TASK-1 Design and Implementation of Digital Circuits in Verilog

VLSI Internship

Submitted By:

Sidhanth Krishnan

To the

Hardware Technology Group

Centre for Development of Advanced Computing

Centre In North East (CDAC-CINE)
Research Park, IIT Guwahati, Guwahati, Assam

June 2025

Contents

1 Design and Implementation of Digital Circuits in Verilog

This section details the implementation of digital logic circuits in Verilog across gate-level, dataflow, and behavioral modeling styles. All simulations were verified using waveform outputs.

1.1 1. Basic Logic Gates

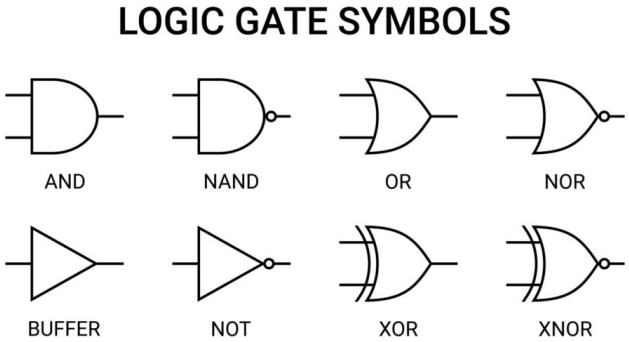

Fig. 1: ALL Basic Logic Gates

Truth Table:

Table 1: AND Gate Truth Table

A	B	Y = A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Table 2: OR Gate Truth Table

A	B	Y = A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Table 3: NAND Gate Truth Table

A	B	Y = A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

Table 4: NOR Gate Truth Table

A	B	Y = A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

Table 5: XOR Gate Truth Table

A	B	Y = A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Table 6: XNOR Gate Truth Table

A	B	Y = A XNOR B
0	0	1
0	1	0
1	0	0
1	1	1

Table 7: NOT Gate Truth Table

A	Y = NOT A
0	1
1	0

Verilog Code Gate level:

```
module Logic_gates_Gatelevel(a,b,o1,o2,o3,o4,o5,o6,o7);
input a,b;
output o1,o2,o3,o4,o5,o6,o7;
and(o1,a,b);
or(o2,a,b);
nand(o3,a,b);
nor(o4,a,b);
xor(o5,a,b);
xnor(o6,a,b);
not(o7,a,b);
endmodule
```

Verilog Code Data flow:

```
module Logic_gates_Dataflow(a,b,o1,o2,o3,o4,o5,o6,o7,o8);
input a,b;
output o1,o2,o3,o4,o5,o6,o7,o8;
assign o1=a&&b;
assign o2=a||b;
assign o3=~(a&&b);
assign o4=~(a||b);
assign o5=a^b;
```

```
assign o6=~(a^b);
assign o7=~a;
assign o8=~b;
endmodule
```

Verilog Code Behavioral:

```
module Logic_gates_Behavioral(a,b,o1,o2,o3,o4,o5,o6,o7);
input a,b;
output o1,o2,o3,o4,o5,o6,o7;
reg o1,o2,o3,o4,o5,o6,o7;
always@(*) begin
o1=a&&b;
o2=a||b;
o3=~(a&&b);
o4=~(a||b);
o5=a^b;
o6=~(a^b);
o7=~a;
end
endmodule
```

1.2 2. Multiplexers

1.2.1 2:1 MUX

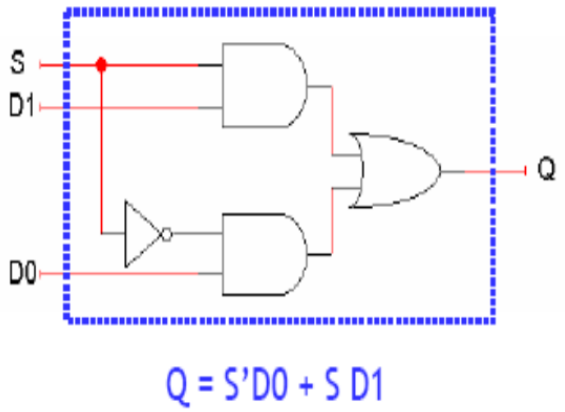

Fig. 2: 2:1 Multiplexer Circuit

Verilog Code 2x1 MUX:

```
module mux2x1(a,b,s,o);
input a,b,s;
output o;
assign o=(s==1)?b:a;
endmodule

module tb_mux2x1();
    reg a, b, s;
    wire o;

    mux2x1 uut(.a(a), .b(b), .s(s), .o(o));

    initial begin
        $display("2:1 MUX Test");
```

```
        $monitor("s=%b -> o=%b", s, o);
        a = 0; b = 1; s = 0; #10;
        s = 1; #10;
        $finish;
    end
endmodule
```

Table 8: 2:1 MUX Truth Table

S	I0	I1	Y
0	0	x	0
0	1	x	1
1	x	0	0
1	x	1	1

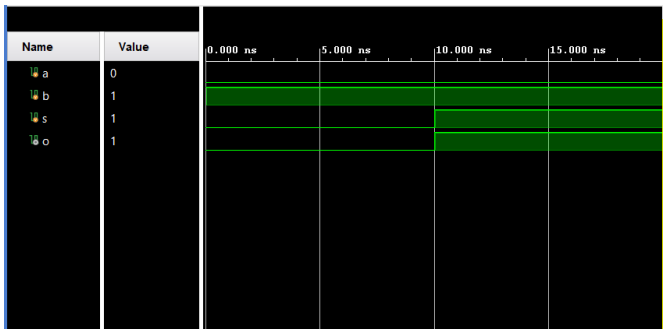

Fig. 3: 2:1 Multiplexer Circuit simulation

Simulation Observations

The 2:1 multiplexer was accurately realized

1.2.2 4:1 MUX

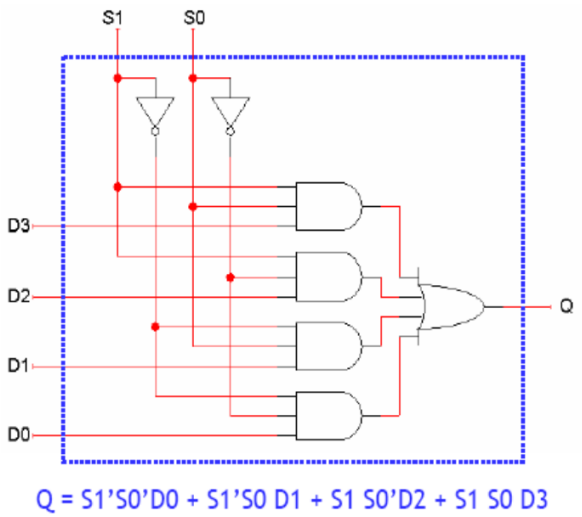

Fig. 4: 4:1 Multiplexer Circuit

Verilog Code 4x1 MUX:

```
module mux4x1(input [3:0] i,input [1:0] s,output o);
wire x1,x2;
mux2x1 m0(.a(i[0]),.b(i[1]),.s(s[0]),.o(x1));
mux2x1 m1(.a(i[2]),.b(i[3]),.s(s[0]),.o(x2));
mux2x1 m3(.a(x1),.b(x2),.s(s[1]),.o(o));
endmodule

module tb_mux4x1();
    reg [3:0] i;
    reg [1:0] s;
    wire o;
    mux4x1 uut(.i(i), .s(s), .o(o));
    initial begin
        i = 4'b1010;
        s = 2'b00; #5; $display("s=%b -> o=%b", s, o);
        s = 2'b01; #5; $display("s=%b -> o=%b", s, o);
        s = 2'b10; #5; $display("s=%b -> o=%b", s, o);
        s = 2'b11; #5; $display("s=%b -> o=%b", s, o);
        $finish;
    end
endmodule
```

Table 9: 4:1 MUX Truth Table

S1	S0	I0	I1	I2	I3	Y
0	0	x	x	x	x	I0
0	1	x	x	x	x	I1
1	0	x	x	x	x	I2
1	1	x	x	x	x	I3

Fig. 5: 4:1 Multiplexer Circuit simulation

Simulation Observations

The 4:1 multiplexer was accurately realized

1.2.3 7:1 MUX

Table 10: 7:1 MUX Truth Table

S2	S1	S0	I0	I1	I2	I3	I4	I5	I6	Y
0	0	0	x	x	x	x	x	x	x	I0
0	0	1	x	x	x	x	x	x	x	I1
0	1	0	x	x	x	x	x	x	x	I2
0	1	1	x	x	x	x	x	x	x	I3
1	0	0	x	x	x	x	x	x	x	I4
1	0	1	x	x	x	x	x	x	x	I5
1	1	0	x	x	x	x	x	x	x	I6

Verilog Code 7x1 MUX:

```
module mux7x1(input [6:0] i, input [2:0] s, output o);
    wire [3:0] w1;
    wire [1:0] w2;
    wire dummy = 1'b0;
    mux2x1 m0 (.a(i[0]), .b(i[1]), .s(s[0]), .o(w1[0]));
    mux2x1 m1 (.a(i[2]), .b(i[3]), .s(s[0]), .o(w1[1]));
    mux2x1 m2 (.a(i[4]), .b(i[5]), .s(s[0]), .o(w1[2]));
    mux2x1 m3 (.a(i[6]), .b(dummy), .s(s[0]), .o(w1[3]));
    mux2x1 m4 (.a(w1[0]), .b(w1[1]), .s(s[1]), .o(w2[0]));
    mux2x1 m5 (.a(w1[2]), .b(w1[3]), .s(s[1]), .o(w2[1]));
    mux2x1 m6 (.a(w2[0]), .b(w2[1]), .s(s[2]), .o(o));
endmodule

module tb_mux7x1();
    reg [6:0] i;
    reg [2:0] s;
    wire o;

    mux7x1 uut(.i(i), .s(s), .o(o));

    initial begin
        $display("7:1 MUX Test");
        i = 7'b1010101;
        for (integer j = 0; j < 8; j = j + 1) begin
            s = j[2:0]; #5;
            $display("s=%b -> o=%b", s, o);
        end
        $finish;
    end
endmodule
```

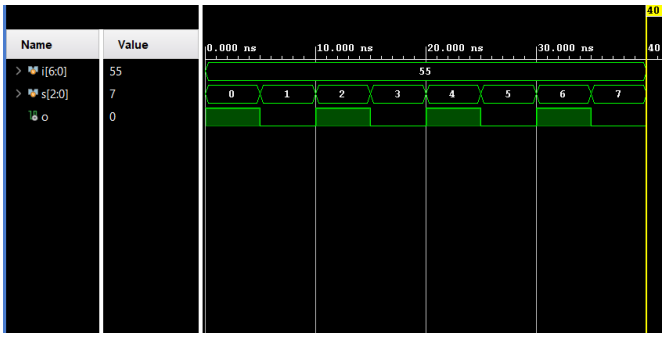

Fig. 6: 7:1 Multiplexer Circuit simulation

Simulation Observations

The 7:1 multiplexer was accurately realized

1.2.4 13:1 MUX

Table 11: 13:1 MUX Truth Table (Partial)

S3	S2	S1	S0	I0	I1	I2	I3	I4	I5	I6	I7	I8	I9	Y
0	0	0	0	x	x	x	x	x	x	x	x	x	x	I0
0	0	0	1	x	x	x	x	x	x	x	x	x	x	I1
0	0	1	0	x	x	x	x	x	x	x	x	x	x	I2
0	0	1	1	x	x	x	x	x	x	x	x	x	x	I3
0	1	0	0	x	x	x	x	x	x	x	x	x	x	I4
0	1	0	1	x	x	x	x	x	x	x	x	x	x	I5
0	1	1	0	x	x	x	x	x	x	x	x	x	x	I6
⋮	⋮	⋮	⋮						...					⋮
⋮	⋮													

Verilog Code 13x1 MUX:

```
module mux13x1(input [12:0] i, input [3:0] s, output o);
    wire [7:0] w1;
    wire [3:0] w2;
    wire [1:0] w3;
    wire dummy0 = 1'b0;
    wire dummy1 = 1'b0;
    wire dummy2 = 1'b0;

    mux2x1 m0 (.a(i[0]),  .b(i[1]),     .s(s[0]), .o(w1[0]));
    mux2x1 m1 (.a(i[2]),  .b(i[3]),     .s(s[0]), .o(w1[1]));
    mux2x1 m2 (.a(i[4]),  .b(i[5]),     .s(s[0]), .o(w1[2]));
    mux2x1 m3 (.a(i[6]),  .b(i[7]),     .s(s[0]), .o(w1[3]));
    mux2x1 m4 (.a(i[8]),  .b(i[9]),     .s(s[0]), .o(w1[4]));
    mux2x1 m5 (.a(i[10]), .b(i[11]),    .s(s[0]), .o(w1[5]));
    mux2x1 m6 (.a(i[12]), .b(dummy0),   .s(s[0]), .o(w1[6]));
    mux2x1 m7 (.a(dummy1),.b(dummy2),   .s(s[0]), .o(w1[7]));

    mux2x1 m8  (.a(w1[0]), .b(w1[1]), .s(s[1]), .o(w2[0]));
    mux2x1 m9  (.a(w1[2]), .b(w1[3]), .s(s[1]), .o(w2[1]));
    mux2x1 m10 (.a(w1[4]), .b(w1[5]), .s(s[1]), .o(w2[2]));
    mux2x1 m11 (.a(w1[6]), .b(w1[7]), .s(s[1]), .o(w2[3]));

    mux2x1 m12 (.a(w2[0]), .b(w2[1]), .s(s[2]), .o(w3[0]));
    mux2x1 m13 (.a(w2[2]), .b(w2[3]), .s(s[2]), .o(w3[1]));

    mux2x1 m14 (.a(w3[0]), .b(w3[1]), .s(s[3]), .o(o));
endmodule

module tb_mux13x1;
    reg [12:0] i;
    reg [3:0] s;
    wire o;

    mux13x1 uut(.i(i), .s(s), .o(o));

    initial begin
        $display("13:1 MUX Test");
        i = 13'b1010011101110;
        for (integer k = 0; k < 16; k = k + 1) begin
```

```
            s = k[3:0]; #5;
            $display("s=%b -> o=%b", s, o);
        end
        $finish;
    end
endmodule
```

Fig. 7: 13:1 Multiplexer Circuit simulation

Simulation Observations

The 13:1 multiplexer was accurately realized

1.3 Binary to Gray and Gray to Binary Converters

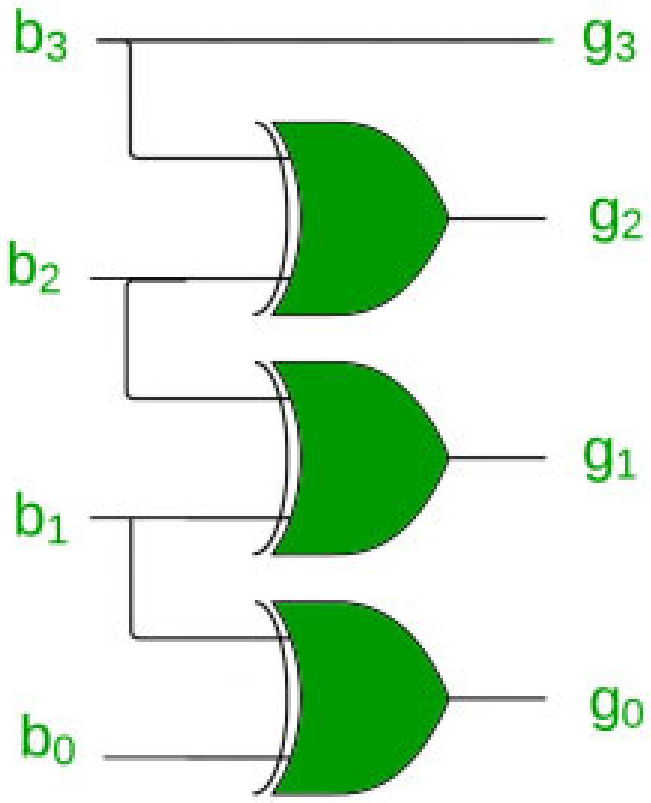

Fig. 8: Binary to Gray Code Converter

Verilog Code (Binary to Gray):

```
module BtoGrey(input [3:0]b,output reg [3:0]g);
reg prev;
integer i;
always@(*)begin
prev=1'b0;
for(i=3;i>=0;i=i-1)begin
g[i]=b[i]^prev;
prev=b[i];
end
end
endmodule
```

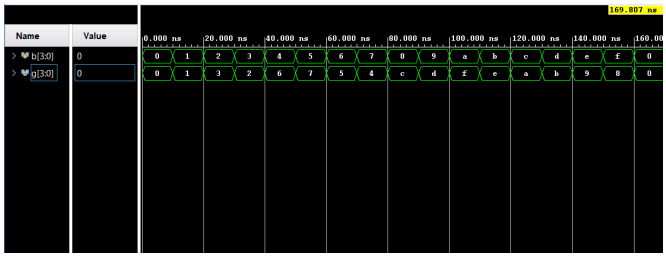

Fig. 9: Binary To Gray Code simulation

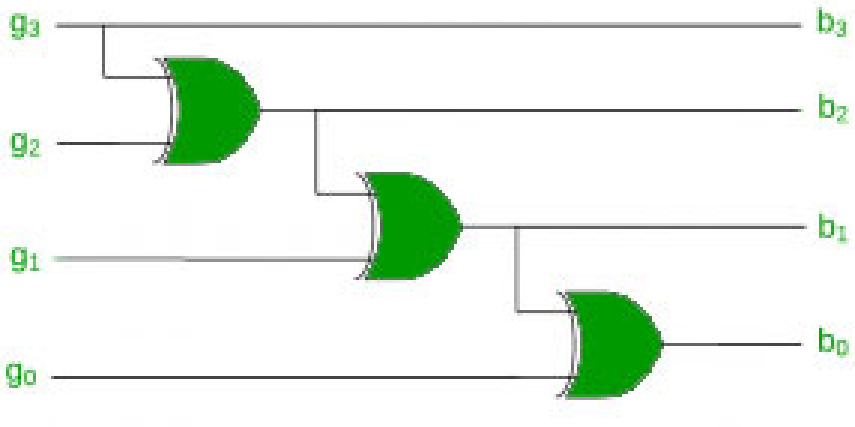

Fig. 10: Gray to Binary Code Converter

Verilog Code (Gray to Binary):

```
module GtoBinary(input [3:0] g,output reg [3:0] b);
reg prev;
integer i;
always@(*)begin
prev=1'b0;
for(i=3;i>=0;i=i-1)begin
b[i]=prev^g[i];
prev=b[i];
end
end
endmodule
```

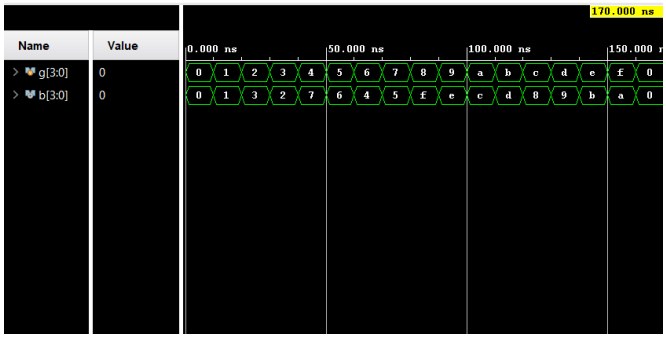

Fig. 11: Gray To Binary Code simulation

1.4 Adders

1.4.1 Half Adder

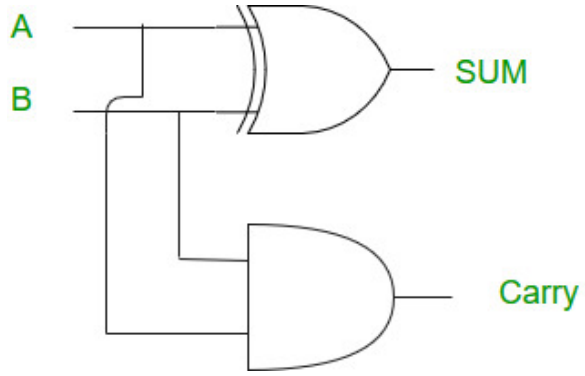

Fig. 12: Half Adder Circuit Diagram

Truth Table:

Table 12: Half Adder Truth Table

A	B	Sum (S), Carry (C)
0	0	0, 0
0	1	1, 0
1	0	1, 0
1	1	0, 1

Verilog Code:

```
module HA(a,b,s,c);
input a,b;
output s,c;
assign s=a^b;
assign c=a&b;
endmodule
```

Testbench:

```
module tb_HA();
reg a,b;
wire s,c;
HA uut(.a(a),.b(b),.s(s),.c(c));
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
```

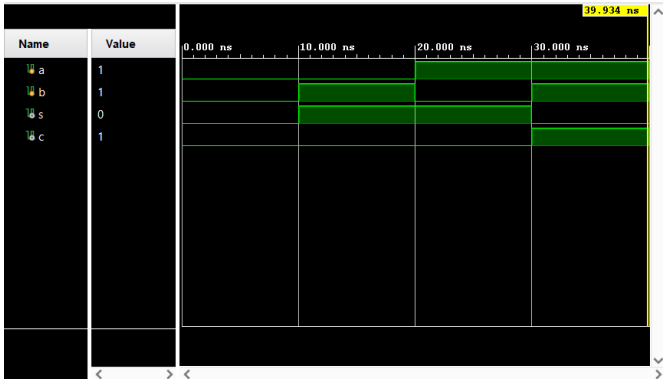

Fig. 13: Gray To Binary Code simulation

1.4.2 Full Adder

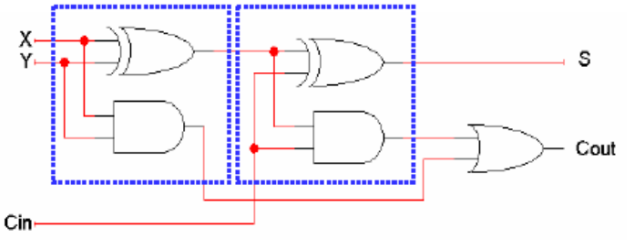

Fig. 14: Full Adder Circuit Diagram

Truth Table:

Table 13: Full Adder Truth Table

A	B	Cin	Sum (S), Cout
0	0	0	0, 0
0	0	1	1, 0
0	1	0	1, 0
0	1	1	0, 1
1	0	0	1, 0
1	0	1	0, 1
1	1	0	0, 1
1	1	1	1, 1

Verilog Code:

```
module FA(input a, input b, input cin, output s, output cout);
    assign s = a ^ b ^ cin;
    assign cout = (a & b) | (b & cin) | (a & cin);
endmodule
```

Testbench:

```
module tb_FA();
reg [3:0] a;
wire s,cout;
```

```
FA uut(.a(a[0]),.b(a[1]),.cin(a[2]),.s(s),.cout(cout));
initial begin
for(a=3'b000;a!=3'b111;a=a+3'b001)begin
#10;
end
#10;
$finish;
end
endmodule
```

Fig. 15: Gray To Binary Code simulation

1.4.3 Half Subtractor

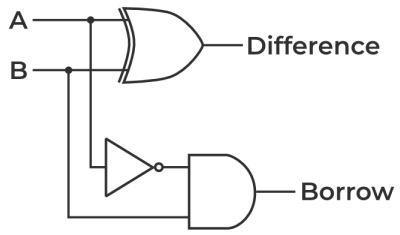

Fig. 16: Half Subtractor Circuit Diagram

Truth Table:

Table 14: Half Subtractor Truth Table

A	B	Diff (D), Borrow (Bo)
0	0	0, 0
0	1	1, 1
1	0	1, 0
1	1	0, 0

Verilog Code:

```
module HS(a, b, d, b_out);
  input a, b;
  output d, b_out;
```

```
  assign d = a ^ b;
  assign b_out = ~a & b;
endmodule
```

Testbench:

```
module tb_HS();
  reg a, b;
  wire d, b_out;

  HS uut(.a(a), .b(b), .d(d), .b_out(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
endmodule
```

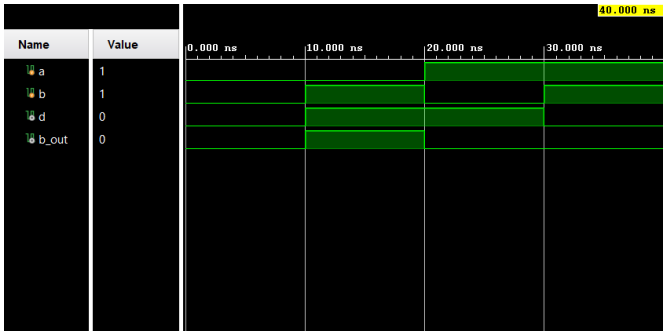

Fig. 17: Gray To Binary Code simulation

1.4.4 Full Subtractor

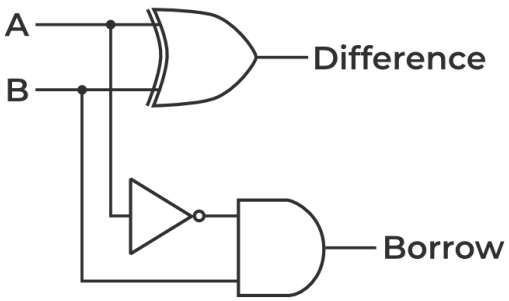

Fig. 18: Full Subtractor Circuit Diagram

Truth Table:

Table 15: Full Subtractor Truth Table

A	B	Bin	Diff (D), Bout
0	0	0	0, 0
0	0	1	1, 1
0	1	0	1, 1
0	1	1	0, 1
1	0	0	1, 0
1	0	1	0, 1
1	1	0	0, 0
1	1	1	1, 1

Verilog Code:

```
module FS(a, b, bin, d, bout);
  input a, b, bin;
  output d, bout;

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

Testbench:

```
module tb_FS();
  reg [2:0] in;
  wire d, bout;

  FS uut(.a(in[0]), .b(in[1]), .bin(in[2]), .d(d), .bout(bout));
  integer i;
  initial begin
    for (i = 0; i < 8; i = i + 1) begin
      in = i[2:0];
      #10;
    end
    $finish;
  end
endmodule
```

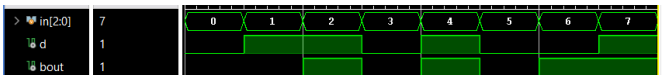

Fig. 19: Full subtractor Simulation

1.5 4-bit Full Adder and Ripple Carry Adder

1.5.1 4-bit Full Adder Module

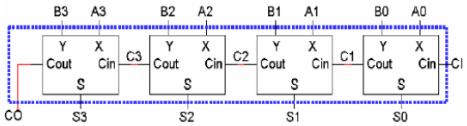

Fig. 20: 4-bit Full Adder Circuit

Verilog Code 4 bit adder:

```
module fourBIT_FA(input [3:0]a,input [3:0] b,input cin,output[3:0] o,output cout);
assign {cout,o}=a+b+cin;
endmodule
```

Verilog Code ripple adder:

```
module rippleadder(input [3:0]a,input [3:0] b,input cin,output[3:0] o,output cout);
wire c1,c2,c3;
FA f0(.a(a[0]),.b(b[0]),.cin(cin),.s(o[0]),.cout(c1));
FA f1(.a(a[1]),.b(b[1]),.cin(c1),.s(o[1]),.cout(c2));
FA f2(.a(a[2]),.b(b[2]),.cin(c2),.s(o[2]),.cout(c3));
FA f3(.a(a[3]),.b(b[3]),.cin(c3),.s(o[3]),.cout(cout));
endmodule
```

Table 16: 4-bit Ripple Carry Adder Sample Truth Table

A (4-bit)	B (4-bit)	Cin	Sum (4-bit)	Cout
0000	0000	0	0000	0
0101	0011	0	1000	0
1111	0001	0	0000	1
1010	0101	0	1111	0

Testbench for 4 bit adder:

```
module tb_fourBIT_FA();
reg [3:0]a;
reg [3:0]b;
reg cin;
wire [3:0]o;
wire cout;
fourBIT_FA uut (.a(a),.b(b),.cin(cin),.o(o),.cout(cout));
initial begin
a=4'b0000;
cin=0;
for(b=4'b0000;b<=4'b1111;b=b+4'b0001)begin
#10;
end
end
endmodule
```

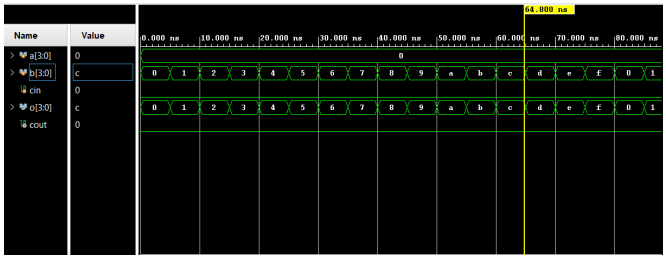

Fig. 21: 4-bit Adder Simulation

Testbench for ripple adder:

```
module tb_rippleadder();
reg [3:0]a;
reg [3:0]b;
reg cin;
wire [3:0]o;
wire cout;
```

```
rippleadder uut (.a(a),.b(b),.cin(cin),.o(o),.cout(cout));
initial begin
a=4'b0000;
cin=0;
for(b=4'b0000;b<=4'b1111;b=b+4'b0001)begin
#10;
end
end
endmodule
```

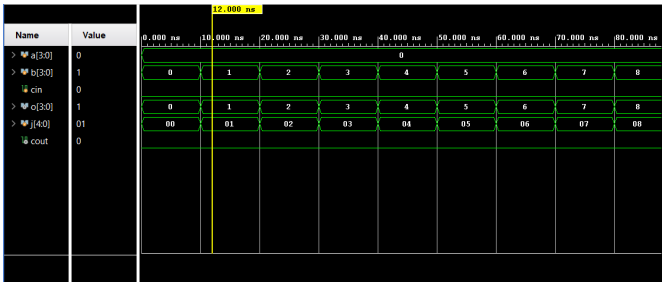

Fig. 22: ripple Adder Simulation

1.6 Priority Encoder

1.6.1 4:2 Priority Encoder

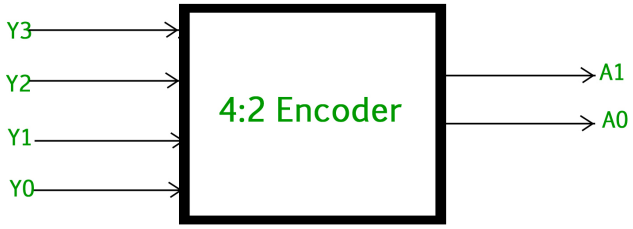

Fig. 23: 4:2 Priority Encoder Circuit

Verilog Code 4:2 Priority Encoder:

```
module priority_encoder(input [3:0] i,output [1:0] o);
assign o[1]=i[3]|i[2];
assign o[0]=i[3]|((~i[2])&i[1]);
endmodule
```

Table 17: 4:2 Priority Encoder Truth Table

I3	I2	I1	I0	O1	O0
0	0	0	1	0	0
0	0	1	x	0	1
0	1	x	x	1	0
1	x	x	x	1	1

Testbench:

```
module tb_priorityencoder();
reg [3:0] i;
wire [1:0] o;
priority_encoder uut(.i(i),.o(o));
initial
begin
i=4'b0000;
repeat(16)
begin
#10;
i=i+1;

end
$finish;

end
endmodule
```

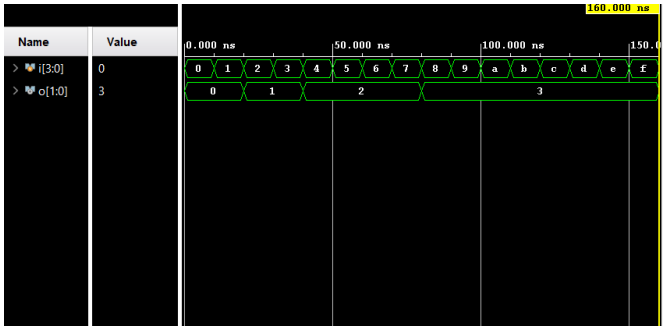

Fig. 24: 4:2 Priority Encoder Simulation

1.7 Latches

1.7.1 SR Latch

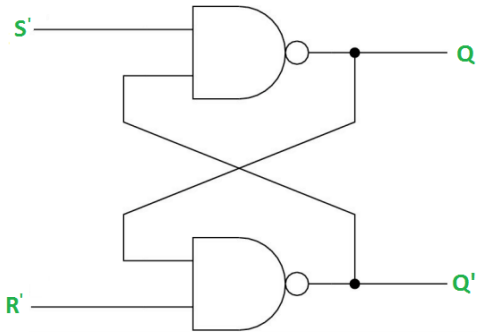

Fig. 25: SR Latch Circuit

Verilog Code:

```
module SR_LATCH(input s,input r,input e,output reg q,output q_bar);
assign q_bar=!q;
always@(*)begin

if(e && (s|r))begin
```

```
    if(s && r) q=1'bx;
    else q=s;
end

end
endmodule
```

Table 18: SR Latch Truth Table

S	R	Q	Q'
0	0	Q	Q'
0	1	0	1
1	0	1	0
1	1	Invalid	Invalid

Testbench:

```
module tb_SR_LATCH();
  reg s, r, e;
  wire q, q_bar;
  reg [3:0] vec;
  SR_LATCH uut (
    .s(s),
    .r(r),
    .e(e),
    .q(q),
    .q_bar(q_bar)
  );

  initial begin
    for (vec = 4'b0000; vec <= 4'b0111; vec = vec + 1) begin
      {e,s,r}=vec[2:0];
    #10;
    end
    $finish;
  end
endmodule
```

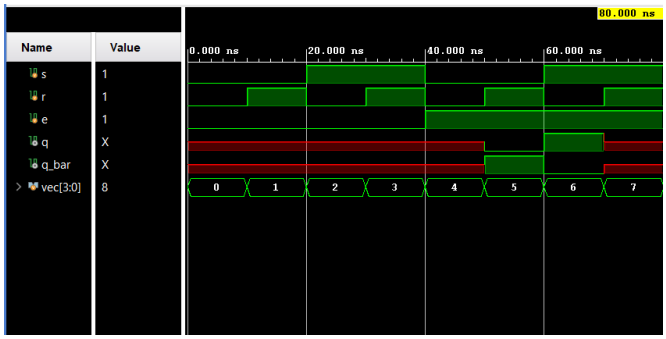

Fig. 26: SR Latch Simulation

1.7.2 JK Latch

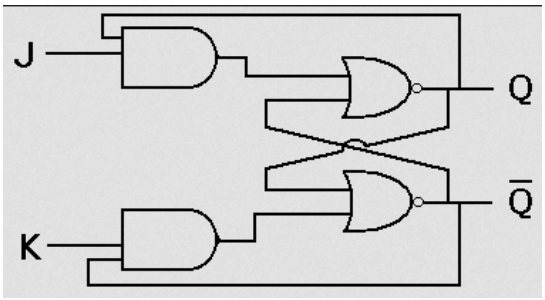

Fig. 27: JK Latch Circuit

Verilog Code:

```
module JK_latch(input j,input k,input e,output reg q,output q_bar);
assign q_bar=~q;
always@(*)begin

if(e)begin

if({j,k}==2'b01) q=0;
if({j,k}==2'b10) q=1;
if({j,k}==2'b11) q=~q;

end

end
endmodule
```

Table 19: JK Latch Truth Table

J	K	EN	Q(next)
0	0	1	Q
0	1	1	0
1	0	1	1
1	1	1	Q

Testbench:

```
module tb_JK_latch();
  reg j, k, e;
  wire q, q_bar;
  reg [3:0] vec;
  JK_latch uut (
    .j(j),
    .k(k),
    .e(e),
    .q(q),
    .q_bar(q_bar)
  );
  initial begin
    for (vec = 4'b0000; vec <= 4'b0111; vec = vec + 1) begin
      {e, j, k} = vec[2:0];
      #10;
    end
    $finish;
  end
endmodule
```

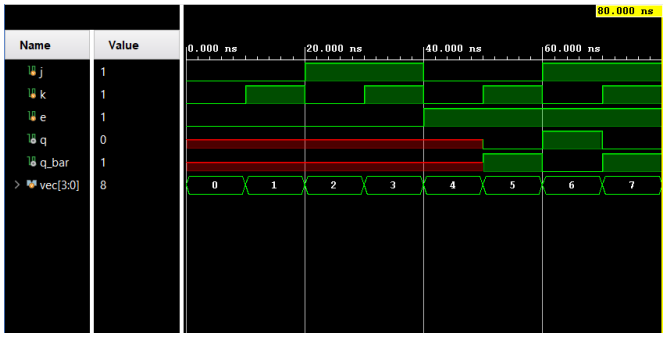

Fig. 28: JK Latch Simulation

1.7.3 T Latch

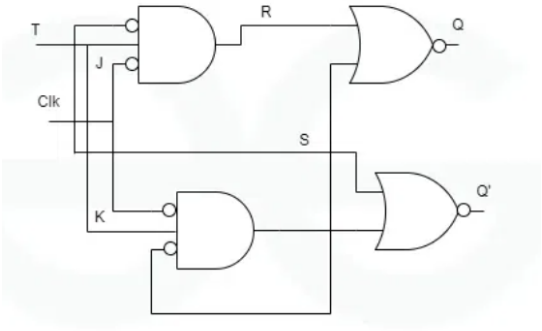

Fig. 29: T Latch Circuit

Verilog Code:

```
module T_latch(input t,input e,output q,output q_bar);
JK_latch m0(.j(t),.k(t),.e(e),.q(q),.q_bar(q_bar));
endmodule
```

Table 20: T Latch Truth Table

T	EN	Q(next)
0	1	Q
1	1	Q

1.7.4 D Latch

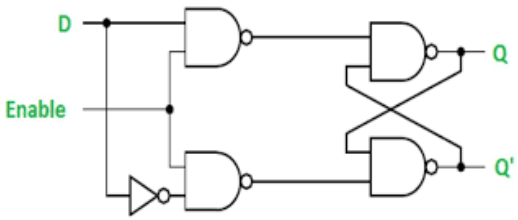

Fig. 30: D Latch Circuit

Verilog Code:

```
module D_latch(input d,input e,output reg q,output q_bar);
assign q_bar=~q;
always@(*)begin

if(e)begin
q=d;
end

end
endmodule
```

Table 21: D Latch Truth Table

D	EN	Q(next)
0	1	0
1	1	1

Testbench:

```
module tb_D_latch();
  reg d, e;
  wire q, q_bar;
  reg [2:0] vec;
  D_latch uut (
    .d(d),
    .e(e),
    .q(q),
    .q_bar(q_bar)
  );

  initial begin
    for (vec = 3'b000; vec <= 3'b011; vec = vec + 1) begin
      {e,d}=vec[1:0];
    #10;
    end
   $finish;
  end
endmodule
```

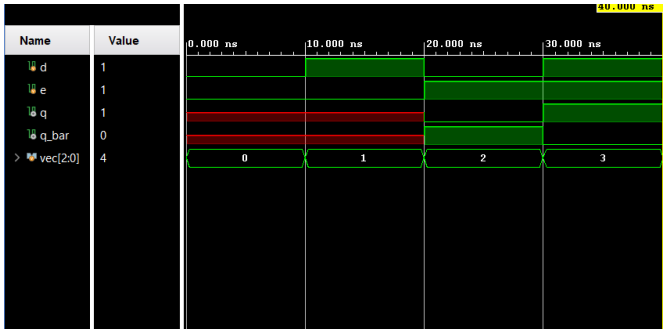

Fig. 31: D Latch Simulation

1.8 Flip-Flops

1.8.1 T Flip-Flop

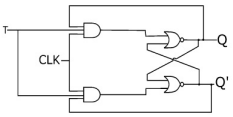

Fig. 32: T Flip-Flop Circuit

Verilog Code:

```
`timescale 1ns / 1ps
module T_ff(input t, input clk, input pr, input cr, output reg q, output q_bar);
  assign q_bar = ~q;

  always @(posedge clk or negedge pr or negedge cr) begin
    if (!cr)
      q <= 0;
    else if (!pr)
      q <= 1;
    else if (t)
      q <= ~q;
  end
endmodule
```

Table 22: T Flip-Flop Truth Table

T	Q (next)	Description
0	Q	Hold
1	∼Q	Toggle

Testbench:

```
`timescale 1ns / 1ps
module tb_T_ff();
  reg t, clk, pr, cr;
  wire q, q_bar;

  T_ff uut(.t(t), .clk(clk), .pr(pr), .cr(cr), .q(q), .q_bar(q_bar));

  initial begin
```

```
    clk = 0;
    t = 0;
    pr = 1; cr = 0; #2;
    cr = 1; #2;
    t = 1; #10;
    t = 0; #10;
    t = 1; #10;
    $finish;
  end

  always #5 clk = ~clk;
endmodule
```

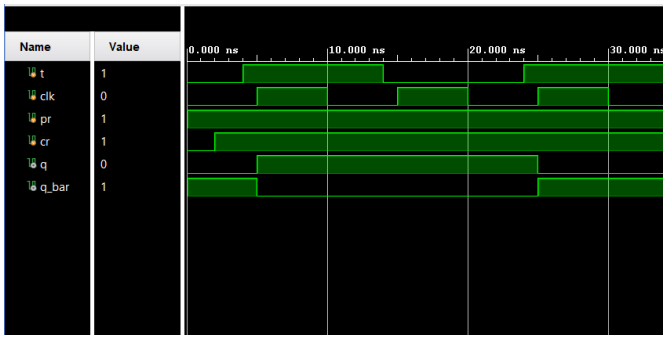

Fig. 33: T Flip-Flop Simulation

1.8.2 SR Flip-Flop

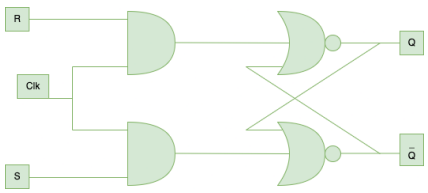

Fig. 34: SR Flip-Flop Circuit

Verilog Code:

```
module SR_ff(input s,input r,input clk,output reg q,output q_bar);
assign q_bar=~q;
always@(posedge clk)begin
if({s,r}==2'b01) q<=0;
if({s,r}==2'b10) q<=1;
if({s,r}==2'b11) q<=1'bx;
end
endmodule
```

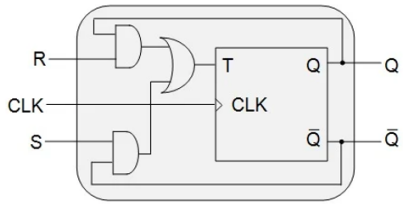

Fig. 35: SR Flip-Flop using a T-FlipFlop

Verilog Code:

```
module SR_using_T(input s,input clk,input r,input pr,input cr,output q,output q_bar);
T_ff t0(.t((s&q_bar)|(r&q)),.clk(clk),.pr(pr),.cr(cr),.q(q),.q_bar(q_bar));
endmodule
```

Table 23: SR Flip-Flop Truth Table

S	R	Q(next)	Description
0	0	Q	Hold
0	1	0	Reset
1	0	1	Set
1	1	X	Invalid

Testbench:

```
module tb_sr_ff();
reg clk;
reg s,r;
wire q,q_bar;
SR_ff uut(.clk(clk),.q(q),.s(s),.r(r),.q_bar(q_bar));
initial begin
clk=0;
s=0;r=0;
#10; s=0;r=1;
#10; s=1;r=0;
#10; s=0;r=0;
#10 $finish;
end

always begin
#5;
clk=~clk;
end

endmodule
```

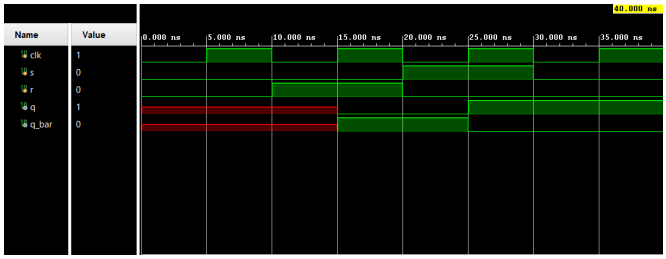

Fig. 36: SR Flip-Flop Simulation

1.8.3 JK Flip-Flop

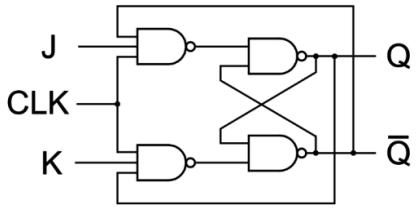

Fig. 37: JK Flip-Flop Circuit

Verilog Code:

```
module JK_ff(input j,input k,input clk,output reg q,output q_bar);
assign q_bar=~q;
always@(posedge clk)begin
if({j,k}==2'b01) q<=0;
if({j,k}==2'b10) q<=1;
if({j,k}==2'b11) q<=~q;
end
endmodule
```

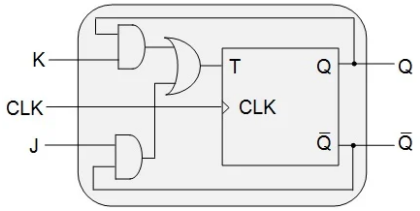

Fig. 38: JK Flip-Flop using T-FlipFlop

Verilog Code:

```
module JK_using_T(input j,input k,input clk,input pr,input cr,output q,output q_bar);
wire t=(j & ~q) | (k & q);
T_ff t0(.t(t),.clk(clk),.pr(pr),.cr(cr),.q(q),.q_bar(q_bar));
endmodule
```

Table 24: JK Flip-Flop Truth Table

J	K	Q(next)	Description
0	0	Q	Hold
0	1	0	Reset
1	0	1	Set
1	1	∼Q	Toggle

Testbench:

```
module tb_jk_ff();
reg clk;
reg j,k;
wire q,q_bar;
JK_ff uut(.clk(clk),.q(q),.j(j),.k(k),.q_bar(q_bar));
initial begin
clk=0;
```

```
j=0;k=0;
#10; j=0;k=1;
#10; j=1;k=0;
#10; j=0;k=0;
#10 $finish;
end

always begin
#5;
clk=~clk;
end

endmodule
```

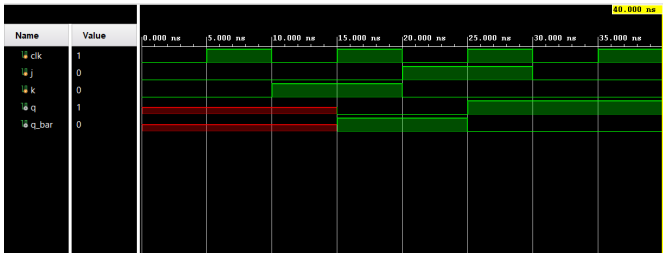

Fig. 39: JK Flip-Flop Simulation

1.8.4 D Flip-Flop

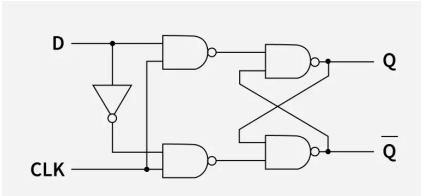

Fig. 40: D Flip-Flop Circuit

Verilog Code:

```
module D_ff(input d,output q,output q_bar,input clk);
SR_ff m0(.s(d),.r(~d),.q(q),.q_bar(q_bar),.clk(clk));
endmodule
```

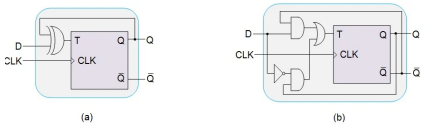

Fig. 41: D Flip-Flop Circuit

Verilog Code:

```
module D_using_T(input d,input clk,input pr,input cr,output q,output q_bar);
wire t=d^q;
T_ff t0(.t(t),.clk(clk),.pr(pr),.cr(cr),.q(q),.q_bar(q_bar));
endmodule
```

Table 25: D Flip-Flop Truth Table

D	Q(next)	Description
0	0	Reset
1	1	Set

Testbench:

```
module tb_d_ff();
  reg clk, d;
  wire q, q_bar;
  D_ff uut(.d(d), .clk(clk), .q(q), .q_bar(q_bar));
  initial begin
  clk=0;
    d = 0; #10;
    d = 1; #10;
    d = 0; #10;
    d = 1; #10;
    $finish;
  end
  always #5 clk = ~clk;
endmodule
```

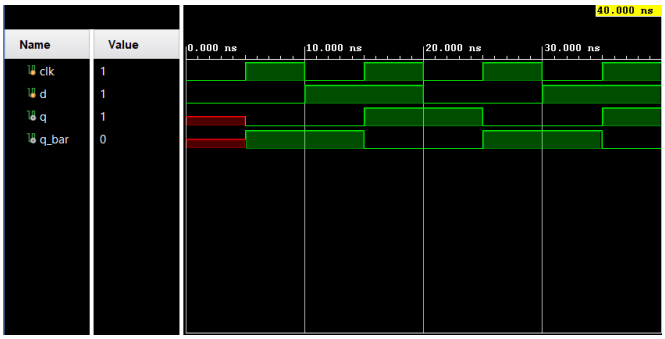

Fig. 42: D Flip-Flop Simulation

1.9 MOD 10 Counter

1.9.1 MOD 10 Asynchronous Counter

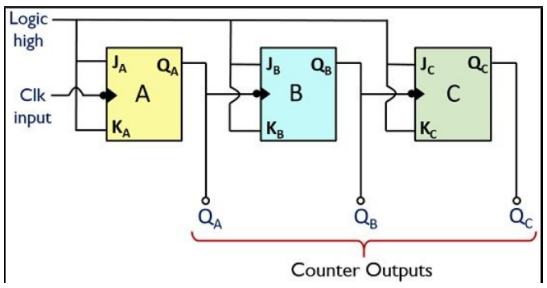

Fig. 43: MOD-10 Asynchronous Counter Circuit

Verilog Code:

```
`timescale 1ns / 1ps
```

```
module T_ff1(input t, input clk, input pr, input cr, output reg q, output q_bar);
  assign q_bar = ~q;

  always @(negedge clk or negedge pr or negedge cr) begin
    if (!cr)
      q = 0;
    else if (!pr)
      q = 1;
    else if (t)
      q <= ~q;
  end
endmodule

module mod10_async_counter(input clk1, input r, output [3:0] count);
  wire [3:0] q;
  wire internal_reset;

  // Active-low reset (0 when q == 10)
  assign internal_reset = ~(q == 4'd10);  // 0 when we want to clear

  wire cr = r & internal_reset;  // final cr input to T_ff1

  T_ff1 t0(.t(1), .clk(clk1),   .pr(1), .cr(cr), .q(q[0]), .q_bar());
  T_ff1 t1(.t(1), .clk(q[0]),   .pr(1), .cr(cr), .q(q[1]), .q_bar());
  T_ff1 t2(.t(1), .clk(q[1]),   .pr(1), .cr(cr), .q(q[2]), .q_bar());
  T_ff1 t3(.t(1), .clk(q[2]),   .pr(1), .cr(cr), .q(q[3]), .q_bar());

  assign count = q;
endmodule
```

Table 26: MOD-10 Asynchronous Counter Sequence

Clock Pulse	Count Output (Decimal)
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	0

Testbench:

```
`timescale 1ns / 1ps
module tb_mod10_async_counter();
  reg clk;
  reg r;
  wire [3:0] count;

  mod10_async_counter uut(.clk1(clk), .r(r), .count(count));

  initial begin
    clk = 0;
    r = 0; #5;
    r = 1;
    #200;
    $finish;
  end

  always #5 clk = ~clk;
```

```
endmodule
```

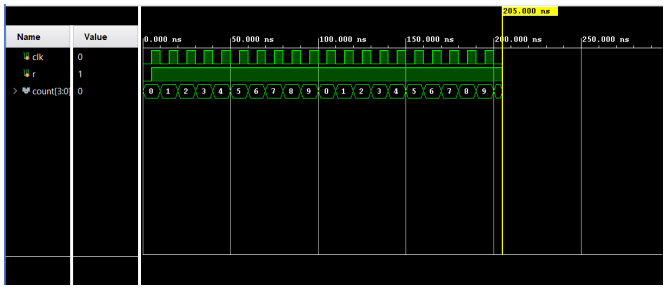

Fig. 44: MOD-10 Asynchronous Counter Simulation

1.9.2 MOD 10 Synchronous Counter

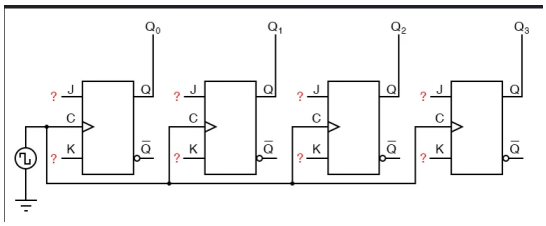

Fig. 45: MOD-10 Synchronous Counter Circuit

Verilog Code:

```
`timescale 1ns / 1ps

// 9. Mod-10 Synchronous Up-Counter
module mod10_sync_counter(input clk, input reset, output reg [3:0] count);
  always @(posedge clk or posedge reset) begin
    if (reset) count <= 0;
    else if (count == 9) count <= 0;
    else count <= count + 1;
  end
endmodule
```

Table 27: MOD-10 Synchronous Counter Sequence

Clock Pulse	Count Output (Decimal)
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	0

Testbench:

```
‘timescale 1ns / 1ps
module tb_mod10_sync_counter();
  reg clk, reset;
  wire [3:0] count;
  mod10_sync_counter uut(.clk(clk), .reset(reset), .count(count));
  initial begin
    clk = 0; reset = 1; #5;
    reset = 0; #100;
    $finish;
  end
  always #5 clk = ~clk;
endmodule
```

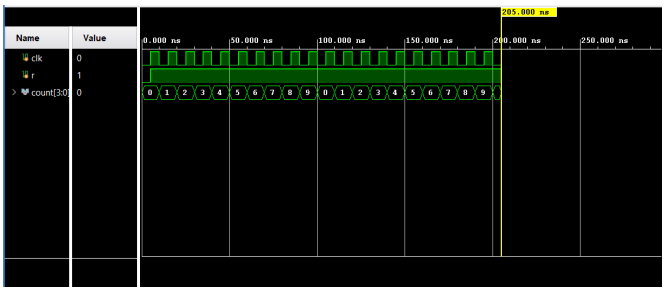

Fig. 46: MOD-10 Synchronous Counter Simulation

1.10 Ring and Johnson Counters

1.10.1 4-bit Ring Counter

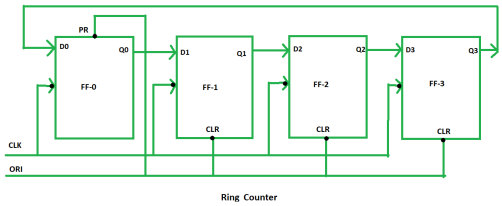

Fig. 47: 4-bit Ring Counter Circuit

Verilog Code:

```
‘timescale 1ns / 1ps
module ring_counter(input clk, input reset, output reg [3:0] out);
  always @(posedge clk or posedge reset) begin
    if (reset) out <= 4’b0001;
    else out <= {out[2:0], out[3]};
  end
endmodule
```

Table 28: 4-bit Ring Counter Sequence

Clock Pulse	Output (Binary)
0	1000
1	0001
2	0010
3	0100
4	1000
5	0001

Testbench:

```
‘timescale 1ns / 1ps
module tb_ring_counter();
  reg clk, reset;
  wire [3:0] out;
  ring_counter uut(.clk(clk), .reset(reset), .out(out));
  initial begin
    clk = 0; reset = 1; #5;
    reset = 0; #100;
    $finish;
  end
  always #5 clk = ~clk;
endmodule
```

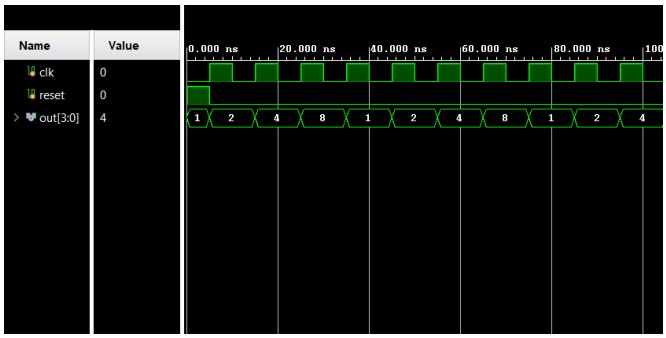

Fig. 48: 4-bit Ring Counter Simulation

1.10.2 4-bit Johnson Counter

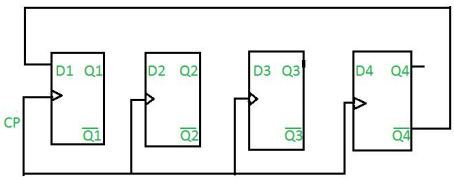

Fig. 49: 4-bit Johnson Counter Circuit

Verilog Code:

```
‘timescale 1ns / 1ps
module johnson_counter(input clk, input reset, output reg [3:0] out);
  always @(posedge clk or posedge reset) begin
    if (reset) out <= 4’b0000;
```

```
    else out <= {~out[0], out[3:1]};
  end
endmodule
```

Table 29: 4-bit Johnson Counter Sequence

Clock Pulse	Output (Binary)
0	0000
1	1000
2	1100
3	1110
4	1111
5	0111
6	0011
7	0001
8	0000

Testbench:

```
`timescale 1ns / 1ps
module tb_johnson_counter();
  reg clk, reset;
  wire [3:0] out;
  johnson_counter uut(.clk(clk), .reset(reset), .out(out));
  initial begin
    clk = 0; reset = 1; #5;
    reset = 0; #100;
    $finish;
  end
  always #5 clk = ~clk;
endmodule
```

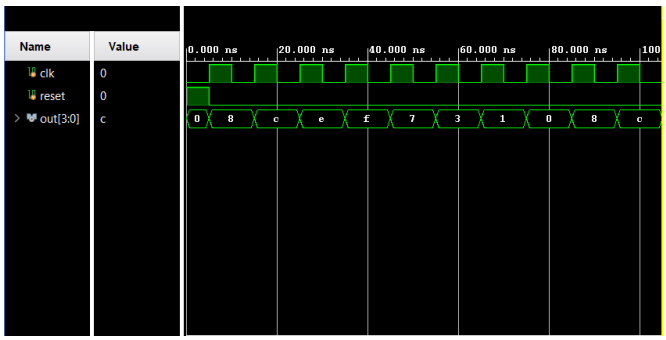

Fig. 50: 4-bit Johnson Counter Simulation

1.11 Universal Shift Register

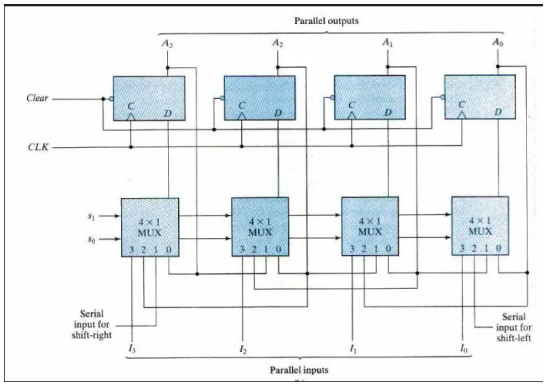

Fig. 51: 4-bit Universal Shift Register Circuit

Verilog Code:

```
‘timescale 1ns / 1ps
module universal_shift_register(input [1:0] sel, input [3:0] d, input clk, input reset, output reg [3:0]
    ↪ q);
  always @(posedge clk or posedge reset) begin
    if (reset) q <= 0;
    else case (sel)
      2’b00: q <= q;
      2’b01: q <= {q[2:0], d[0]};
      2’b10: q <= {d[3], q[3:1]};
      2’b11: q <= d;
    endcase
  end
endmodule
```

Table 30: Universal Shift Register Operations

Select	Operation	Description	Example (q = 1010)
00	Hold	No change	1010
01	Shift Right	q = S_in_R, q3, q2, q1	e.g., S_in_R = 1 → 1101
10	Shift Left	q = q2, q1, q0, S_in_L	e.g., S_in_L = 0 → 0100
11	Parallel Load	q = Parallel Input	e.g., P_in = 1111 → 1111

Testbench:

```
‘timescale 1ns / 1ps
module tb_universal_shift_register();
  reg [1:0] sel;
  reg [3:0] d;
  reg clk, reset;
  wire [3:0] q;
  universal_shift_register uut(.sel(sel), .d(d), .clk(clk), .reset(reset), .q(q));
  initial begin
    clk = 0; reset = 1; #5;
    reset = 0; sel = 2’b11; d = 4’b1010; #10;
    sel = 2’b01; d = 4’b0001; #10;
    sel = 2’b10; #10;
    sel = 2’b00; #10;
    $finish;
  end
  always #5 clk = ~clk;
endmodule
```

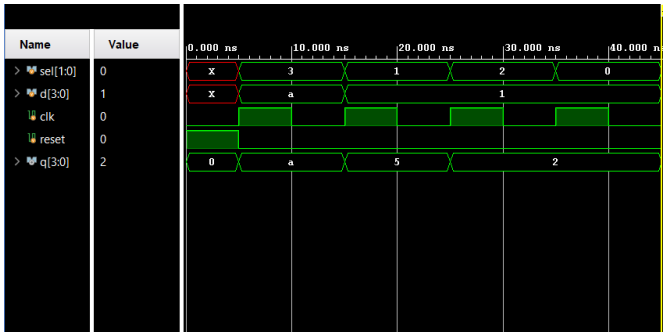

Fig. 52: 4-bit Universal Shift Register Simulation

1.12 Sequence Detectors for 10011 and 10101

1.12.1 1. Mealy Machine Sequence Detector for 10011 (Overlapping)

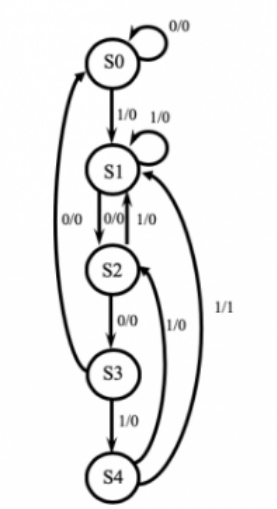

Fig. 53: Mealy FSM for sequence 10011 (Overlapping)

Verilog Code:

```
module mealy_10011_ov(input clk, input rst, input in, output reg out);
  reg [2:0] state, next;
  parameter S0=0, S1=1, S2=2, S3=3, S4=4;
  always @(posedge clk or posedge rst) begin
    if (rst) state <= S0;
    else state <= next;
  end
  always @(*) begin
    out = 0;
    case (state)
      S0: next = in ? S1 : S0;
      S1: next = in ? S1 : S2;
      S2: next = in ? S3 : S0;
      S3: next = in ? S1 : S4;
      S4: begin next = in ? S1 : S0; out = in; end
    endcase
  end
endmodule
```

2. Moore Machine Sequence Detector for 10011 (Overlapping)

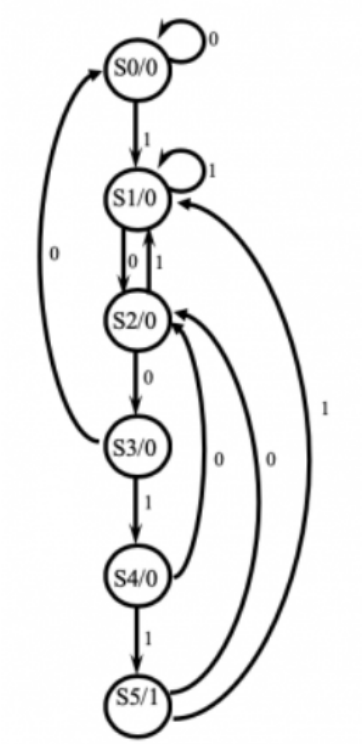

Fig. 54: Moore FSM for sequence 10011 (Overlapping)

Verilog Code:

```
‘timescale 1ns / 1ps
module moore_10011_ov(input clk, input rst, input in, output reg out);
  reg [2:0] state, next;
  parameter S0=0, S1=1, S2=2, S3=3, S4=4, S5=5;
  always @(posedge clk or posedge rst) begin
    if (rst) state <= S0;
    else state <= next;
  end
  always @(*) begin
    case (state)
      S0: next = in ? S1 : S0;
      S1: next = in ? S1 : S2;
      S2: next = in ? S3 : S0;
      S3: next = in ? S1 : S4;
      S4: next = in ? S5 : S0;
      S5: next = in ? S1 : S0;
    endcase
  end
  always @(*) begin
    out = (state == S5);
  end
endmodule
```

1.12.2 Mealy Machine Sequence Detector for 10011 (NON Overlapping)

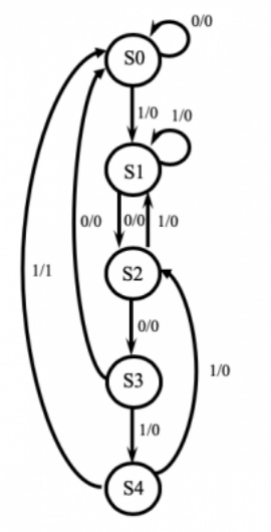

Fig. 55: Mealy FSM for sequence 10011 (Non Overlapping)

Verilog Code:

```
module mealy_10011_nov(input clk, input rst, input in, output reg out);
  reg [2:0] state, next;
  parameter S0=0, S1=1, S2=2, S3=3, S4=4;
  always @(posedge clk or posedge rst) begin
    if (rst) state <= S0;
    else state <= next;
  end
  always @(*) begin
    out = 0;
    case (state)
      S0: next = in ? S1 : S0;
      S1: next = in ? S1 : S2;
      S2: next = in ? S3 : S0;
      S3: next = in ? S1 : S4;
      S4: begin next = in ? S0 : S0; out = in; end
    endcase
  end
endmodule
```

Moore Machine Sequence Detector for 10011 (NON Overlapping)

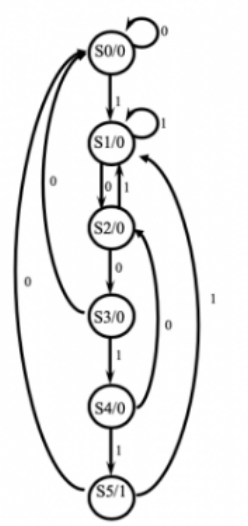

Fig. 56: Moore FSM for sequence 10011 (Non Overlapping)

Verilog Code:

```
module moore_10011_nov(input clk, input rst, input in, output reg out);
  reg [2:0] state, next;
  parameter S0=0, S1=1, S2=2, S3=3, S4=4, S5=5;
  always @(posedge clk or posedge rst) begin
    if (rst) state <= S0;
    else state <= next;
  end
  always @(*) begin
    case (state)
      S0: next = in ? S1 : S0;
      S1: next = in ? S1 : S2;
      S2: next = in ? S3 : S0;
      S3: next = in ? S1 : S4;
      S4: next = in ? S5 : S0;
      S5: next = in ? S0 : S0;
    endcase
  end
  always @(*) begin
    out = (state == S5);
  end
endmodule
```

1.12.3 Mealy Machine Sequence Detector for 10101 (Overlapping)

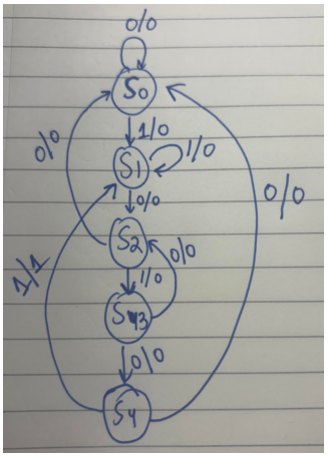

Fig. 57: Mealy FSM for sequence 10101 (Overlapping)

Verilog Code:

```
module mealy_10101_ov(input clk, input rst, input in, output reg out);
  reg [2:0] state;
  parameter S0=0, S1=1, S2=2, S3=3, S4=4;

  always @(posedge clk or posedge rst) begin
    if (rst)
      state <= S0;
    else begin
      case (state)
        S0: state <= (in) ? S1 : S0;
        S1: state <= (in) ? S1 : S2;
        S2: state <= (in) ? S3 : S0;
        S3: state <= (in) ? S1 : S4;
        S4: state <= (in) ? S1 : S0;
      endcase
    end
  end

  always @(*) begin
    case (state)
      S4: out = (in) ? 1 : 0;
      default: out = 0;
    endcase
  end
endmodule
```

1.12.4 Mealy Machine Sequence Detector for 10101 (Non Overlapping)

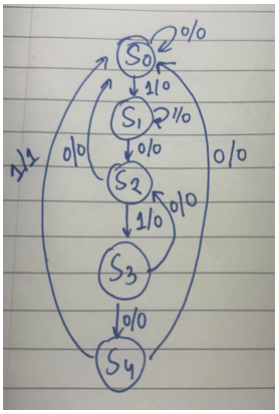

Fig. 58: Mealy FSM for sequence 10101 (Non overlapping)

Verilog Code:

```
module mealy_10101_nov(input clk, input rst, input in, output reg out);
  reg [2:0] state, next;
  parameter S0=0, S1=1, S2=2, S3=3, S4=4;
  always @(posedge clk or posedge rst) begin
    if (rst) state <= S0;
    else state <= next;
  end
  always @(*) begin
    out = 0;
    case (state)
      S0: next = in ? S1 : S0;
      S1: next = in ? S1 : S2;
      S2: next = in ? S3 : S0;
      S3: next = in ? S4 : S2;
      S4: begin next = in ? S0 : S0; out = ~in; end
    endcase
  end
endmodule
endmodule
```

1.12.5 Moore Machine Sequence Detector for 10101 (Overlapping)

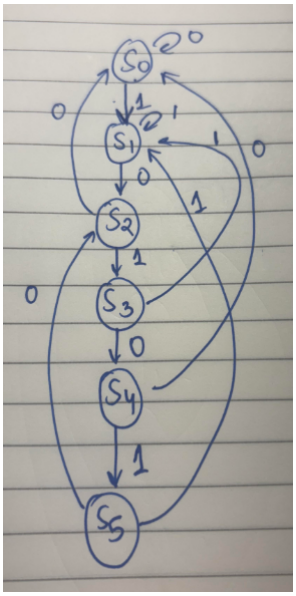

Fig. 59: Moore FSM for sequence 10101 (Overlapping)

Verilog Code:

```
`timescale 1ns / 1ps
module moore_10101_ov(input clk, input rst, input in, output reg out);
  reg [2:0] state, next;
  parameter S0=0, S1=1, S2=2, S3=3, S4=4, S5=5;
  always @(posedge clk or posedge rst) begin
    if (rst) state <= S0;
    else state <= next;
  end
  always @(*) begin
    case (state)
      S0: next = in ? S1 : S0;
      S1: next = in ? S1 : S2;
      S2: next = in ? S3 : S0;
      S3: next = in ? S4 : S2;
      S4: next = in ? S5 : S0;
      S5: next = in ? S1 : S0;
    endcase
  end
  always @(*) begin
    out = (state == S5);
  end
endmodule
```

1.12.6 Moore Machine Sequence Detector for 10101 (Non Overlapping)

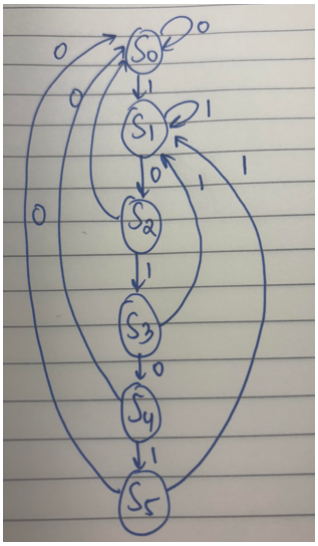

Fig. 60: Moore FSM for sequence 10101 (Non Overlapping)

Verilog Code:

```
`timescale 1ns / 1ps
module moore_10101_nov(
  input clk,
  input reset,
  input in,
  output reg out
);

  typedef enum reg [2:0] {
    S0, S1, S2, S3, S4, S5
  } state_t;

  state_t state, next_state;

  always @(posedge clk or posedge reset) begin
    if (reset)
      state <= S0;
    else
      state <= next_state;
  end

  always @(*) begin
    case (state)
      S0: next_state = (in == 1) ? S1 : S0;
      S1: next_state = (in == 0) ? S2 : S1;
      S2: next_state = (in == 1) ? S3 : S0;
      S3: next_state = (in == 0) ? S4 : S1;
      S4: next_state = (in == 1) ? S5 : S0;
      S5: next_state = (in == 1) ? S1 : S0;
      default: next_state = S0;
    endcase
  end

  always @(*) begin
    out = (state == S5) ? 1 : 0;
  end
endmodule
```

1.13 Serial Moore and Mealy Adders

Moore Machine Serial Adder

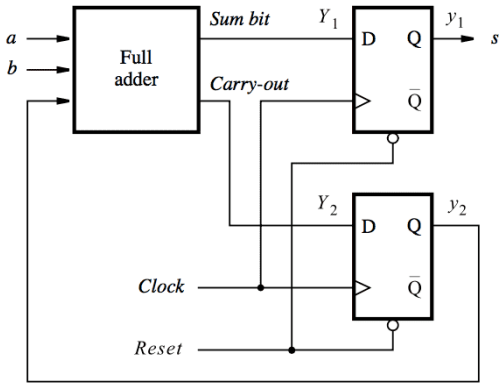

Fig. 61: Serial Adder using Moore Machine

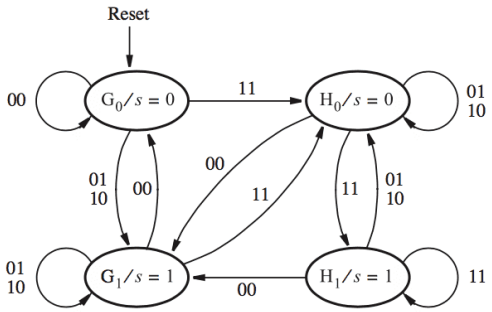

Fig. 62: Serial Adder using Moore Machine

Verilog Code:

```
‘timescale 1ns / 1ps
module serial_adder_moore(
  input clk, rst, a_bit, b_bit,
  output reg sum, carry_out
);
  reg carry;

  always @(posedge clk or posedge rst) begin
    if (rst) begin
      carry <= 0;
      sum <= 0;
    end else begin
      sum <= a_bit ^ b_bit ^ carry;
      carry <= (a_bit & b_bit) | (a_bit & carry) | (b_bit & carry);
    end
  end

  always @(posedge clk)
    carry_out <= carry;
endmodule
```

Table 31: Moore Serial Adder Operation

A	B	Carry In	Sum	Carry Out
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Testbench:

```
`timescale 1ns / 1ps
module tb_serial_adder_moore();

  reg clk, rst, a_bit, b_bit;
  wire sum, carry_out;

  serial_adder_moore uut(
    .clk(clk),
    .rst(rst),
    .a_bit(a_bit),
    .b_bit(b_bit),
    .sum(sum),
    .carry_out(carry_out)
  );

  // Clock generation
  initial begin
    clk = 0;
    forever #5 clk = ~clk;
  end

  // Stimulus
  initial begin
    rst = 1; #10;
    rst = 0;

    // Test input bits: A = 1011, B = 1101 (LSB first)
    a_bit = 1; b_bit = 1; #10;
    a_bit = 1; b_bit = 0; #10;
    a_bit = 0; b_bit = 1; #10;
    a_bit = 1; b_bit = 1; #10;

    $finish;
  end

endmodule
```

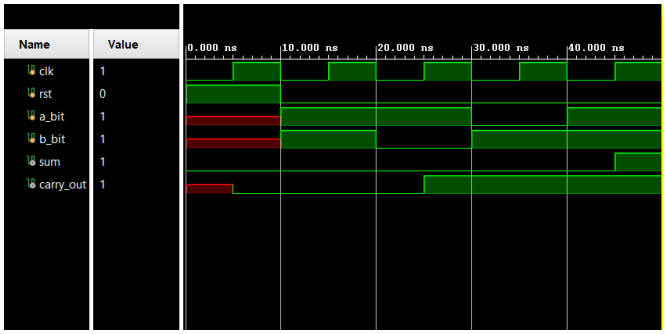

Fig. 63: Moore Serial Adder Simulation

Mealy Machine Serial Adder

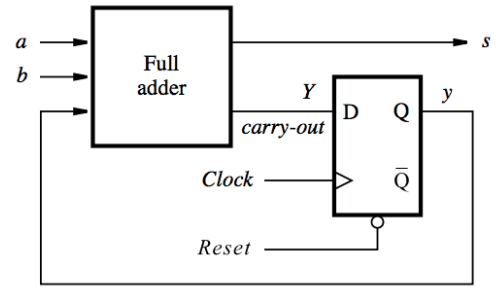

Fig. 64: Serial Adder using Mealy Machine

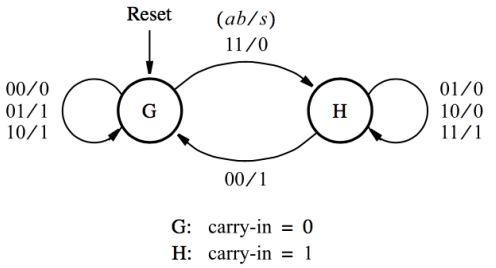

Fig. 65: Serial Adder using Mealy Machine

Verilog Code:

```
`timescale 1ns / 1ps
module serial_adder_mealy(
  input clk, rst, a_bit, b_bit,
  output reg sum, carry_out
);
  reg carry;

  always @(posedge clk or posedge rst) begin
    if (rst)
      carry <= 0;
    else begin
      sum <= a_bit ^ b_bit ^ carry;
      carry <= (a_bit & b_bit) | (a_bit & carry) | (b_bit & carry);
    end
  end

  always @(*) begin
    carry_out = carry;
  end
endmodule
```

Table 32: Mealy Serial Adder Operation

A	B	Carry In	Sum	Carry Out
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Testbench:

```
`timescale 1ns / 1ps
module tb_serial_adder_mealy();

  reg clk, rst, a_bit, b_bit;
  wire sum, carry_out;

  serial_adder_mealy uut(
    .clk(clk),
    .rst(rst),
    .a_bit(a_bit),
    .b_bit(b_bit),
    .sum(sum),
    .carry_out(carry_out)
  );

  // Clock generation
  initial begin
    clk = 0;
    forever #5 clk = ~clk;
  end

  // Stimulus
  initial begin
    rst = 1; #10;
    rst = 0;

    // Test input bits: A = 1011, B = 1101 (LSB first)
    a_bit = 1; b_bit = 1; #10;
    a_bit = 1; b_bit = 0; #10;
    a_bit = 0; b_bit = 1; #10;
    a_bit = 1; b_bit = 1; #10;

    $finish;
  end

endmodule
```

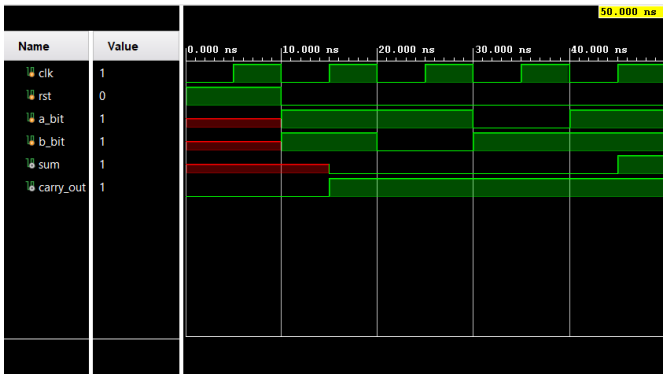

Fig. 66: Mealy Serial Adder Simulation