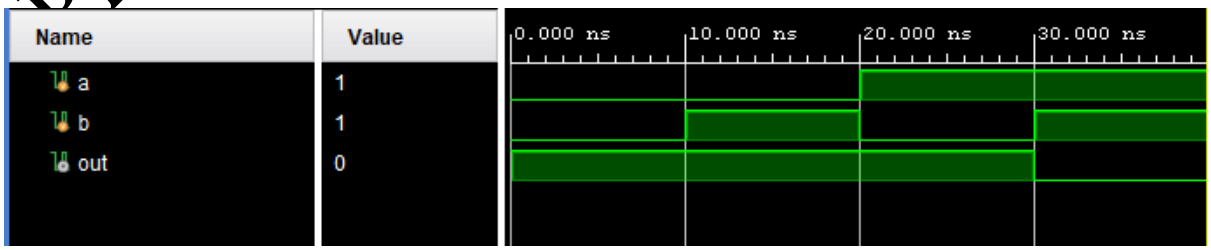
4. NAND Gate

```
module nand_gate(
input a, b,
output reg out);
always @(*) begin
if (a==1'b1 && b==1'b1)
out = 1'b0;
```

```
else
out = 1'b1;
end
endmodule
```

TB

```
module nand_gate_tb;
reg a, b;
wire out;

nand_gate uut (
    .a(a),
    .b(b),
    .out(out)
);

initial begin
    a = 0; b = 0; #10;
    a = 0; b = 1; #10;
    a = 1; b = 0; #10;
    a = 1; b = 1; #10;
    $finish;
end
endmodule
```

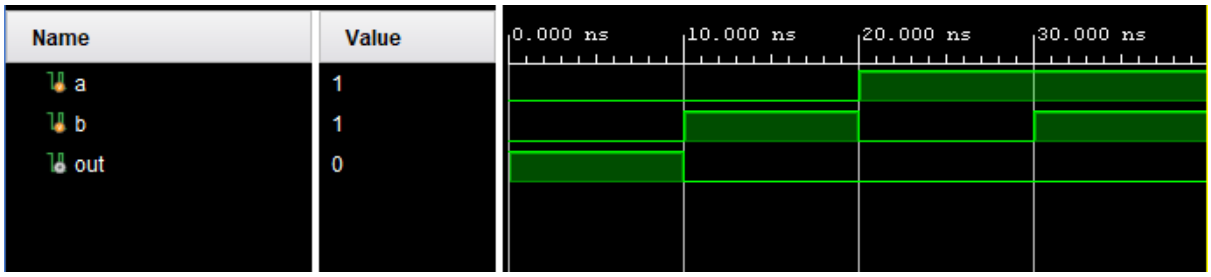
5. NOR Gate

```
module gate_nor(
input a, b,
output reg out);

always @(*) begin
if (a==1'b0 && b==1'b0)
out = 1'b1;
else
out = 1'b0;
end
endmodule
```

TB

```
module gate_nor_tb;
reg a, b;
wire out;

gate_nor uut (
    .a(a),
    .b(b),
    .out(out)
);

initial begin
    a = 0; b = 0; #10;
    a = 0; b = 1; #10;
    a = 1; b = 0; #10;
    a = 1; b = 1; #10;
    $finish;
end
```

endmodule



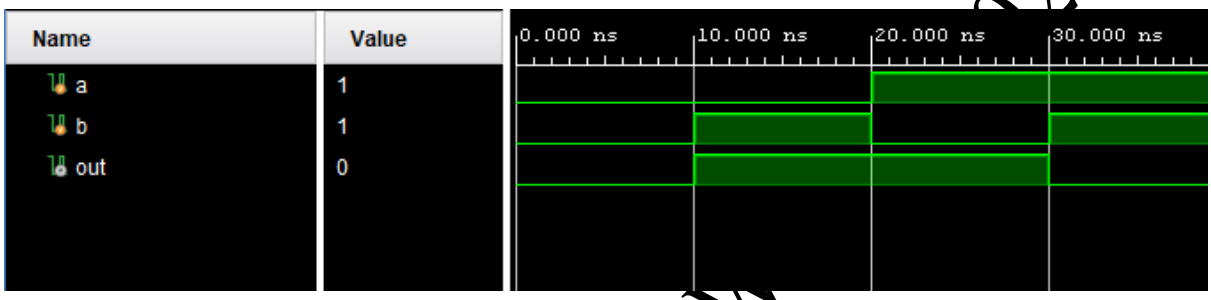
6. XOR Gate

```
module gate_xor(
input a, b,
output reg out);


always @(*) begin
if ((a==1'b0 && b==1'b0) || (a==1'b1 && b==1'b1))
out = 1'b0;
else
out = 1'b1;
end
endmodule
```

TB

```
module gate_xor_tb;
reg a, b;
wire out;

gate_xor uut (
    .a(a),
    .b(b),
    .out(out)
);
```

```
initial begin
    a = 0; b = 0; #10;
    a = 0; b = 1; #10;
    a = 1; b = 0; #10;
    a = 1; b = 1; #10;
    $finish;
end
endmodule
```



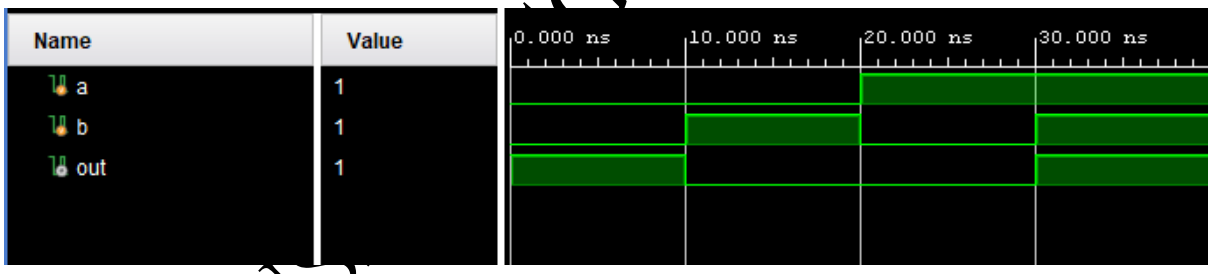
7. XNOR Gate

```
module gate_xnor(
input a, b,
output reg out);

always @(*) begin
if ((a==1'b0 && b==1'b0) || (a==1'b1 && b==1'b1))
out = 1'b1;
else
out = 1'b0;
end
endmodule
```

TB

```
module gate_xnor_tb;
reg a, b;
wire out;
```

```
gate_xnor uut (
    .a(a),
    .b(b),
    .out(out)
);

initial begin
    a = 0; b = 0; #10;
    a = 0; b = 1; #10;
    a = 1; b = 0; #10;
    a = 1; b = 1; #10;
    $finish;
end
endmodule
```



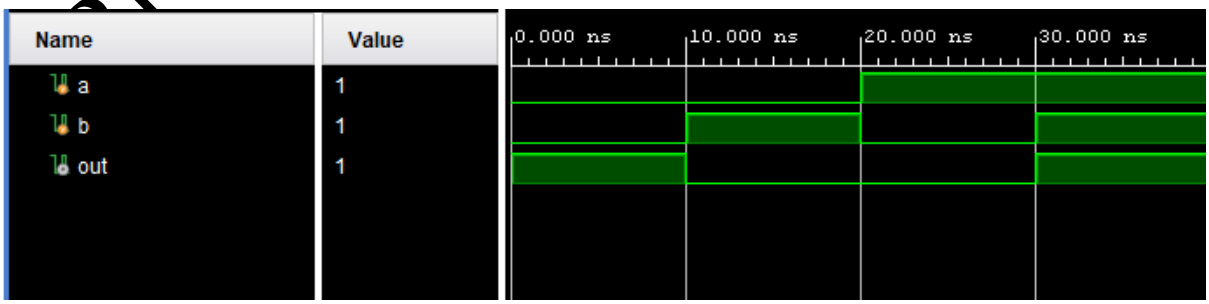
8. Gate Combo

```
module gate_combo(
input a, b,
output out);

wire w0, w1, w2;

assign w0 = a^b;
assign w1 = w0&a;
assign w2 = w1|w0;
```

```
assign out = ~w2;


endmodule
```

TB

```
module gate_combo_tb;
reg a, b;
wire out;

gate_combo uut (
    .a(a),
    .b(b),
    .out(out)
);

initial begin
    a = 0; b = 0; #10;
    a = 0; b = 1; #10;
    a = 1; b = 0; #10;
    a = 1; b = 1; #10;
    $finish;
end
endmodule
```



9. 4 input AND Gate

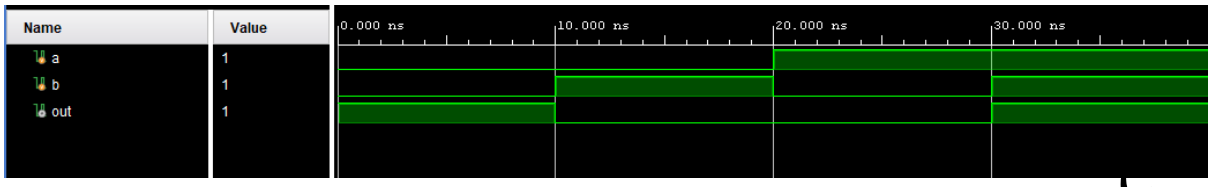
```
module and4_gate(
```

```
input a, b, c, d,
output reg out);
always @(*) begin
if (a==1'b1 && b==1'b1 && c==1'b1 && d==1'b1)
out = 1'b1;
else
out = 1'b0;
end
endmodule
```

TB

```
module and4_gate_tb;
reg a, b, c, d;
wire out;

and4_gate uut (
    .a(a),
    .b(b),
    .c(c),
    .d(d),
    .out(out)
);

initial begin
    a = 0; b = 0; c = 0; d = 0; #10;
    a = 0; b = 1; c = 1; d = 1; #10;
    a = 1; b = 0; c = 1; d = 1; #10;
    a = 1; b = 1; c = 0; d = 1; #10;
    a = 1; b = 1; c = 1; d = 0; #10;
    a = 1; b = 1; c = 1; d = 1; #10;
```

```
    $finish;
end
endmodule
```



10. 4 Input OR Gate

```
module gate_4or(
input a, b, c, d,
output reg out);

always @(*) begin
if (a==1'b0 && b==1'b0 && c==1'b0 && d==1'b0)
out = 1'b0;
else
out = 1'b1;
end
endmodule
```

TB

```
module gate_4or_tb;
reg a, b, c, d;
wire out;

gate_4or uut (
    .a(a),
    .b(b),
    .c(c),
    .d(d),
    .out(out)
```

);

initial begin

a = 0; b = 0; c = 0; d = 0; #10;

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

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

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

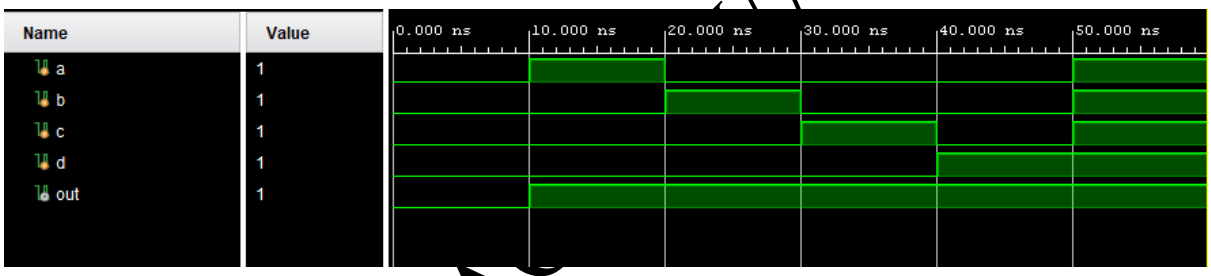
a = 0; b = 0; c = 0; d = 1; #10;

a = 1; b = 1; c = 1; d = 1; #10;

$finish;

end

endmodule



11. Half Adder

module ha(

input a, b,

output sum, carry);

assign sum = a^b;

assign carry = a&b;

endmodule

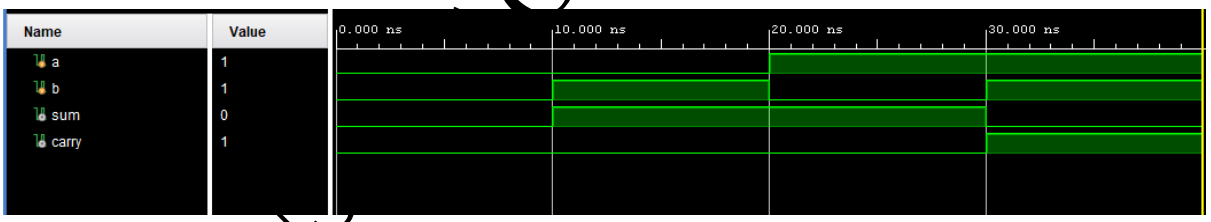
TB

module ha_tb;

reg a, b;

wire sum, carry;

```
ha uut (
    .a(a),
    .b(b),
    .sum(sum),
    .carry(carry)
);

initial begin
    a = 0; b = 0; #10;
    a = 0; b = 1; #10;
    a = 1; b = 0; #10;
    a = 1; b = 1; #10;
    $finish;
end
endmodule
```

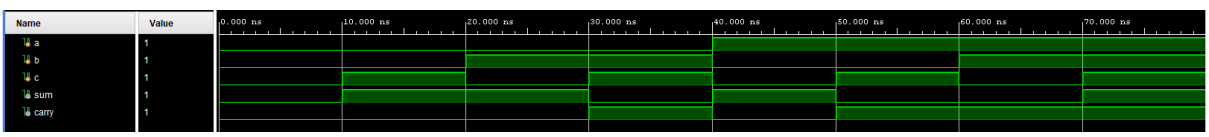


12. Full Adder

```
module fa(
input a, b, c
output sum, carry);

assign sum = a^b^c;
assign carry = (a&b) | (b&c) | (c&a);

endmodule
```

TB

```
module fa_tb;
reg a, b, c;
wire sum, carry;

fa uut (
    .a(a),
    .b(b),
    .c(c),
    .sum(sum),
    .carry(carry)
);

initial begin
    a = 0; b = 0; c = 0; #10;
    a = 0; b = 0; c = 1; #10;
    a = 0; b = 1; c = 0; #10;
    a = 0; b = 1; c = 1; #10;
    a = 1; b = 0; c = 0; #10;
    a = 1; b = 0; c = 1; #10;
    a = 1; b = 1; c = 0; #10;
    a = 1; b = 1; c = 1; #10;
    $finish;
end
endmodule
```

13. 4 Bit Ripple Carry Adder

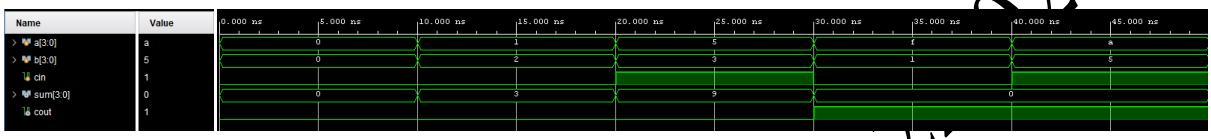
```
module rca4(
```

```
input [3:0] a, b,
input cin,
output [3:0] sum,
output cout);

wire [2:0] w;

fa u0(a[0], b[0], cin, sum[0], w[0]);
fa u1(a[1], b[1], w[0], sum[1], w[1]);
fa u2(a[2], b[2], w[1], sum[2], w[2]);
fa u3(a[3], b[3], w[2], sum[3], cout);

endmodule
```

TB

```
module rca4_tb;
reg [3:0] a, b;
reg cin;
wire [3:0] sum;
wire cout;

rca4 uut (
    .a(a),
    .b(b),
    .cin(cin),
    .sum(sum),
    .cout(cout)
);

initial begin
```

```
    a = 4'b0000; b = 4'b0000; cin = 0; #10;
    a = 4'b0001; b = 4'b0010; cin = 0; #10;
    a = 4'b0101; b = 4'b0011; cin = 1; #10;
    a = 4'b1111; b = 4'b0001; cin = 0; #10;
    a = 4'b1010; b = 4'b0101; cin = 1; #10;
    $finish;
end
endmodule
```



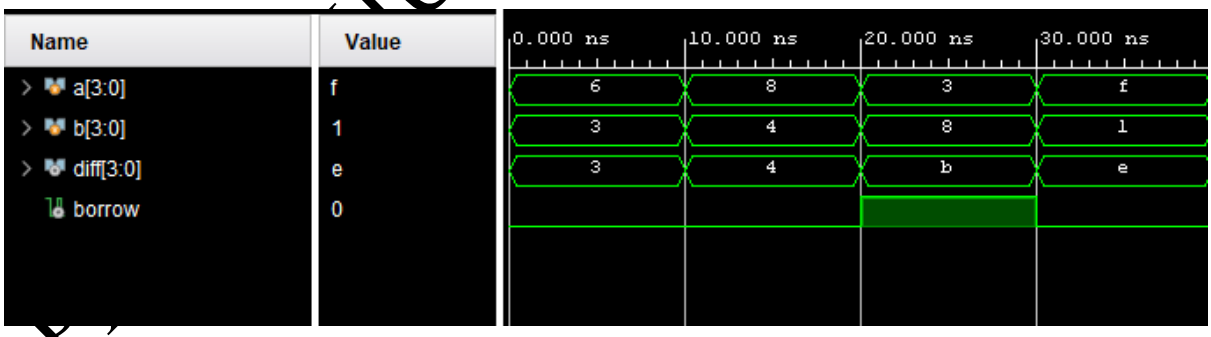
14. 4 Bit Subtractor

```
module sub4(
    input [3:0] a, b,
    output [3:0] diff,
    output borrow
);

wire [4:0] temp;

assign temp = {1'b0, a} - {1'b0, b};
assign diff = temp[3:0];
assign borrow = temp[4];

endmodule
```

TB

```
module sub4TB;

reg [3:0] a, b;
wire [3:0] diff;
```

```
wire borrow;

sub4 uut (
    .a(a),
    .b(b),
    .diff(diff),
    .borrow(borrow)
);

initial begin
    a = 4'b0110; b = 4'b0011; #10;
    a = 4'b1000; b = 4'b0100; #10;
    a = 4'b0011; b = 4'b1000; #10;
    a = 4'b1111; b = 4'b0001; #10;
    $finish;
end

endmodule
```



15. 4 Bit ADDER Subtractor

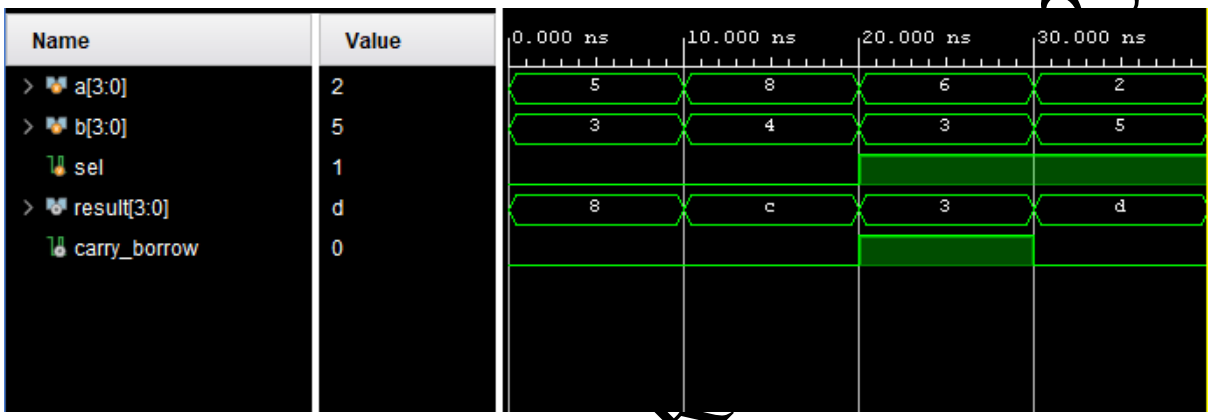
```
module add_sub4(
    input [3:0] a, b,
    input sel,              // sel = 0 for addition, sel = 1 for subtraction
    output [3:0] result,
```

```
    output carry_borrow
);

wire [3:0] b_xor;
wire cin;
assign b_xor = b ^ {4{sel}};  // XOR b with sel: if sel=1, invert b (2's comp)
assign cin = sel;

assign {carry_borrow, result} = a + b_xor + cin;

endmodule
```

TB

```
module add_sub4TB;

reg [3:0] a, b;
reg sel;
wire [3:0] result;
wire carry_borrow;

add_sub4 uut (
    .a(a),
    .b(b),
    .sel(sel),
    .result(result),
    .carry_borrow(carry_borrow)
);

initial begin
    a = 4'b0101; b = 4'b0011; sel = 0; #10;  // 5 + 3
```

```
    a = 4'b1000; b = 4'b0100; sel = 0; #10;  // 8 + 4
    a = 4'b0110; b = 4'b0011; sel = 1; #10;  // 6 - 3
    a = 4'b0010; b = 4'b0101; sel = 1; #10;  // 2 - 5
    $finish;
end

endmodule
```



16. Mux 2X1

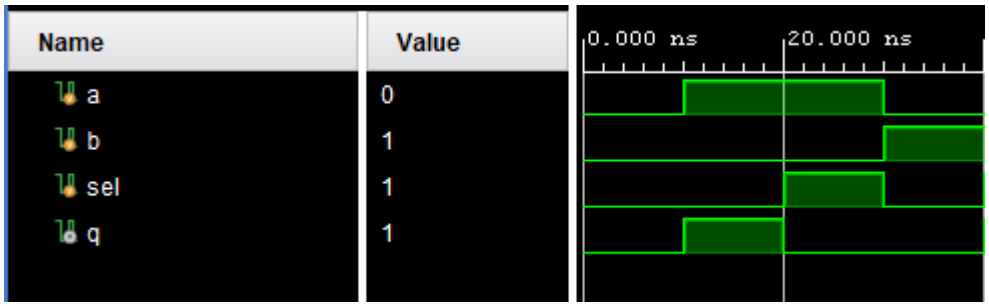
```
module mux_2X1(
input a, b,sel,
output q
);

assign q = (sel) ? b : a;

endmodule
```

TB

```
module mux_2X1_TB();
reg a, b, sel;
wire q;

mux_2X1 dut(a, b, sel ,q);
```

```
initial begin
a = 0;
b = 0;
sel = 0;
#10;

a = 1;
b = 0;
sel = 0;
#10;

a = 1;
b = 0;
sel = 1;
#10;

a = 0;
b = 1;
sel = 0;
#10;

a = 0;
b = 1;
sel = 1;
#10;

end
```

endmodule



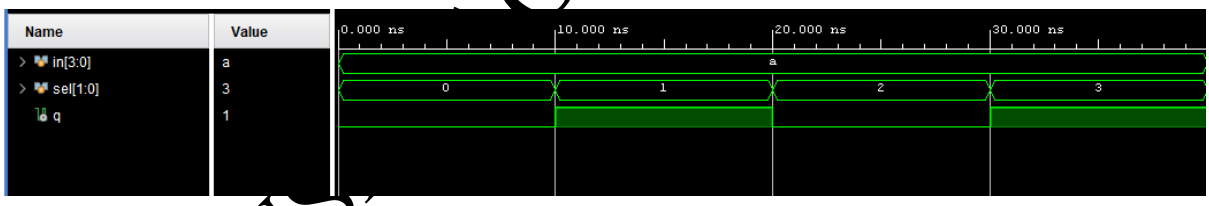
17. MUX 4X1

```
module mux_4X1(
input [3:0] in,
input [1:0] sel,
output reg q);

always @(*) begin
case (sel)
2'b00 : q = in[0];
2'b01 : q = in[1];
2'b10 : q = in[2];
2'b11 : q = in[3];

endcase
end
endmodule
```

TB

```
module mux_4X1_tb;
reg [3:0] in;
reg [1:0] sel;
wire q;

mux_4X1 uut (
```

```
    .in(in),
    .sel(sel),
    .q(q)
);

initial begin
    in = 4'b1010;

    sel = 2'b00; #10;
    sel = 2'b01; #10;
    sel = 2'b10; #10;
    sel = 2'b11; #10;

    $finish;
end
endmodule
```



18. MUX 8X1

```
module mux_8X1(
input [7:0] in,
input [2:0] sel,
output reg q);

always @(*) begin

case (sel)
3'b000 : q = in[0];
```

3'b001 : q = in[1];

3'b010 : q = in[2];

3'b011 : q = in[3];

3'b100 : q = in[4];

3'b101 : q = in[5];

3'b110 : q = in[6];

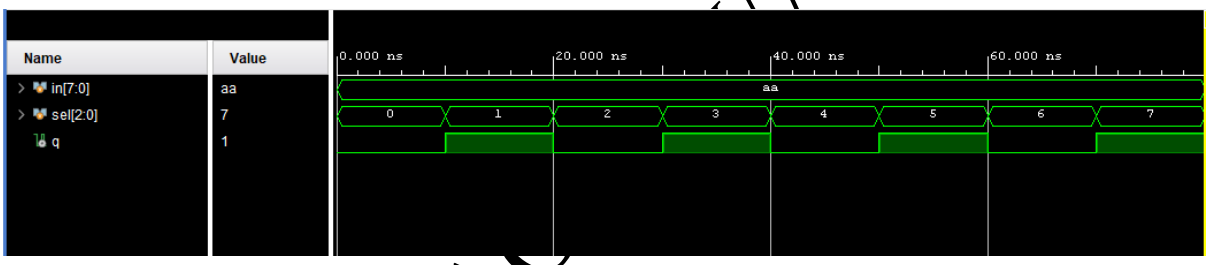
3'b111 : q = in[7];

endcase

end

endmodule

TB



19. DEMUX 1X2

module demux_1X2(

input a, sel,

output [1:0] out);

assign out = a << sel;

endmodule

TB

module demux_1X2_tb;

reg a, sel;

wire [1:0] out;

```
demux_1X2 uut (
    .a(a),
    .sel(sel),
    .out(out)
);

initial begin
    a = 0; sel = 0; #10;
    a = 1; sel = 0; #10;
    a = 1; sel = 1; #10;
    a = 0; sel = 1; #10;
    $finish;
end
endmodule
```

20. DEMUX 1X4

```
module demux_1X4(
input a,
input [1:0] sel,
output reg [3:0] q);

//assign q = a << sel;
always @(*) begin
case (sel)
2'b00 : q[0] = a;
```

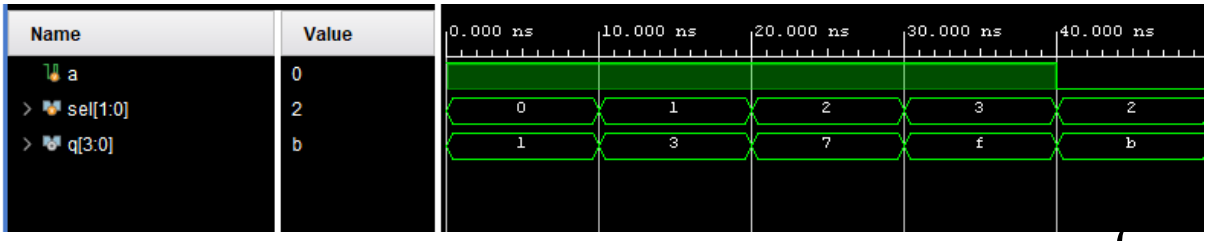
```
2'b01 : q[1] = a;
2'b10 : q[2] = a;
2'b11 : q[3] = a;
endcase
end
endmodule
```

TB

```
module demux_1X4_tb;
reg a;
reg [1:0] sel;
wire [3:0] q;

demux_1X4 uut (
    .a(a),
    .sel(sel),
    .q(q)
);

initial begin
    a = 1;

    sel = 2'b00; #10;
    sel = 2'b01; #10;
    sel = 2'b10; #10;
    sel = 2'b11; #10;

    a = 0; sel = 2'b10; #10;

    $finish;
```

end

endmodule



21. Decoder 2X4

module decod_2X4(

input [1:0] a,

output reg [3:0] out);

always @(*) begin

case (a)

2'b00 : out[0] = 1'b1;

2'b01 : out[1] = 1'b1;

2'b10 : out[2] = 1'b1;

2'b11 : out[3] = 1'b1;

endcase

end

endmodule

TB

module decod_2X4_tb;

reg [1:0] a;

wire [3:0] out;

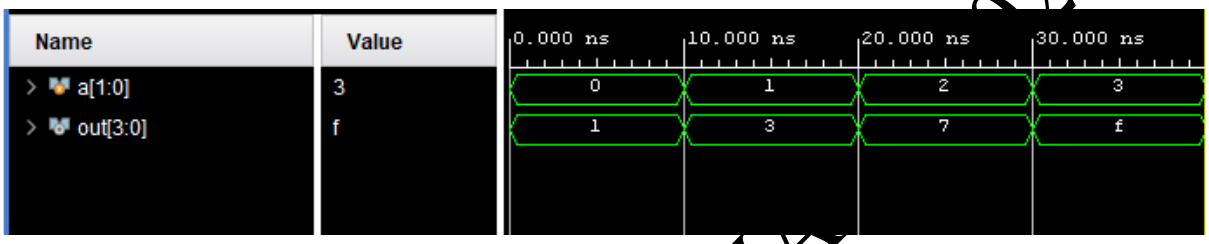
decod_2X4 uut (

.a(a),

.out(out)

);

```
initial begin
    a = 2'b00; #10;
    a = 2'b01; #10;
    a = 2'b10; #10;
    a = 2'b11; #10;
    $finish;
end
endmodule
```



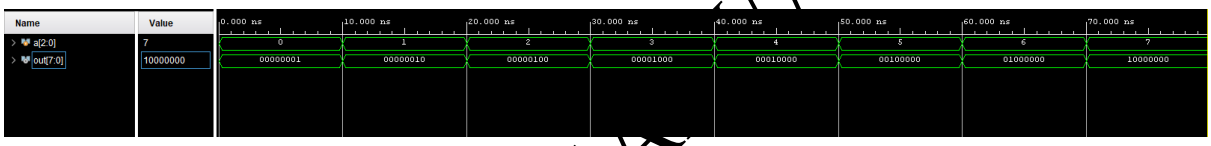
22. Decoder 3X8

```
module decod_3X8(
input [2:0] a,
output  [7:0] out);

assign out = 8'b00000001 << a;
endmodule
```

TB

```
module decod_3X8_tb;
reg [2:0] a;
wire [7:0] out;

decod_3X8 uut (
    .a(a),
    .out(out)
);
```

```
initial begin
    a = 3'b000; #10;
    a = 3'b001; #10;
    a = 3'b010; #10;
    a = 3'b011; #10;
    a = 3'b100; #10;
    a = 3'b101; #10;
    a = 3'b110; #10;
    a = 3'b111; #10;
    $finish;
end
endmodule
```



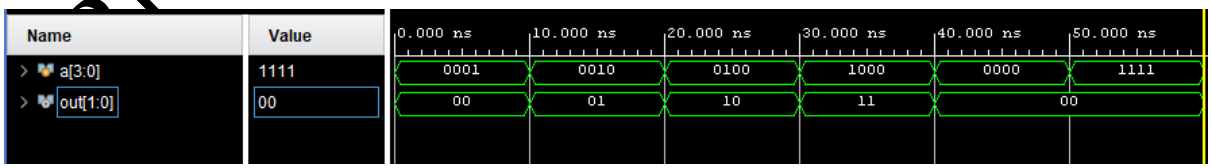
23. Encoder 4X2

```
module encod_4X2(
input [3:0] a,
output reg [1:0] out);

always @(*) begin
case (a)

4'b0001 : out = 2'b00;
4'b0010 : out = 2'b01;
4'b0100 : out = 2'b10;
4'b1000 : out = 2'b11;
default : out = 2'b00;

endcase
```

```
end
endmodule
```

TB

```
module encod_4X2_tb;
reg [3:0] a;
wire [1:0] out;

encod_4X2 uut (
    .a(a),
    .out(out)
);

initial begin
    a = 4'b0001; #10;
    a = 4'b0010; #10;
    a = 4'b0100; #10;
    a = 4'b1000; #10;
    a = 4'b0000; #10;
    a = 4'b1111; #10;
    $finish;
end
endmodule
```

24. Encoder 8X3

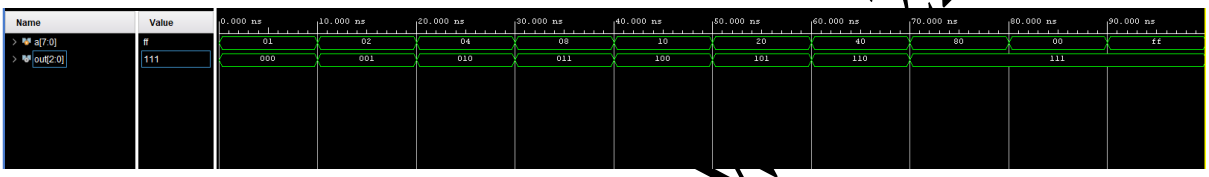
```
module encod_8X3(
input [7:0] a,
output reg [2:0] out);
```

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

TB

```
module encod_8X3_tb;
reg [7:0] a;
wire [2:0] out;

encod_8X3 uut (
    .a(a),
    .out(out)
);

initial begin
    a = 8'b00000001; #10;
    a = 8'b00000010; #10;
    a = 8'b00000100; #10;
```

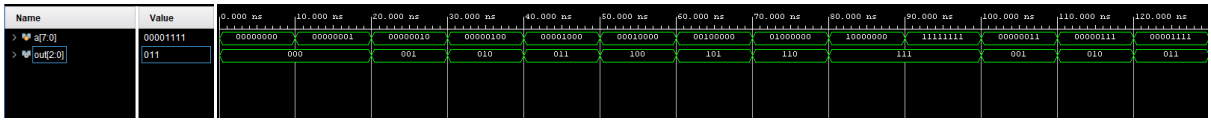
```
    a = 8'b00001000; #10;
    a = 8'b00010000; #10;
    a = 8'b00100000; #10;
    a = 8'b01000000; #10;
    a = 8'b10000000; #10;
    a = 8'b00000000; #10;
    a = 8'b11111111; #10;
    $finish;
end
endmodule
```



25. Priority Encoder 8X3

```
module pencod_8X3(
input [7:0] a,
output reg [2:0] out);

always @(*) begin
casez (a)
8'b00000000 : out = 3'b000;
8'b00000001 : out = 3'b000;
8'b0000001? : out = 3'b001;
8'b000001?? : out = 3'b010;
8'b00001??? : out = 3'b011;
8'b0001???? : out = 3'b100;
8'b001????? : out = 3'b101;
8'b01?????? : out = 3'b110;
8'b1??????? : out = 3'b111;
```

```
endcase
end
endmodule
TB
module pencod_8X3_tb;
reg [7:0] a;
wire [2:0] out;

pencod_8X3 uut(a, out);

initial begin
  a = 8'b00000000; #10;
  a = 8'b00000001; #10;
  a = 8'b00000010; #10;
  a = 8'b00000100; #10;
  a = 8'b00001000; #10;
  a = 8'b00010000; #10;
  a = 8'b00100000; #10;
  a = 8'b01000000; #10;
  a = 8'b10000000; #10;
  a = 8'b11111111; #10;
  a = 8'b00000011; #10;
  a = 8'b00000111; #10;
  a = 8'b00001111; #10;
  $finish;
end
endmodule
```



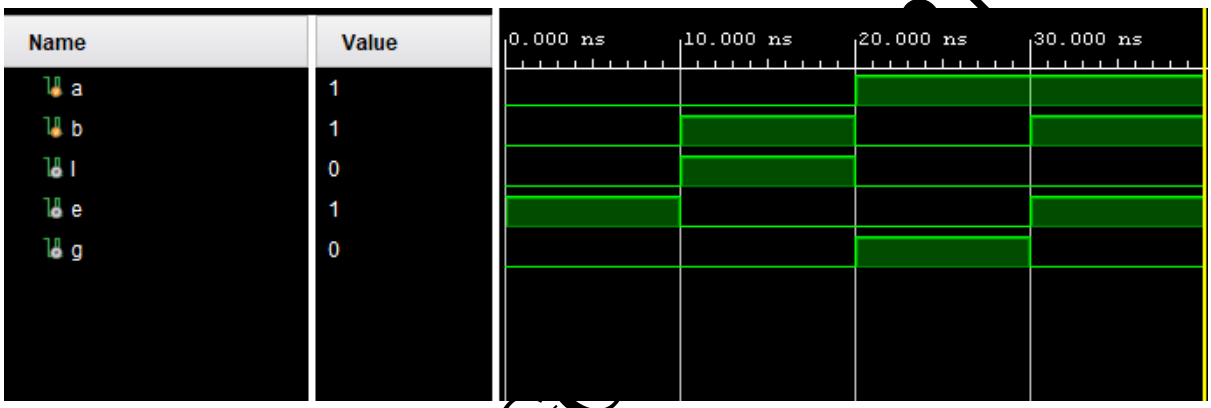
26. 1 Bit Comparator

```
module comp1(
input a, b,
output reg l ,e, g);
initial begin
e = 0;
l = 0;
g = 0;
end
always @(*) begin
if (a > b) begin
g = 1'b1; l = 0; e = 0;
end
else if ( a < b) begin
l = 1; g =0; e = 0;
end
else
e = 1; g = 0; l = 0;
end

endmodule
```

TB

```
module comp1_tb;
reg a, b;
wire l, e, g;

comp1 uut(a, b, l, e, g);
```

```
initial begin
  a = 0; b = 0; #10;
  a = 0; b = 1; #10;
  a = 1; b = 0; #10;
  a = 1; b = 1; #10;
  $finish;
end
endmodule
```



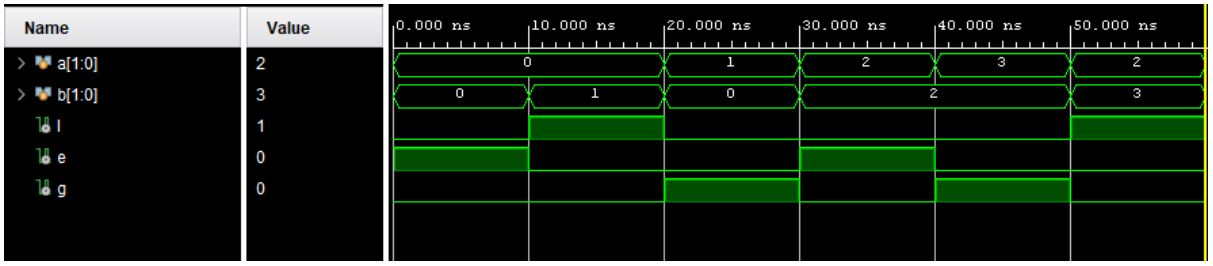
27. 2 Bit Comparator

```
module comp2(
input [1:0] a,b,
output reg l,e,g );


always @(*) begin
if (a>b) begin
g = 1;
l = 0;
e = 0;
end
else if (a<b) begin
l = 1;
```

```
g = 0;
e = 0;
end
else begin
e = 1;
l = 0;
g = 0;
end
end

endmodule
```

TB

```
module comp2_tb;
reg [1:0] a, b;
wire l, e, g;

comp2 uut(a, b, l, e, g);

initial begin
  a = 2'b00; b = 2'b00; #10;
  a = 2'b00; b = 2'b01; #10;
  a = 2'b01; b = 2'b00; #10;
  a = 2'b10; b = 2'b10; #10;
  a = 2'b11; b = 2'b10; #10;
  a = 2'b10; b = 2'b11; #10;
  $finish;
end
endmodule
```

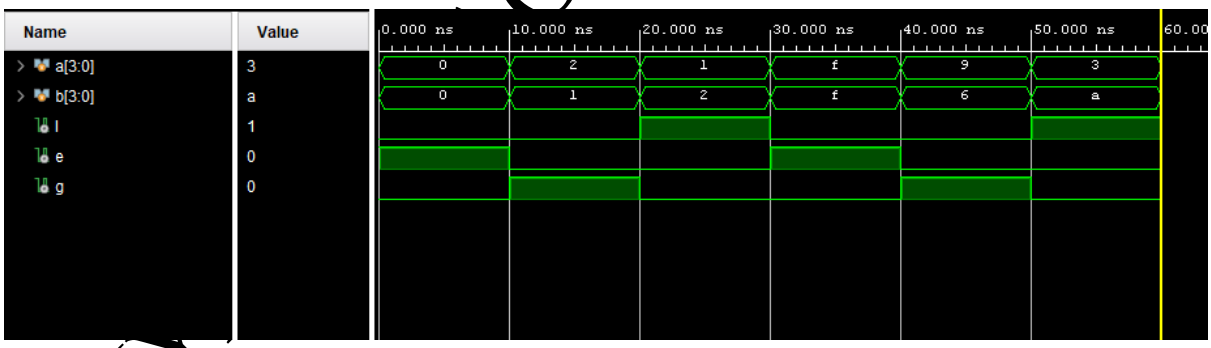


28. 4 Bit Comparator

```
module comp4(
input [3:0] a,b,
output reg l,e,g );


always @(*) begin
if (a>b) begin
g = 1;
l = 0;
e = 0;
end
else if (a<b) begin
l = 1;
g = 0;
e = 0;
end
else begin
e = 1;
l = 0;
g = 0;
end
end


endmodule
```

TB

```
module comp4_tb;
reg [3:0] a, b;
wire l, e, g;

comp4 uut(a, b, l, e, g);

initial begin
  a = 4'b0000; b = 4'b0000; #10;
  a = 4'b0010; b = 4'b0001; #10;
  a = 4'b0001; b = 4'b0010; #10;
  a = 4'b1111; b = 4'b1111; #10;
  a = 4'b1001; b = 4'b0110; #10;
  a = 4'b0011; b = 4'b1010; #10;
  $finish;
end
endmodule
```



29. 8 Bit Comparator

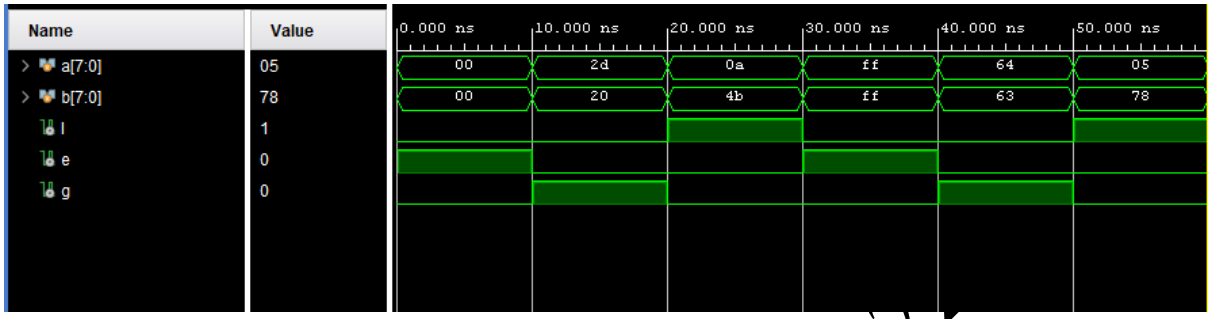
```
module comp8(
input [7:0] a,b,
output reg l,e,g );

always @(*) begin
if (a>b) begin
```

```
g = 1;
l = 0;
e = 0;
end
else if (a<b) begin
l = 1;
g = 0;
e = 0;
end
else begin
e = 1;
l = 0;
g = 0;
end
end

endmodule
```

TB

```
module comp8_tb;
reg [7:0] a, b;
wire l, e, g;

comp8 uut(a, b, l, e, g);

initial begin
  a = 8'd0;   b = 8'd0;   #10;
  a = 8'd45;  b = 8'd32;  #10;
  a = 8'd10;  b = 8'd75;  #10;
  a = 8'd255; b = 8'd255; #10;
```

```
  a = 8'd100; b = 8'd99;  #10;
  a = 8'd5;   b = 8'd120; #10;
  $finish;
end
endmodule
```



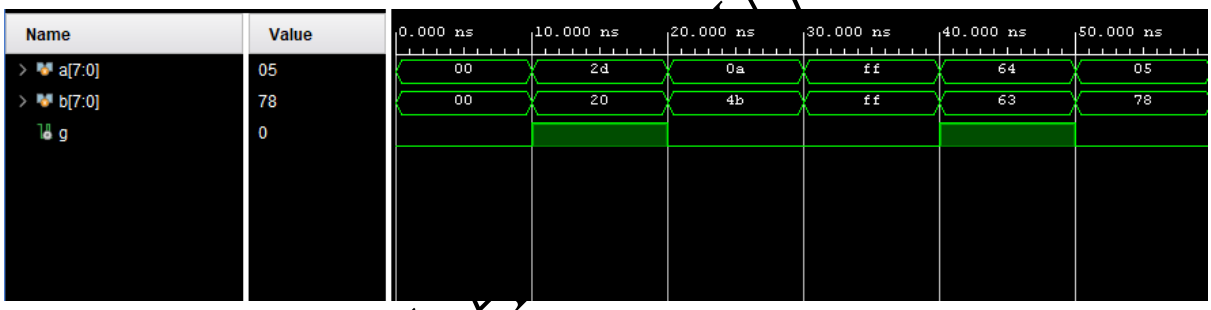
30. Greater Than Comparator

```
module compgt(
input [7:0] a,b,
output reg g );

always @(*) begin
if (a>b)
g = 1;

else
g = 0;
end
endmodule
```

TB

```
module compgt_tb;
reg [7:0] a, b;
wire l, e, g;
```

```
compgt uut(a, b, l, e, g);


initial begin
  a = 8'd0;   b = 8'd0;   #10;
  a = 8'd45;  b = 8'd32;  #10;
  a = 8'd10;  b = 8'd75;  #10;
  a = 8'd255; b = 8'd255; #10;
  a = 8'd100; b = 8'd99;  #10;
  a = 8'd5;   b = 8'd120; #10;
  $finish;
end
endmodule
```



31. 4 Bit Carry Look Ahead Adder

```
module cla4 (
    input [3:0] a, b,
    input cin,
    output [3:0] sum,
    output cout
);
    wire [3:0] g, p, c;

    assign g = a & b;
    assign p = a ^ b;
```

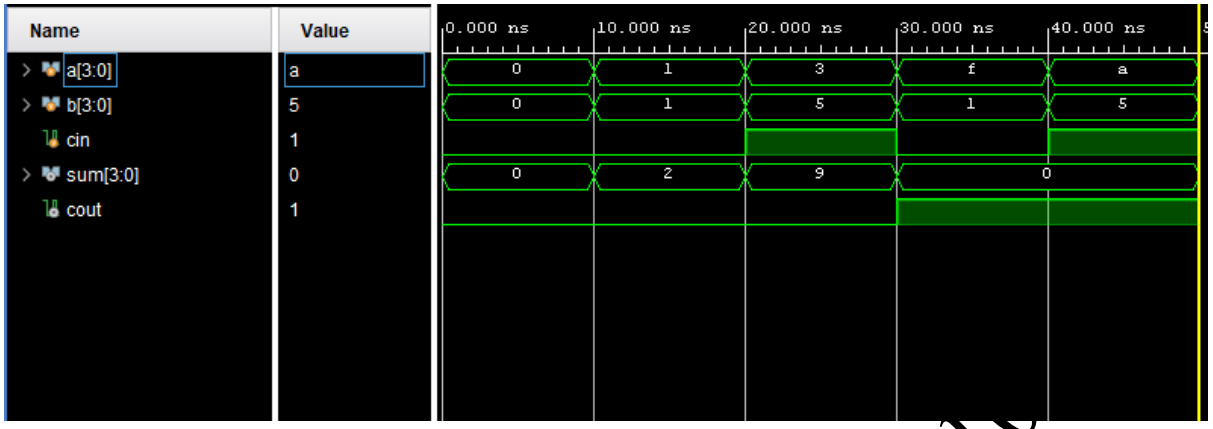
```
    assign c[0] = cin;
    assign c[1] = g[0] | (p[0] & c[0]);
    assign c[2] = g[1] | (p[1] & g[0]) | (p[1] & p[0] & c[0]);
    assign c[3] = g[2] | (p[2] & g[1]) | (p[2] & p[1] & g[0]) | (p[2] & p[1] & p[0] & c[0]);

    assign sum = p ^ c;
    assign cout = g[3] | (p[3] & g[2]) | (p[3] & p[2] & g[1]) |
                  (p[3] & p[2] & p[1] & g[0]) |
                  (p[3] & p[2] & p[1] & p[0] & cin);
Endmodule
```

TB

```
module cla4_TB;
    reg [3:0] a, b;
    reg cin;
    wire [3:0] sum;
    wire cout;

    cla4 uut (
        .a(a), .b(b), .cin(cin),
        .sum(sum), .cout(cout)
    );

    initial begin
        a = 4'b0000; b = 4'b0000; cin = 0;
        #10 a = 4'b0001; b = 4'b0001; cin = 0;
        #10 a = 4'b0011; b = 4'b0101; cin = 1;
        #10 a = 4'b1111; b = 4'b0001; cin = 0;
        #10 a = 4'b1010; b = 4'b0101; cin = 1;
        #10 $finish;
```

end

endmodule



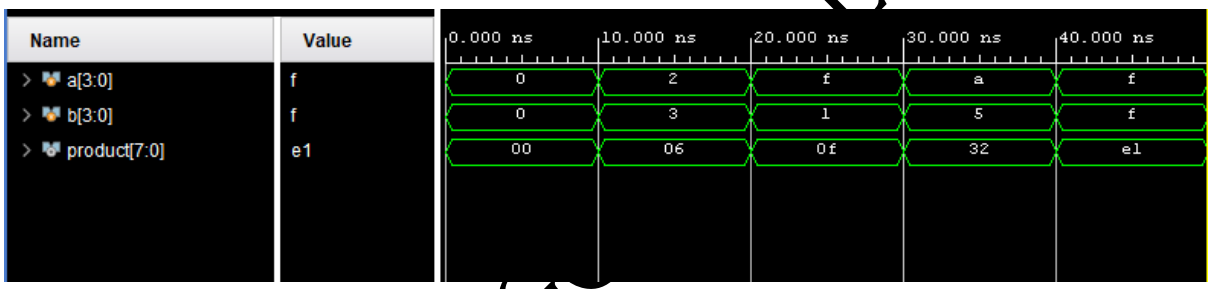
32. 4 Bit Binary Multiplier

```
module mult4 (
    input [3:0] a, b,
    output [7:0] product
);
    wire [7:0] p0 = b[0] ? {4'b0000, a}       : 8'b0;
    wire [7:0] p1 = b[1] ? {3'b000, a, 1'b0}  : 8'b0;
    wire [7:0] p2 = b[2] ? {2'b00, a, 2'b00}  : 8'b0;
    wire [7:0] p3 = b[3] ? {1'b0, a, 3'b000}  : 8'b0;

    assign product = p0 + p1 + p2 + p3;
endmodule
```

TB

```
module mult4_TB;
    reg [3:0] a, b;
    wire [7:0] product;

    mult4 uut (
        .a(a), .b(b),
        .product(product)
```

```
    );

    initial begin
        a = 4'b0000; b = 4'b0000;
        #10 a = 4'b0010; b = 4'b0011;
        #10 a = 4'b1111; b = 4'b0001;
        #10 a = 4'b1010; b = 4'b0101;
        #10 a = 4'b1111; b = 4'b1111;
        #10 $finish;
    end
endmodule
```



33. Booth Multiplier

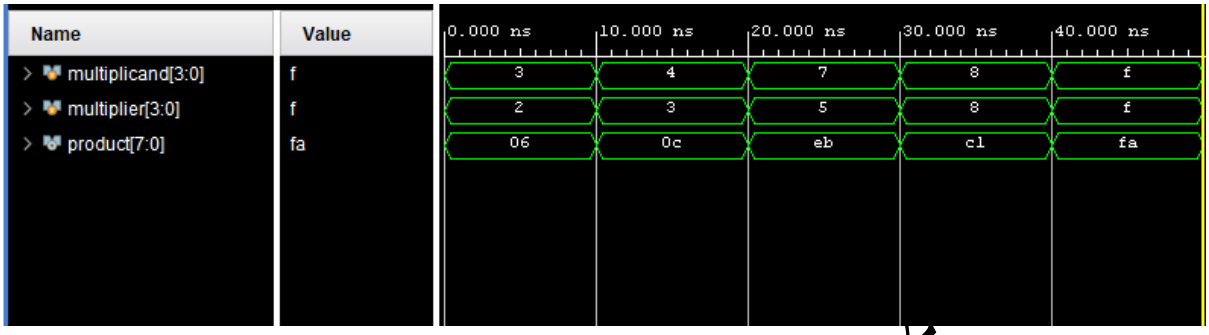
```
module booth_multiplier (
    input  [3:0] multiplicand,
    input  [3:0] multiplier,
    output [7:0] product
);
    reg [7:0] A, S, P;
    reg [3:0] M, Q;
    integer i;

    always @(*) begin
        A = {multiplicand, 4'b0000};
        S = {~multiplicand + 1'b1, 4'b0000};
```

```
        P = {4'b0000, multiplier, 1'b0};

        for (i = 0; i < 4; i = i + 1) begin
            case (P[1:0])
                2'b01: P = P + A;
                2'b10: P = P + S;
                default: P = P;
            endcase
            P = {P[7], P[7:1]}; // arithmetic right shift
        end
    end

    assign product = P[7:0];
endmodule
```

TB

```
module booth_multiplier_TB;
    reg  [3:0] multiplicand, multiplier;
    wire [7:0] product;

    booth_multiplier uut (
        .multiplicand(multiplicand),
        .multiplier(multiplier),
        .product(product)
    );

    initial begin
        multiplicand = 4'd3;  multiplier = 4'd2;  #10;
        multiplicand = 4'd4;  multiplier = 4'd3;  #10;
        multiplicand = 4'd7;  multiplier = 4'd5;  #10;
```

```
        multiplicand = 4'd8;  multiplier = 4'd8;  #10;
        multiplicand = 4'd15; multiplier = 4'd15; #10;
        $finish;
    end
endmodule
```



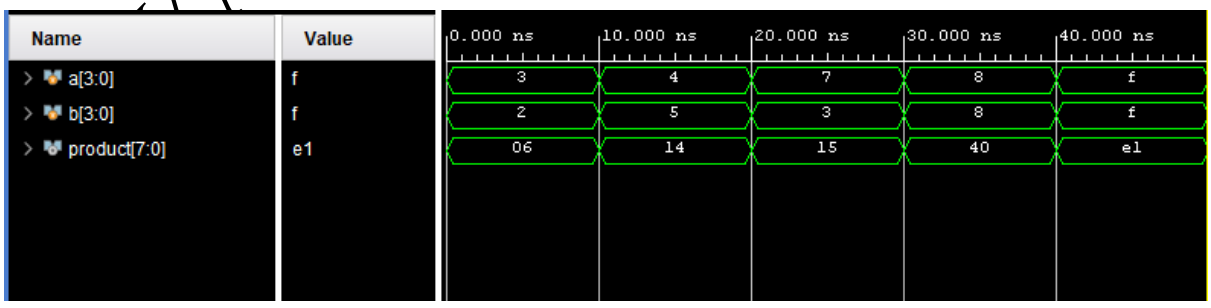
34. Array Multiplier

```
module array_multiplier (
    input  [3:0] a,
    input  [3:0] b,
    output [7:0] product
);
    wire [3:0] p0, p1, p2, p3;
    wire [7:0] s1, s2, s3;

    assign p0 = a & {4{b[0]}};
    assign p1 = a & {4{b[1]}};
    assign p2 = a & {4{b[2]}};
    assign p3 = a & {4{b[3]}};

    assign s1 = {1'b0, p1, 1'b0}; // shifted left by 1
    assign s2 = {2'b0, p2, 2'b0}; // shifted left by 2
    assign s3 = {3'b0, p3, 3'b0}; // shifted left by 3

    assign product = {4'b0000, p0} + s1 + s2 + s3;
```

```
endmodule
```

TB

```
module array_multiplier_TB;
    reg  [3:0] a, b;
    wire [7:0] product;

    array_multiplier uut (
        .a(a),
        .b(b),
        .product(product)
    );

    initial begin
        a = 4'd3;  b = 4'd2;  #10;
        a = 4'd4;  b = 4'd5;  #10;
        a = 4'd7;  b = 4'd3;  #10;
        a = 4'd8;  b = 4'd8;  #10;
        a = 4'd15; b = 4'd15; #10;
        $finish;
    end
endmodule
```

35. Divider using Repeated Subtraction

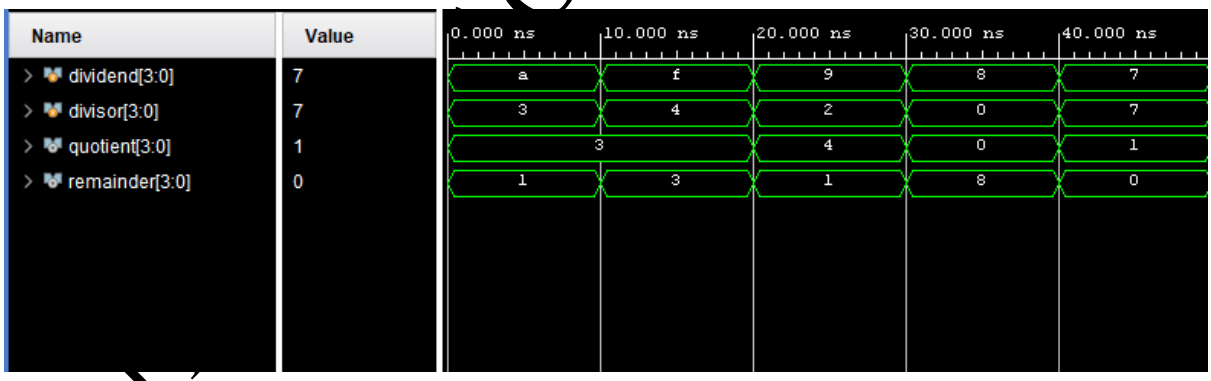
```
module divider_subtract (
    input      [3:0] dividend,
```

```
    input      [3:0] divisor,
    output reg [3:0] quotient,
    output reg [3:0] remainder
);
    integer i;
    reg [3:0] temp_dividend;

    always @(*) begin
        quotient = 0;
        remainder = 0;
        temp_dividend = dividend;

        if (divisor == 0) begin
            quotient = 4'b0000;
            remainder = dividend;
        end else begin
            while (temp_dividend >= divisor) begin
                temp_dividend = temp_dividend - divisor;
                quotient = quotient + 1;
            end
            remainder = temp_dividend;
        end
    end
endmodule
```

TB

```
module divider_subtract_TB;
    reg  [3:0] dividend, divisor;
    wire [3:0] quotient, remainder;
```

```
    divider_subtract uut (
        .dividend(dividend),
        .divisor(divisor),
        .quotient(quotient),
        .remainder(remainder)
    );

    initial begin
        dividend = 4'd10; divisor = 4'd3; #10;
        dividend = 4'd15; divisor = 4'd4; #10;
        dividend = 4'd9;  divisor = 4'd2; #10;
        dividend = 4'd8;  divisor = 4'd0; #10;
        dividend = 4'd7;  divisor = 4'd7; #10;
        $finish;
    end
endmodule
```



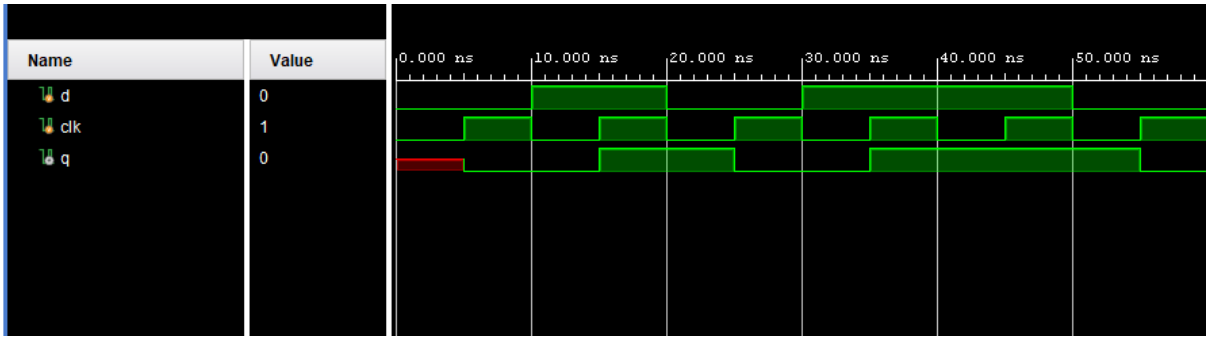
3) D Flip Flop

```
module d_ff (
    input d,
    input clk,
    output reg q
);
    always @(posedge clk)
```

```
        q <= d;
endmodule
```

TB

```
module d_ff_TB;
    reg d, clk;
    wire q;

    d_ff uut (
        .d(d),
        .clk(clk),
        .q(q)
    );

    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    initial begin
        d = 0; #10;
        d = 1; #10;
        d = 0; #10;
        d = 1; #10;
        d = 1; #10;
        d = 0; #10;
        $finish;
    end
endmodule
```



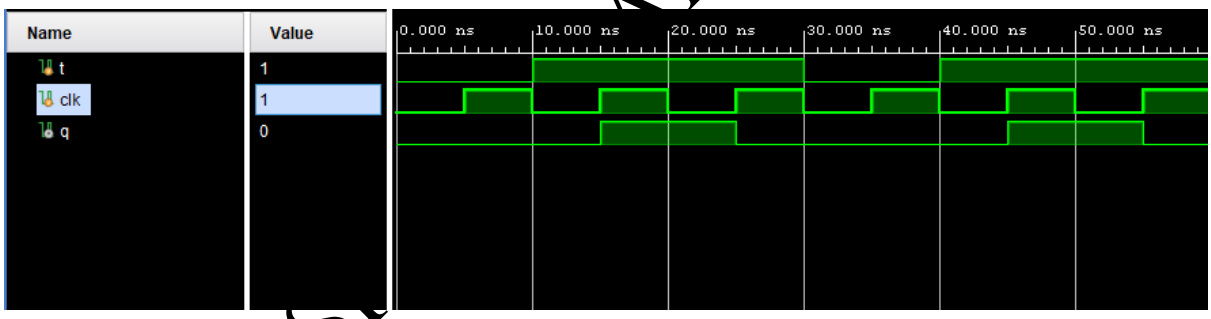
37. T Flip Flop

```
module t_ff (
    input t,
    input clk,
    output reg q
);
    always @(posedge clk) begin
        if (t)
            q <= ~q;
    end
endmodule
```

TB

```
module t_ff_TB;
    reg t, clk;
    wire q;

    t_ff dut (
        .t(t),
        .clk(clk),
        .q(q)
    );

    initial begin
```

```
    clk = 0;
    forever #5 clk = ~clk;
  end

  initial begin
    t = 0; #10;
    t = 1; #10;
    t = 1; #10;
    t = 0; #10;
    t = 1; #10;
    t = 1; #10;
    $finish;
  end
endmodule
```



38. JK Flip Flop

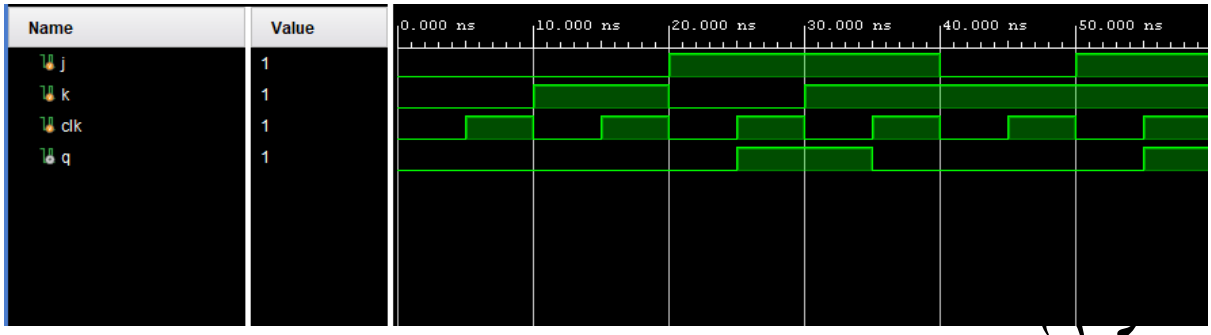
```
module jk_ff(
  input j, k, clk,
  output reg q
);
  always @(posedge clk) begin
    case ({j, k})
      2'b00: q <= q;
      2'b01: q <= 0;
      2'b10: q <= 1;
```

```
            2'b11: q <= ~q;
        endcase
    end
endmodule
```

TB

```
module jk_ff_TB;
    reg j, k, clk;
    wire q;

    jk_ff uut (
        .j(j),
        .k(k),
        .clk(clk),
        .q(q)
    );

    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    initial begin
        j = 0; k = 0; #10;
        j = 0; k = 1; #10;
        j = 1; k = 0; #10;
        j = 1; k = 1; #10;
        j = 0; k = 1; #10;
        j = 1; k = 1; #10;
        $finish;
```

```
    end
endmodule
```



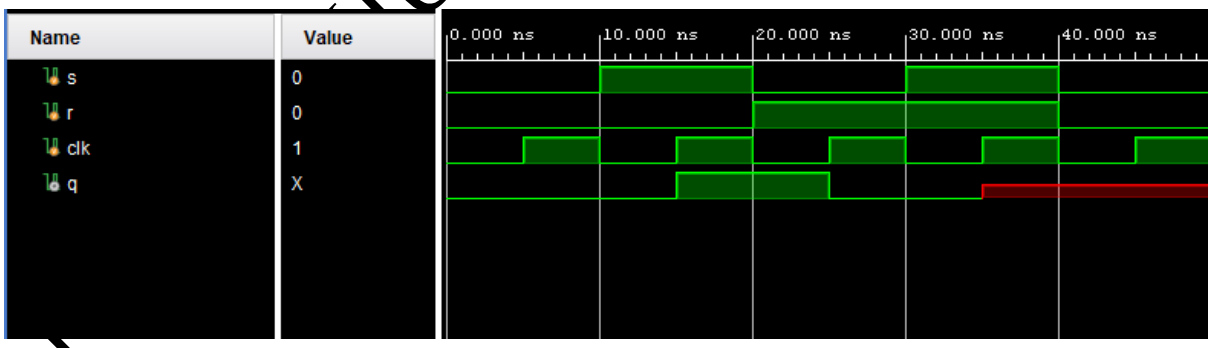
39. SR Flip Flop

```
module sr_ff (
    input s, r, clk,
    output reg q
);
    always @(posedge clk) begin
        case ({s, r})
            2'b00: q <= q;
            2'b01: q <= 0;
            2'b10: q <= 1;
            2'b11: q <= 1'bx; // Invalid condition
        endcase
    end
endmodule
```

TB

```
module sr_ff_TB;
    reg s, r, clk;
    wire q;

    sr_ff uut (
        .s(s),
        .r(r),
```

```
        .clk(clk),
        .q(q)
    );

    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    initial begin
        s = 0; r = 0; #10;
        s = 1; r = 0; #10;
        s = 0; r = 1; #10;
        s = 1; r = 1; #10;
        s = 0; r = 0; #10;
        $finish;
    end
endmodule
```



40. Master-Slave Flip-Flop

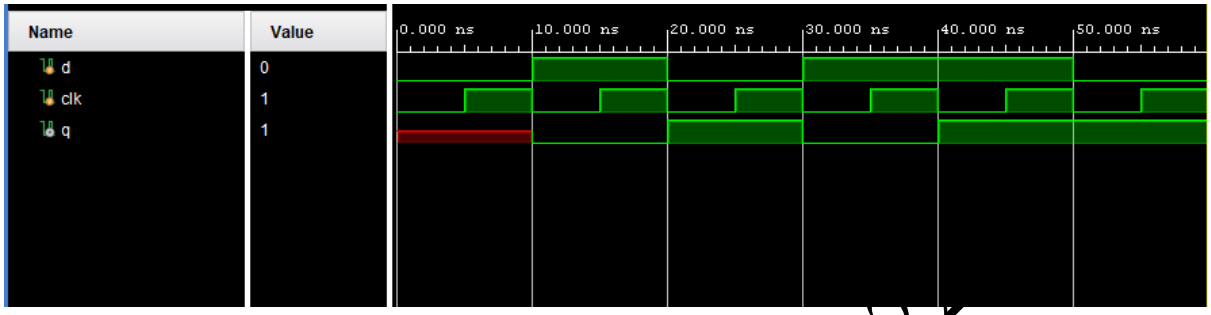
```
module ms_d_ff(
    input d, clk,
    output reg q
);
```

```
    reg master;

    always @(posedge clk)
        master <= d;

    always @(negedge clk)
        q <= master;
endmodule
```

TB

```
module ms_d_ff_TB;
    reg d, clk;
    wire q;

    ms_d_ff uut (
        .d(d),
        .clk(clk),
        .q(q)
    );

    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    initial begin
        d = 0; #10;
        d = 1; #10;
        d = 0; #10;
        d = 1; #10;
```

```
        d = 1; #10;
        d = 0; #10;
        $finish;
    end
endmodule
```



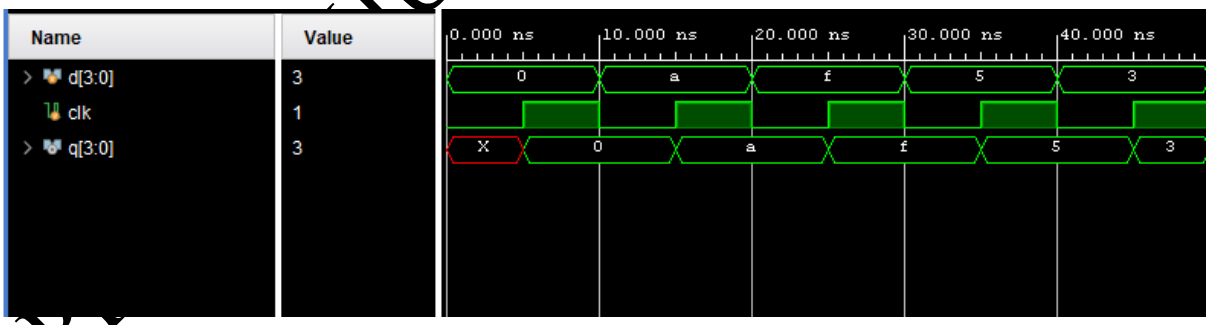
41. 4-bit Register with Positive Clock Edge

```
module reg4bit(
    input [3:0] d,
    input clk,
    output reg [3:0] q
);

always @(posedge clk)
    q <= d;

endmodule
```

TB

```
module reg4bit_TB;
    reg [3:0] d;
    reg clk;
    wire [3:0] q;

    reg4bit uut (
        .d(d),
```

```
    .clk(clk),
    .q(q)
);

initial begin
    clk = 0;
    forever #5 clk = ~clk;
end

initial begin
    d = 4'b0000; #10;
    d = 4'b1010; #10;
    d = 4'b1111; #10;
    d = 4'b0101; #10;
    d = 4'b0011; #10;
    $finish;
end
endmodule
```



42. 4-bit Shift Register (Left)

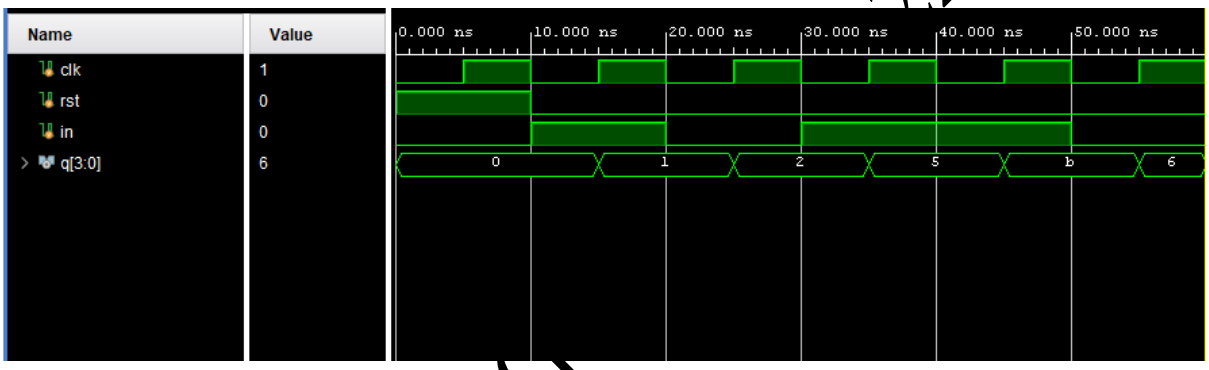
```
module shift_register_left(
    input clk,
    input rst,
    input in,
    output reg [3:0] q
```

```
);

always @(posedge clk or posedge rst) begin
    if (rst)
        q <= 4'b0000;
    else
        q <= {q[2:0], in};
end

endmodule
```

TB

```
module shift_register_left_TB;
    reg clk;
    reg rst;
    reg in;
    wire [3:0] q;

    shift_register_left uut (
        .clk(clk),
        .rst(rst),
        .in(in),
        .q(q)
    );

    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end
```

```
    initial begin
        rst = 1; in = 0; #10;
        rst = 0; in = 1; #10;
        in = 0; #10;
        in = 1; #10;
        in = 1; #10;
        in = 0; #10;
        $finish;
    end
endmodule
```



43. 4-bit Shift Register (Right)

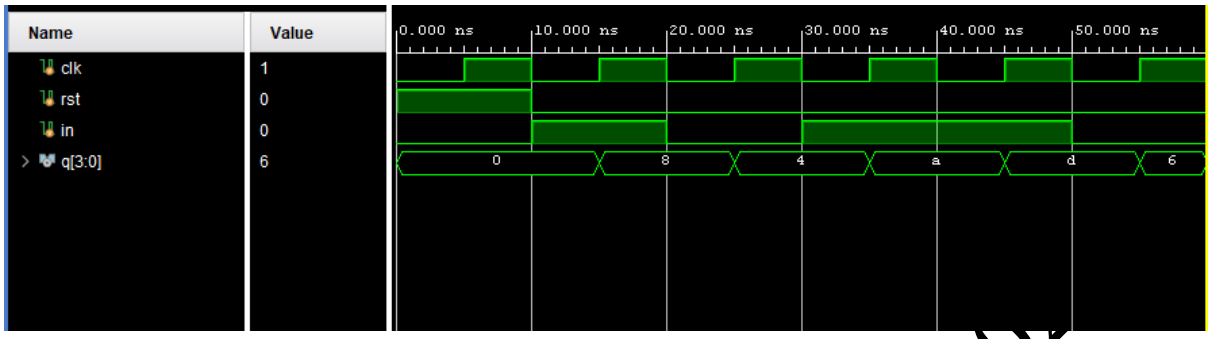
```
module shift_register_right(
    input clk,
    input rst,
    input in,
    output reg [3:0] q
);

always @(posedge clk or posedge rst) begin
    if (rst)
        q <= 4'b0000;
    else
        q <= {in, q[3:1]};
```

```
end


endmodule
```

TB

```
module shift_register_right_TB;
    reg clk;
    reg rst;
    reg in;
    wire [3:0] q;

    shift_register_right uut (
        .clk(clk),
        .rst(rst),
        .in(in),
        .q(q)
    );

    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    initial begin
        rst = 1; in = 0; #10;
        rst = 0; in = 1; #10;
        in = 0; #10;
        in = 1; #10;
        in = 1; #10;
        in = 0; #10;
```

```
        $finish;
    end
endmodule
```



44. Bidirectional Shift Register

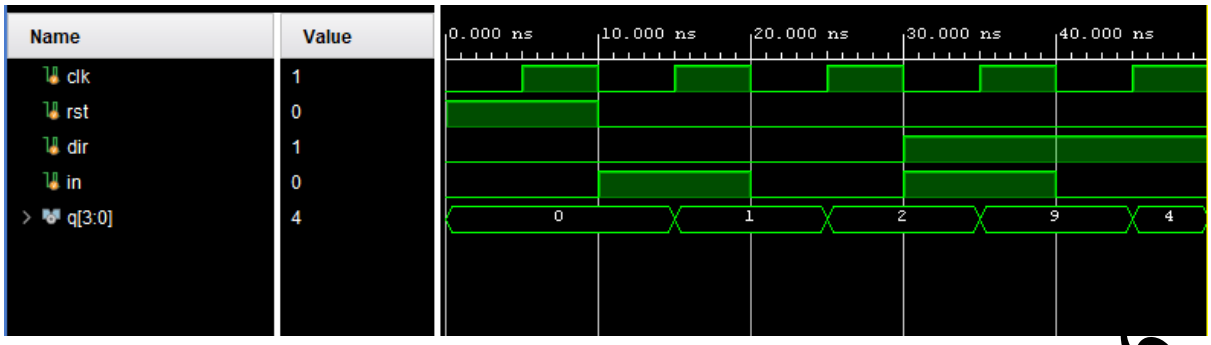
```
module shift_register_bidir(
    input clk,
    input rst,
    input dir,          // 0 = left, 1 = right
    input in,
    output reg [3:0] q
);

always @(posedge clk or posedge rst) begin
    if (rst)
        q <= 4'b0000;
    else if (dir)       // Right shift
        q <= {in, q[3:1]};
    else                // Left shift
        q <= {q[2:0], in};
end

endmodule
```

TB

```
module shift_register_bidir_TB;
```

```
reg clk;
reg rst;
reg dir;
reg in;
wire [3:0] q;

shift_register_bidir uut (
    .clk(clk),
    .rst(rst),
    .dir(dir),
    .in(in),
    .q(q)
);

initial begin
    clk = 0;
    forever #5 clk = ~clk;
end

initial begin
    rst = 1; dir = 0; in = 0; #10;
    rst = 0;
    in = 1; #10; // left shift in 1
    in = 0; #10; // left shift in 0
    dir = 1;
    in = 1; #10; // right shift in 1
    in = 0; #10; // right shift in 0
    $finish;
end
```

endmodule



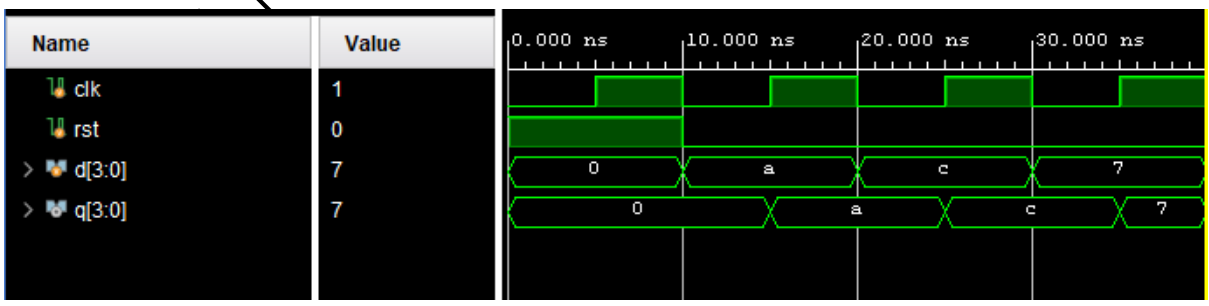
45. Parallel-In Parallel-Out Register

```
module pipo_register (
    input clk,
    input rst,
    input [3:0] d,
    output reg [3:0] q
);

always @(posedge clk or posedge rst) begin
    if (rst)
        q <= 4'b0000;
    else
        q <= d;
end

endmodule
```

TB

```
module pipo_register_TB;
    reg clk;
    reg rst;
    reg [3:0] d;
    wire [3:0] q;
```

```
    pipo_register uut (
        .clk(clk),
        .rst(rst),
        .d(d),
        .q(q)
    );

    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    initial begin
        rst = 1; d = 4'b0000; #10;
        rst = 0; d = 4'b1010; #10;
        d = 4'b1100; #10;
        d = 4'b0111; #10;
        $finish;
    end
endmodule
```

46. 4-bit Asynchronous Counter

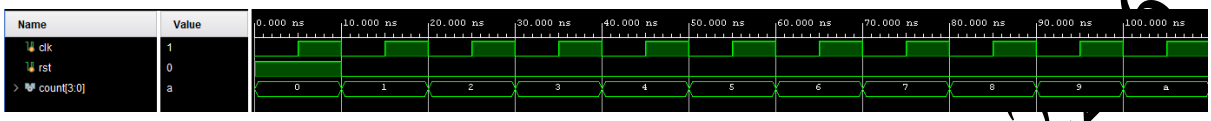
```
module async_counter_4bit (
    input clk,
    input rst,
```

```
    output reg [3:0] count
);

always @(negedge clk or posedge rst) begin
    if (rst)
        count <= 4'b0000;
    else
        count <= count + 1;
end

endmodule
```

TB

```
module tb_async_counter_4bit;

reg clk, rst;
wire [3:0] count;

async_counter_4bit uut (
    .clk(clk),
    .rst(rst),
    .count(count)
);

initial begin
    clk = 0;
    forever #5 clk = ~clk;
end

initial begin
```

```
    rst = 1;
    #10 rst = 0;
    #100 $finish;
end

endmodule
```



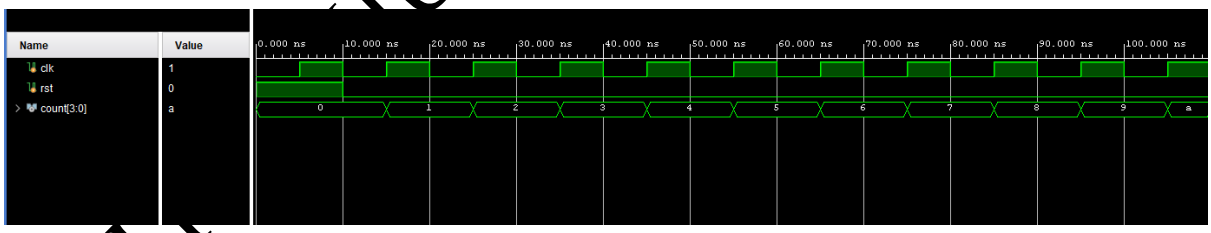
47. 4-bit Synchronous Counter

```
module sync_counter_4bit(
    input clk,
    input rst,
    output reg [3:0] count
);

always @(posedge clk or posedge rst) begin
    if (rst)
        count <= 4'b0000;
    else
        count <= count + 1;
end

endmodule
```

TB

```
module sync_counter_4bitTB;

reg clk, rst;
wire [3:0] count;
```

```
sync_counter_4bit uut (
    .clk(clk),
    .rst(rst),
    .count(count)
);

initial begin
    clk = 0;
    forever #5 clk = ~clk;
end

initial begin
    rst = 1;
    #10 rst = 0;
    #100 $finish;
end

endmodule
```



4B. Up Counter

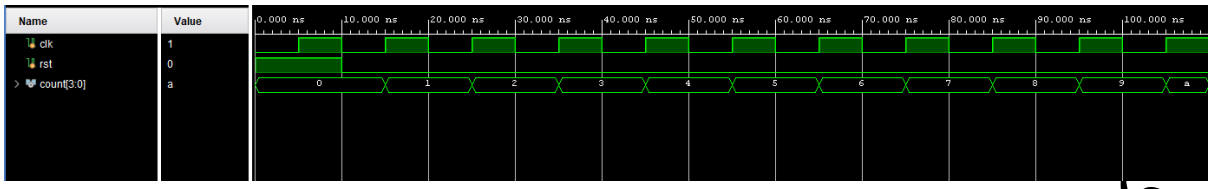
```
module up_counter(
    input clk,
    input rst,
    output reg [3:0] count
);
```

```
always @(posedge clk or posedge rst) begin
    if (rst)
        count <= 4'b0000;
    else
        count <= count + 1;
end


endmodule
```

TB

```
module up_counterTB;


reg clk, rst;
wire [3:0] count;


up_counter uut (
    .clk(clk),
    .rst(rst),
    .count(count)
);


initial begin
    clk = 0;
    forever #5 clk = ~clk;
end


initial begin
    rst = 1;
    #10 rst = 0;
    #100 $finish;
```

```
end


endmodule
```



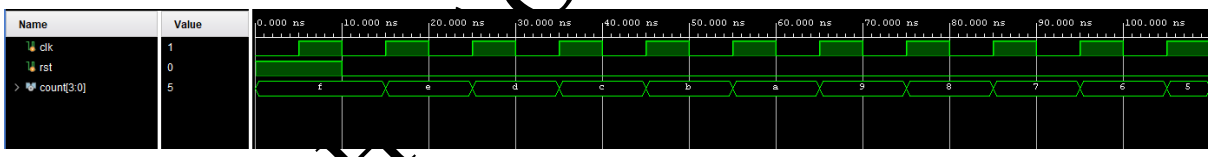
49. Down Counter

```
module down_counter(
    input clk,
    input rst,
    output reg [3:0] count
);

always @(posedge clk or posedge rst) begin
    if (rst)
        count <= 4'b1111;
    else
        count <= count - 1;
end

endmodule
```

TB

```
module down_counterTB;

reg clk, rst;
wire [3:0] count;

down_counter uut (
    .clk(clk),
```

```
    .rst(rst),
    .count(count)
);

initial begin
    clk = 0;
    forever #5 clk = ~clk;
end

initial begin
    rst = 1;
    #10 rst = 0;
    #100 $finish;
end

endmodule
```



50. Up/Down Counter

```
module up_down_counter(
    input clk,
    input rst,
    input up_down, // 1: up, 0: down
    output reg [3:0] count
);

always @(posedge clk or posedge rst) begin
    if (rst)
```

```
        count <= 4'b0000;
    else if (up_down)
        count <= count + 1;
    else
        count <= count - 1;
end

endmodule
```

TB

```
module up_down_counterTB;

reg clk, rst, up_down;
wire [3:0] count;

up_down_counter uut (
    .clk(clk),
    .rst(rst),
    .up_down(up_down),
    .count(count)
);

initial begin
    clk = 0;
    forever #5 clk = ~clk;
end

initial begin
    rst = 1; up_down = 1;
    #10 rst = 0;
```

```
    #50 up_down = 0;
    #50 $finish;
end

endmodule
```