IoT workshop on Digital IC/FPGA

Dr. Muhammad Fahim Ul Haque

Design Requirement

- Mandatory Requirement
  - Functionality
- Cost
  - Power consumption
  - Size
  - Speed
- Design should fulfill mandatory requirement.

System Description

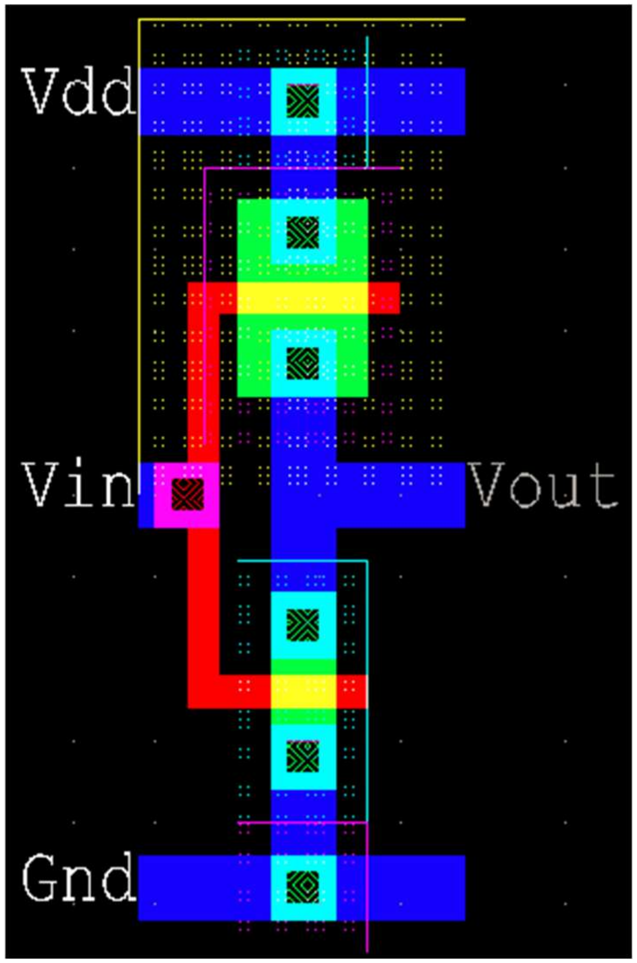
- Behavioral (RTL)

  - Input and output behaviour
- Functional (Data flow)

  - Boolean expression
- Structural (Gate level Description)

  - Gate level schematic
- Physical

  - Detail Layout



Hardware Description Language

- HDLs are popular mode of design entry.
- Popular HDLs are

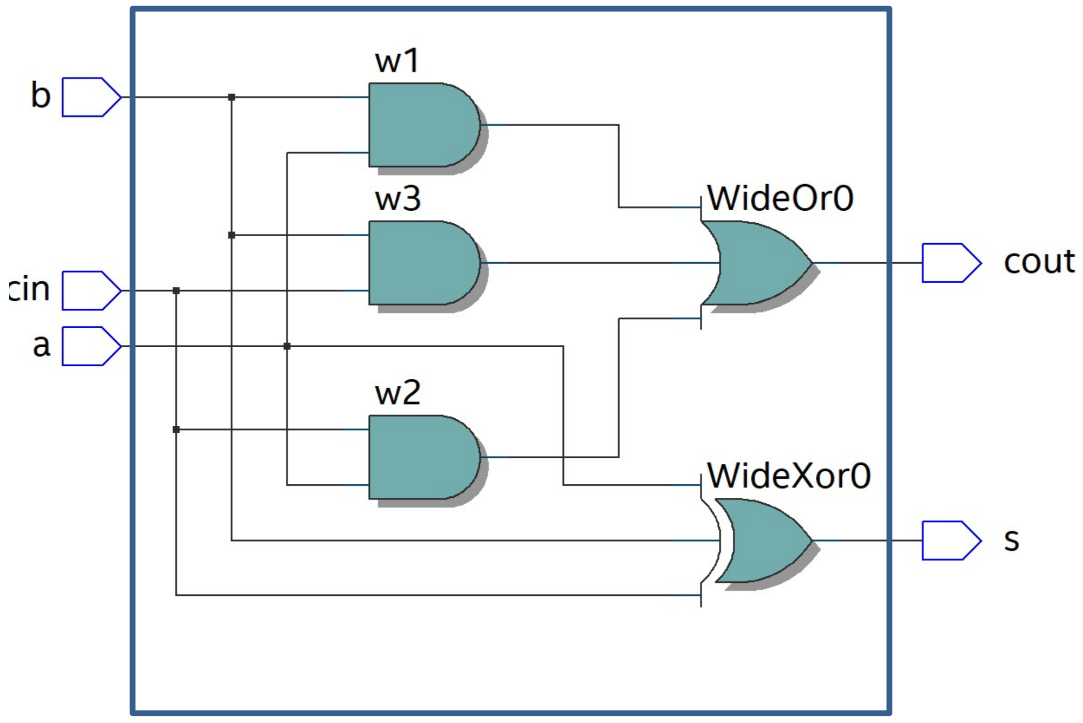
  - VHDL

  -Verilog HDL

  -System Verilog
- Both HDLs can describe digital systems at several different levels –**behavioral**, **data flow**, and **structural**.

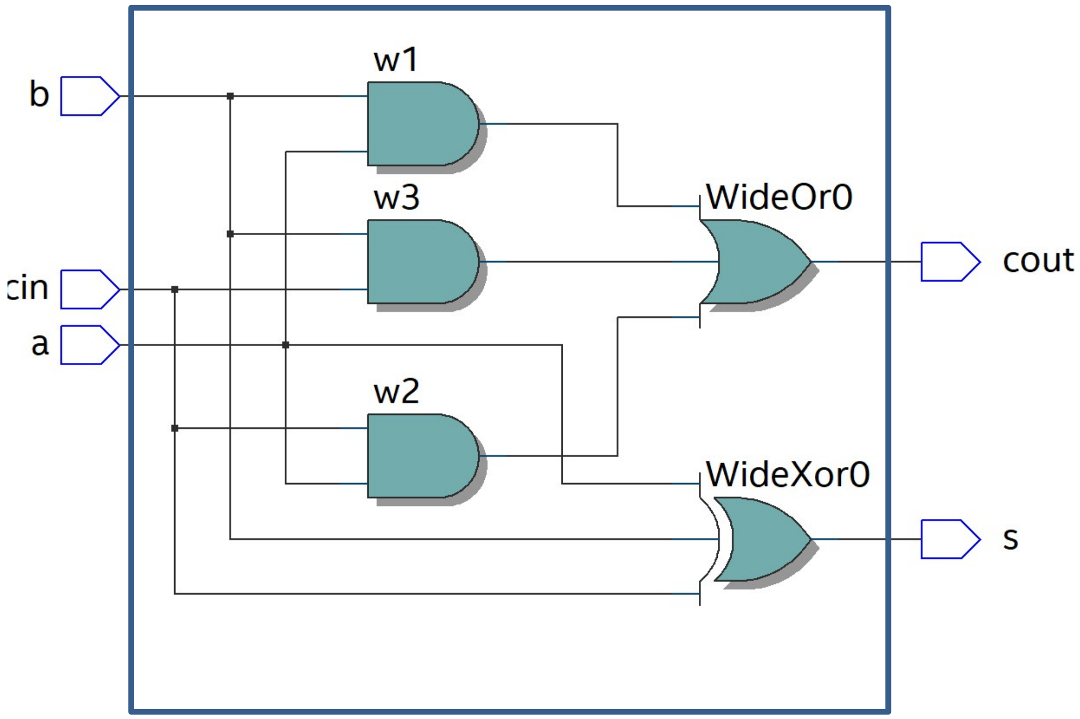
Full Adder (Structural Code)

```
module FullAdder (s,cout,a,b,cin);
output s, cout;
input a,b,cin;
wire w1, w2, w3;
//Sum Circuit
xor(s,a,b,cin);
//Cout Circuit
and(w1,a,b);
and(w2,a,cin);
and(w3,b,cin);
or(cout,w1,w2,w3);
endmodule
```



Full Adder (Functional)

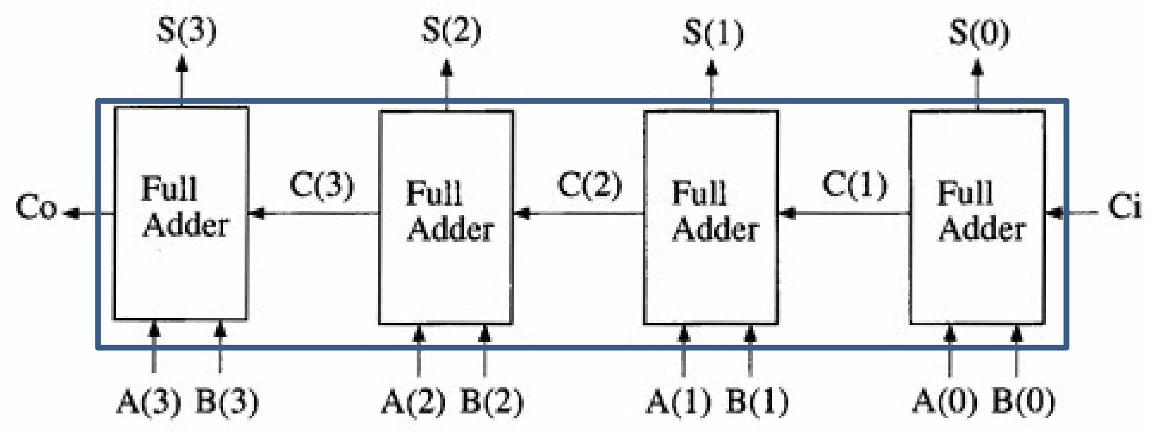
```
module fulladder (s,cout,a,b,cin);
    output s, cout;
    input a,b,cin;
    assign s = a^b^cin;
    assign cout = (a & b) | (a & cin)|(b & cin);
endmodule
```



Four bit Adder (Structural)

```
module adder4(sum,cout,a,b,cin);
output[3:0] sum;
output cout;
input[3:0] a,b;
input cin;
wire[2:0] c;
fulladder fa0 (.s(sum[0]) , .cout(c[0]) , .a(a[0]) , .b(b[0]) , .cin(cin));
fulladder fa1 (.s(sum[1]) , .cout(c[1]) , .a(a[1]) , .b(b[1]) , .cin(c[0]));
fulladder fa2 (.s(sum[2]) , .cout(c[2]) , .a(a[2]) , .b(b[2]) , .cin(c[1]));
fulladder fa3 (.s(sum[3]) , .cout(cout) , .a(a[3]) , .b(b[3]) , .cin(c[2]));
endmodule
```

Four bit Adder (Structural)



```
module adder4(sum,cout,a,b,cin);
output[3:0] sum;
output cout;
input[3:0] a,b;
input cin;
wire[2:0] c;
fulladder fa0 (.s(sum[0]) , .cout(c[0]) , .a(a[0]) , .b(b[0]) , .cin(cin));
fulladder fa1 (.s(sum[1]) , .cout(c[1]) , .a(a[1]) , .b(b[1]) , .cin(c[0]));
fulladder fa2 (.s(sum[2]) , .cout(c[2]) , .a(a[2]) , .b(b[2]) , .cin(c[1]));
fulladder fa3 (.s(sum[3]) , .cout(cout) , .a(a[3]) , .b(b[3]) , .cin(c[2]));
endmodule
```

Four bit Adder (Behavioral)

```
module adder_behave #(parameter width = 8)(sum,cout,a,b,cin);
    output [width-1:0] sum;
    output cout;
    input [width-1:0] a,b;
    input cin;
    assign {cout,sum} = a + b + cin;
endmodule
```

MUX (Behavioral)

```
module mux2to1#(parameter width = 8)(f,I0,I1,sel);
    output [width-1:0] f;
    input [width-1:0] I0,I1;
    input sel;
    assign f = sel? I1:I0;
endmodule
```

ALU (Behavioral)

```
module ALU#(parameter width = 4)(result,cout,a,b,cont);
    output [width-1:0] result;
    output cout;
    input [width-1:0] a, b;
    input [1:0]cont;
    wire [width-1:0] b_bar, mux2to1_out, sum, and_N_out, or_N_out;
    assign and_N_out = a & b;
    assign or_N_out = a | b;
    assign b_bar = ~b;
    mux2to1#(width)mux1(mux2to1_out,b,b_bar,cont[0]);
    adder#(width) add1(sum,cout,a,mux2to1_out,cont[0]);
    mux4to1 #(width)mux2(result,or_N_out,and_N_out,sum,sum,cont);
endmodule
```

Process

- Initial Process

```
initial
begin
        sequential statements

end
```

- Sequential process

```
always @ (sensitivity-list)
begin
        sequential statements
end
```

Process

```
module sequential_module (A, B, C, D, clk);
input  clk;
output A, B, C, D;
reg    A, B, C, D;

always @(posedge clk)
begin
   A = B;
   B = A;
end

always @(posedge clk)
begin
   C <= D;
   D <= C;
end

endmodule
```

Assignments

- Type of Assignments
  - Continuous Assignments
    - Explicit Assignment
    - Implicit Assignment
  - Procedural Assignments
    - Blocking Assignment
    - Non-blocking Assignment

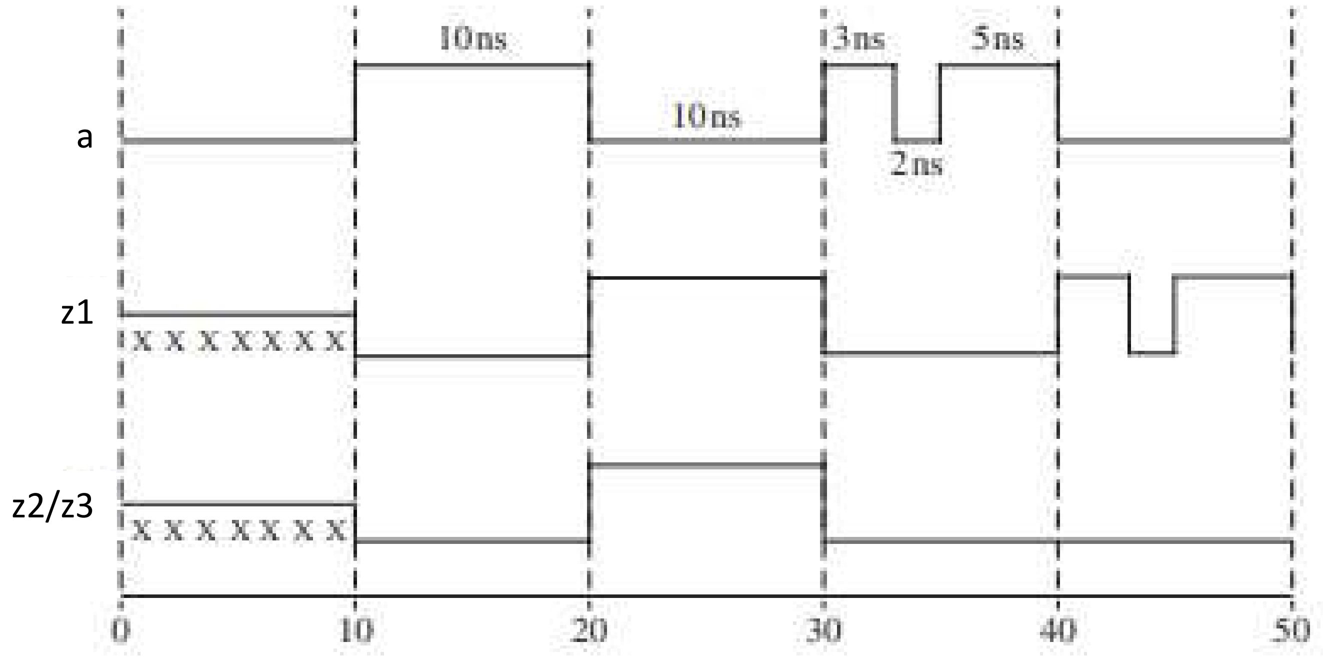
Procedural Assignments

```
module sequential_module (A, B, C, D, clk);
input  clk;
output A, B, C, D;
reg    A, B, C, D;

always @(posedge clk)
begin
  A = B;      // blocking statement 1
  B = A;      // blocking statement 2
end

always @(posedge clk)
begin
  C <= D;     // non-blocking statement 1
  D <= C;     // non-blocking statement 2
end

endmodule
```

Delay

- Inertial Delay

- Transport Delay



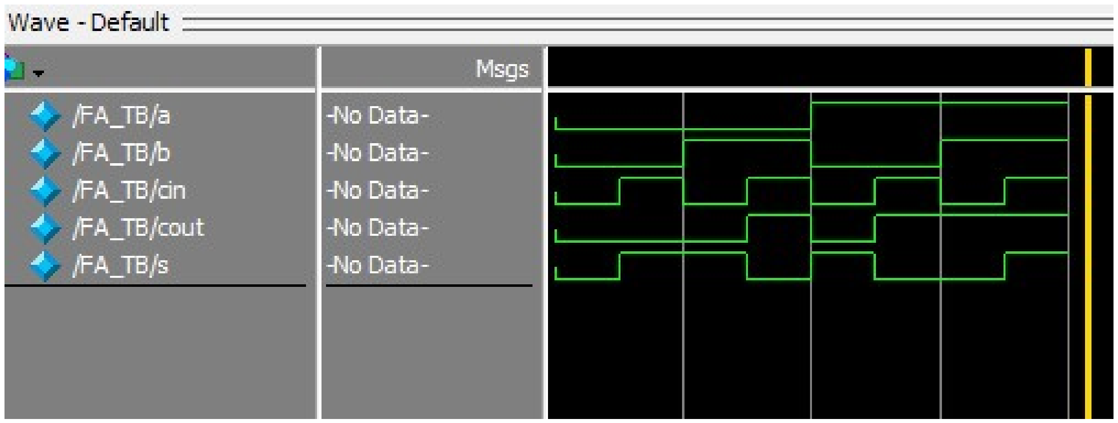
```
`timescale 1ns/1ps
module delay (z1, z2, z3, a);
output z1,z2,z3;
output z4;
input a;
reg z1,z2;
// Inertial Delay
assign #10 z3 = a;
always@(*)
Begin
// Inertial Delay in process
            #10 z2 <= a;
end
// Transport Delay
always@(a)
begin
            z1 <= #10 (a);
end
endmodule
```

Test Benches

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

```
module FA_TB();
reg a,b,cin;
wire s,cout;
// Applying test signal from test bench
initial
begin
a = 0; b = 0; cin = 0; #10;
a = 0; b = 0; cin = 1; #10;
a = 0; b = 1; cin = 0; #10;
a = 0; b = 1; cin = 1; #10;
a = 1; b = 0; cin = 0; #10;
a = 1; b = 0; cin = 1; #10;
a = 1; b = 1; cin = 0; #10;
a = 1; b = 1; cin = 1;#10;
end
//Instantiating Fulladder module
FA dut(s,cout,a,b,cin);
endmodule
```

Test Benches

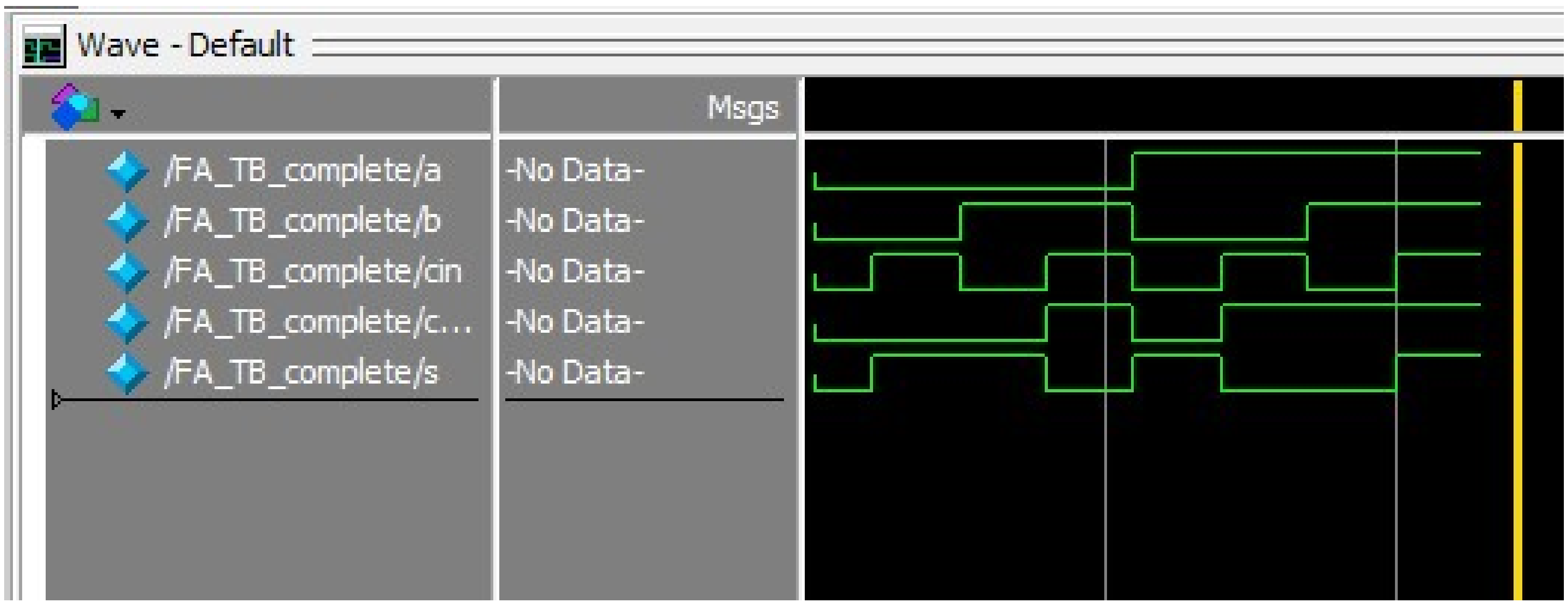


```
module FA_TB();
reg a,b,cin;
wire s,cout;
// Applying test signal from test bench
initial
begin
a = 0; b = 0; cin = 0; #10;
a = 0; b = 0; cin = 1; #10;
a = 0; b = 1; cin = 0; #10;
a = 0; b = 1; cin = 1; #10;
a = 1; b = 0; cin = 0; #10;
a = 1; b = 0; cin = 1; #10;
a = 1; b = 1; cin = 0; #10;
a = 1; b = 1; cin = 1;#10;
end
//Instantiating Fulladder module
FA dut(s,cout,a,b,cin);
endmodule
```

Test Benches

```
module FA_TB_complete();
reg a,b,cin; wire s, cout;
initial
begin
a = 0; b = 0; cin = 0; #5;
assert ((s===0)&&(cout===0)) else $error("000 failed");
#10; a = 0; b = 0; cin = 1; #5;
assert ((s===1)&&(cout===0)) else $error("001 failed");
#10; a = 0; b = 1; cin = 0; #5;
assert ((s===1)&&(cout===0)) else $error("010 failed");
#10; a = 0; b = 1; cin = 1; #5;
assert ((s===0)&&(cout===1)) else $error("011 failed");
#10; a = 1; b = 0; cin = 0; #5;
assert ((s===1)&&(cout===0)) else $error("100 failed");
#10; a = 1; b = 0; cin = 1; #5;
assert ((s===0)&&(cout===1)) else $error("101 failed");
#10; a = 1; b = 1; cin = 0; #5;
assert ((s===0)&&(cout===1)) else $error("110 failed");
#10; a = 1; b = 1; cin = 1; #5;
assert ((s===1)&&(cout===1)) else $error("111 failed"); #10;
end
FA dut(s,cout,a,b,cin);
endmodule
```

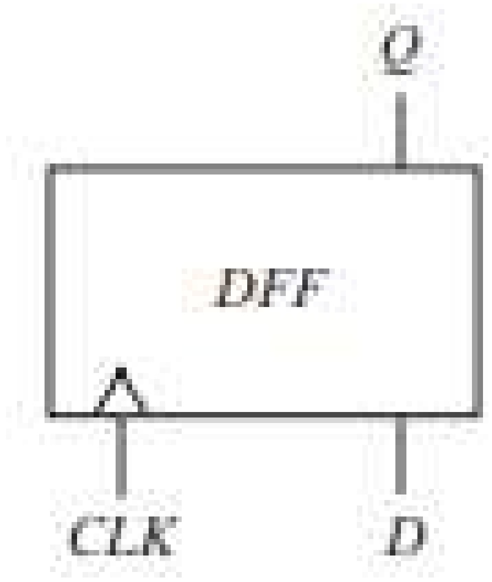
Test Benches



```
/FA_TB_complete/a     -No Data-
/FA_TB_complete/b     -No Data-
/FA_TB_complete/cin   -No Data-
/FA_TB_complete/c...  -No Data-
/FA_TB_complete/s     -No Data-
```

Sequential Circuits (D Flip Flop)

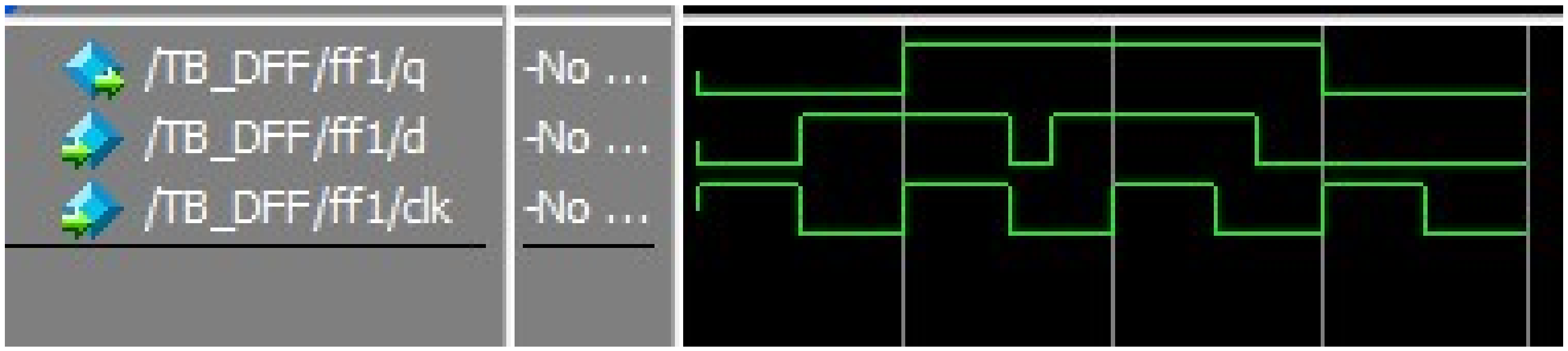
```
module DFF(q,d,clk);
output q;
input d, clk;
reg q;
always @(posedge clk)
begin
q <= d;
end
endmodule
```



Sequential Circuits (D Flip Flop TB)

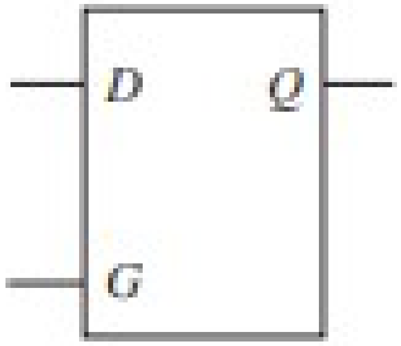
```
`timescale 1ns/1ps
module TB_DFF();
logic clk, d, q;
initial
begin
d = 0; #5;
d = 1; #10;
d = 0; #2;
d = 1; #10;
d = 0;
end
always
begin
clk =1; #5;
clk =0; #5;
end
```

```
initial
begin
#5; assert(q === 0) else $error("1 cycle failed");
#10; assert(q === 1) else $error("2 cycle failed");
#10; assert(q === 1) else $error("3 cycle failed");
#10; assert(q === 0) else $error("4 cycle failed"); #5;
$stop;
end
DFF ff1(q,d,clk);
endmodule
```



Sequential Circuits (D Latch)

```
module Dlatch(q,d,clk);
output q;
input d,clk;
reg q;
always @(clk,d)
begin
if (clk)
q <= d;
end
endmodule
```

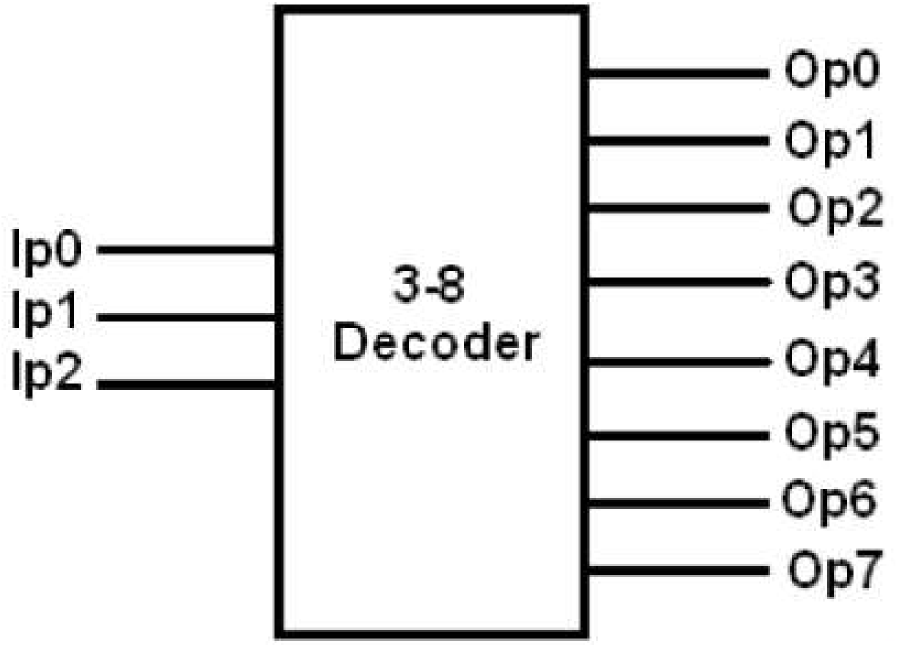


Combinational Ckt through process

```
module transmission_gate(y,x,en);
output y;
input x,en;
reg y;
always @(x,en)
begin
if (en)
y = x;
else
y = 1'bz;
end
endmodule
```

Combinational Ckt (Decoder/Encoder)

```
module bin_decoder(a,z);
input [2:0] a;
output [7:0] z;
reg [7:0] z;
always @(*)
begin
case(a)
3'b000: z = 8'b00000001;
3'b001: z = 8'b00000010;
3'b010: z = 8'b00000100;
3'b011: z = 8'b00001000;
3'b100: z = 8'b00010000;
3'b101: z = 8'b00100000;
3'b110: z = 8'b01000000;
3'b111: z = 8'b10000000;
endcase
end
endmodule
```

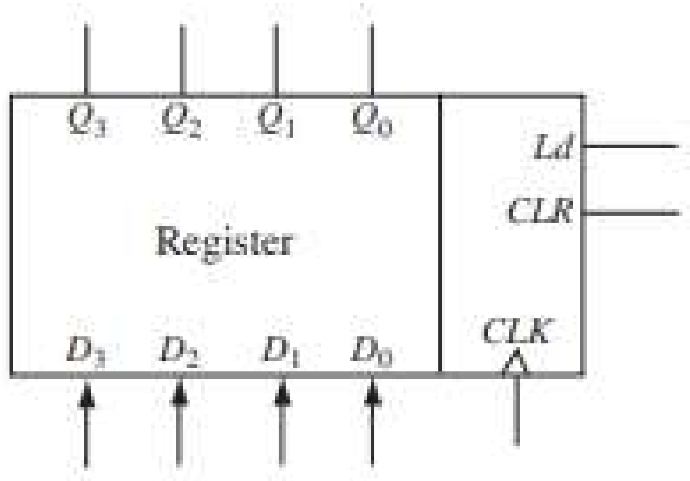


Combinational Ckt (Decoder TB)

```
module tb_bin_decoder();
logic clk, reset;
logic [2:0] a;
logic [7:0] z;
logic [7:0] z_expected;
logic [2:0] iter, error;
logic [11:0] testvector[7:0];
bin_decoder decode1(a,z);
always
begin
clk = 1; #5;
clk = 0; #5;
end
initial
begin
$readmemb("test_vector.txt",testvector);
iter = 0;
error = 0;
reset = 1; #22;
reset = 0;
end
always@(posedge clk)
begin
{a,z_expected} = testvector[iter];
end
// checking result on falling edge of clock
always @(negedge clk)
begin
if (~reset)
begin
if (z !== z_expected)
begin
$display("Error: input =%b", a);
$display("Error: output = %b(%b expected)",z,z_expected);
error = error + 1;
end
if (iter === 3'b111)
begin
$display("%d total error",error);
$stop;
end
iter = iter + 1;
end
end
endmodule
```

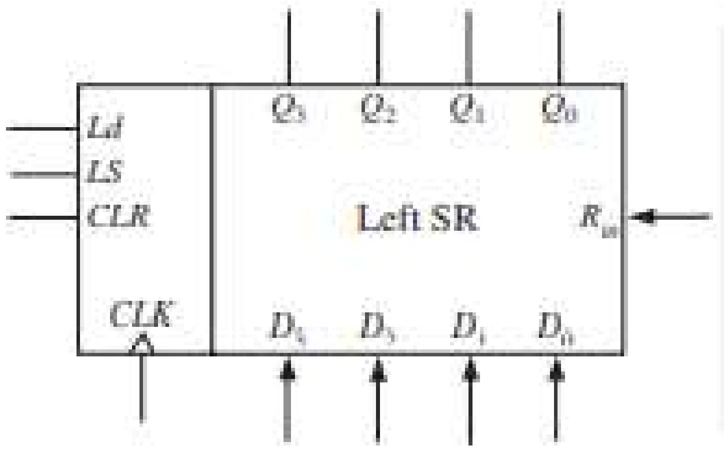
Sequential Circuits (Register)

```
module register #(parameter width = 4)(q,d,clr,ld,clk);
output reg [width-1:0] q;
input [width-1:0] d;
input clr,ld,clk;
always @(posedge clk)
begin
if (clr)
q <= {width{1'b0}};
else if (ld)
q <= d;
end
endmodule
```



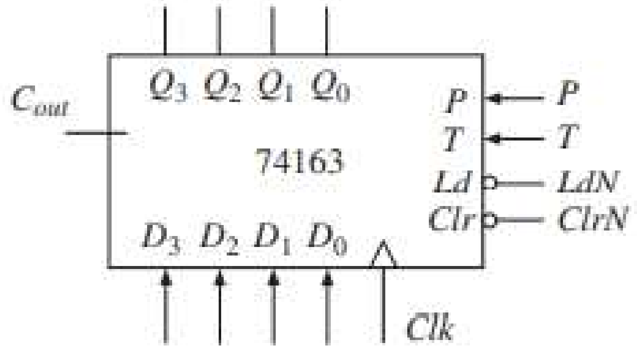
Sequential Circuits (Shift Register)

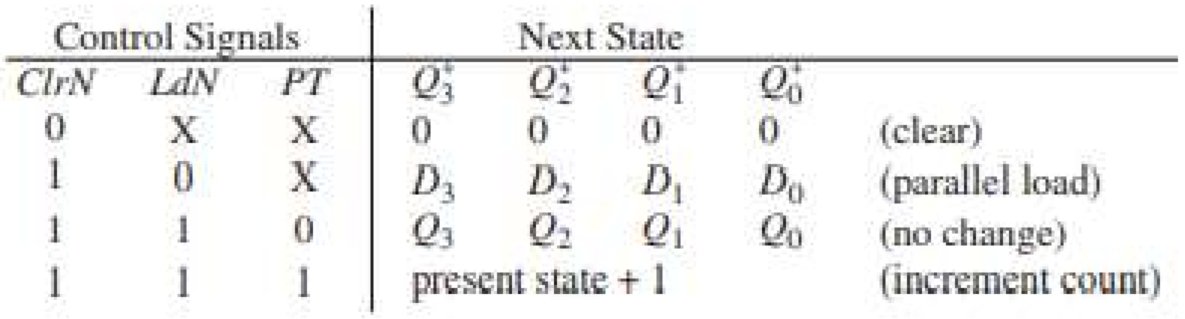
```
module shift_register
#(parameter width = 4)
(q,d,clr,ls,ld,rin,clk) ;
output reg [width-1:0] q;
input [width-1:0] d;
input clr, ls, ld, rin, clk;
always @(posedge clk, posedge clr)
if(clr)
q <= {width{1'b0}};
else if(ld)
q <= d;
else if(ls)
q <= {q[width-2:0],rin};
endmodule
```



Sequential Circuits (Counter)

```
module c74163(LdN, ClrN, P, T, Clk, D, Cout, Qout);
input LdN;
input ClrN;
input P;
input T;
input Clk;
input [3:0] D;
output Cout;
output [3:0] Qout;
reg [3:0] Q;
assign Qout = Q;
assign Cout = Q[3] & Q[2] & Q[1] & Q[0] & T;
always @(posedge Clk)
begin
 if (~ClrN) Q <= 4'b0000;
 else if (~LdN) Q <= D;
 else if (P & T) Q <= Q 1 1;
end
endmodule
```





Control Signals			Next State				
ClrN	*LdN*	*PT*	Q_3^+	Q_2^+	Q_1^+	Q_0^+	
0	X	X	0	0	0	0	(clear)
1	0	X	D_3	D_2	D_1	D_0	(parallel load)
1	1	0	Q_3	Q_2	Q_1	Q_0	(no change)
1	1	1	present state + 1				(increment count)

Clock Divider

```
module clk_divider(clk, clk_50);
input clk_50;
output reg clk;
reg [31:0] count;
initial
count = 32'h00000000;
//reg count;
always@(posedge (clk_50))
begin
if (count >= 32'd50000000)
count <= 32'h00000000;
else
begin
if (count<=32'd25000000)
clk <= 1;
else
clk <= 0;
count <= count + 1;
end
end
endmodule
```

RAM

```
module ram #(parameter N =6, M=32)(clk, we, adr, din, dout);
input logic clk, we;
input logic [N-1:0] adr;
input logic [M-1:0] din;
output logic [M-1:0] dout;
logic [M-1:0] mem [2**N-1:0];
always @ (posedge clk)
begin
if(we)
mem[adr] <= din;
end
assign dout = mem[adr];

endmodule
```

ROM/Combinational Ckt through ROM

```
module parity_gen(X, Y);
input [3:0] X;
output [4:0] Y;
wire ParityBit;
parameter [0:15] OT = {1'b1, 1'b0, 1'b0, 1'b1, 1'b0, 1'b1, 1'b1, 1'b0, 1'b0, 1'b1, 1'b1,
      1'b0, 1'b1, 1'b0, 1'b0, 1'b1};
 assign ParityBit = OT[X];
assign Y = {X, ParityBit};
endmodule
```

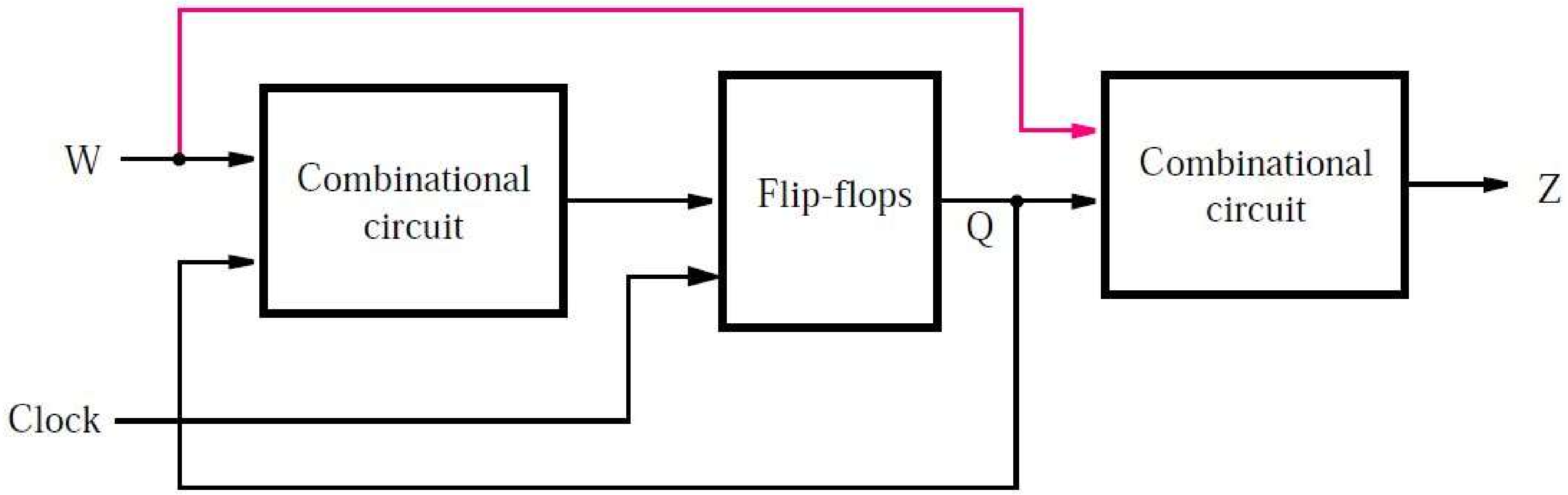
ROM/Combinational Ckt through ROM

```
module parity_gen(X,Y);
input [2:0] X;
output [3:0] Y;
parameter [3:0] OT[7:0] =
{5'b1000,
5'b0001,
5'b0010,
5'b1011,
5'b0100,
5'b1101,
5'b1110,
5'b0111};
assign Y = OT[X];
endmodule
```

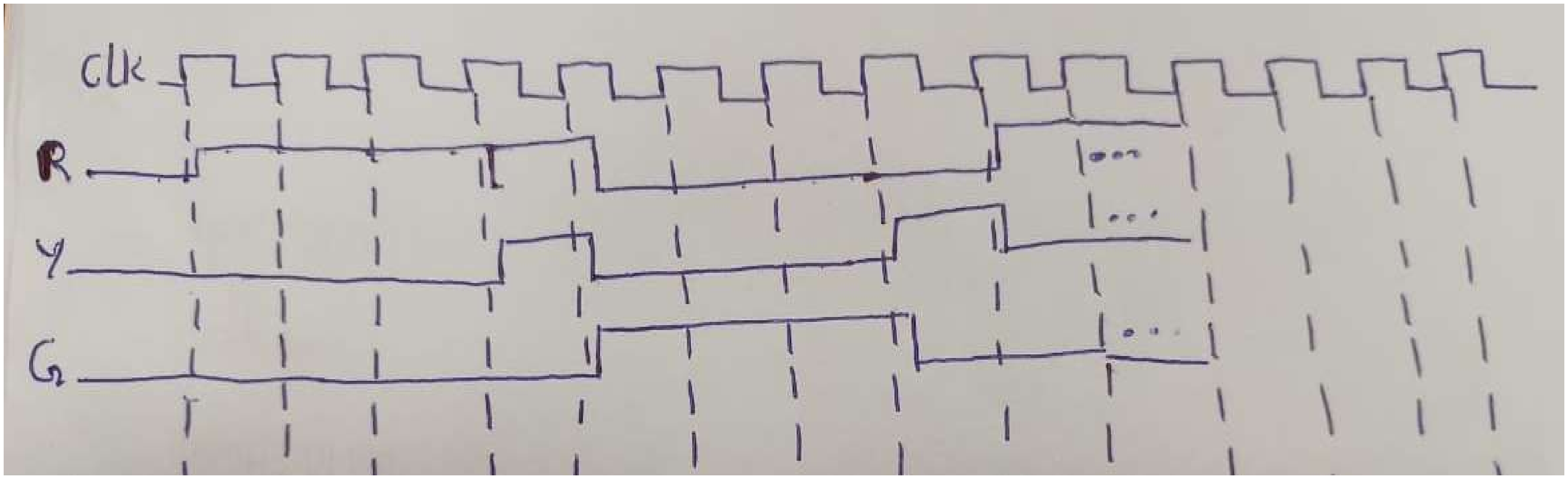
State Machines

Dr. Muhammad Fahim Ul Haque

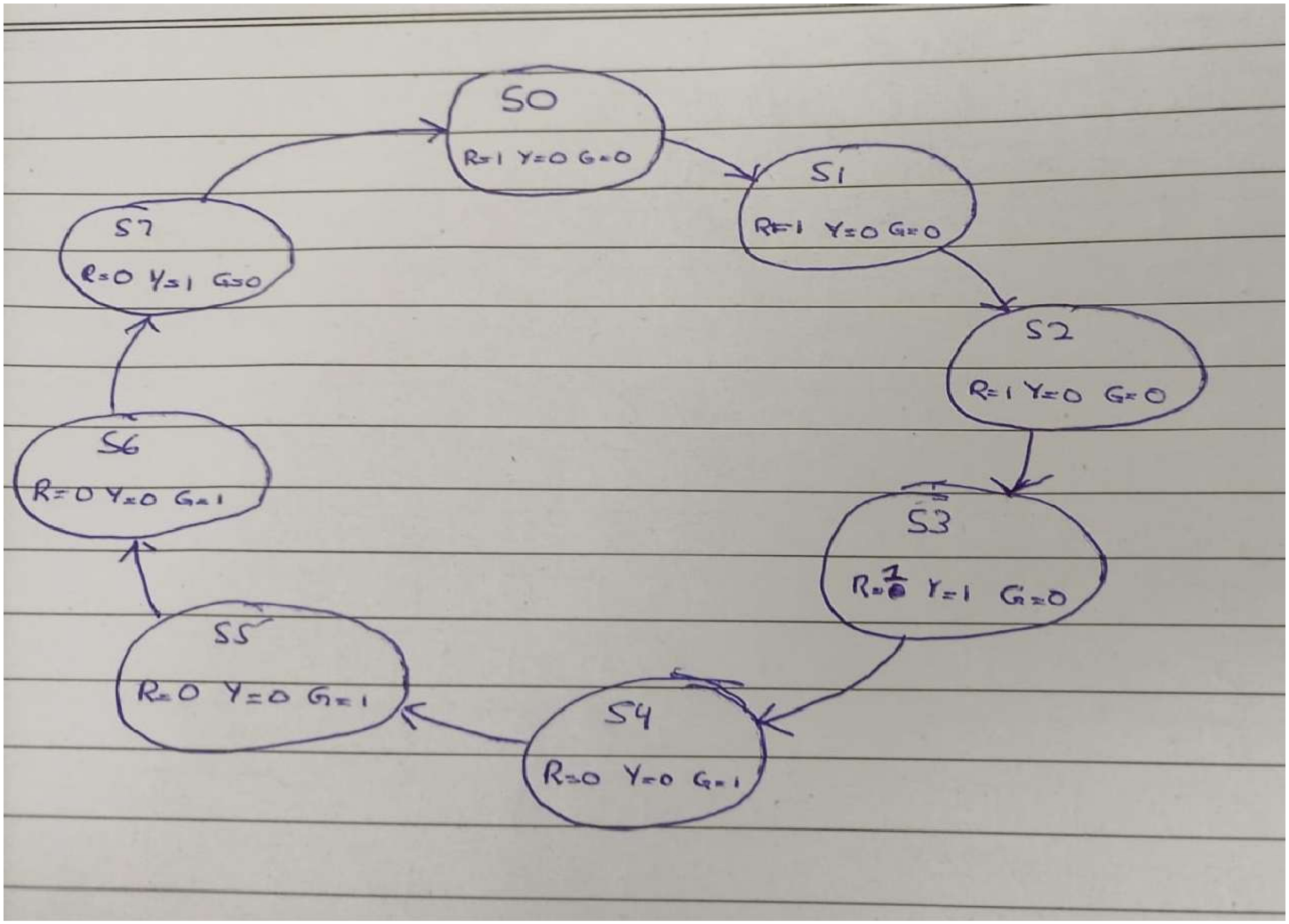
Moore and Mealy Machine



Single Way Traffic Signal Design



Single Way Traffic Signal Design



```
module trafficSignal (clk_50, r, y, g);
output reg r,y,g;
input clk_50;
wire clk;
reg [2:0] state, nextstate;
initial
begin
state = 3'b000; nextstate = 3'b000; {r,y,g} = 3'b000;
end
always @(state)
begin
case(state)
3'b000: nextstate = 3'b001;
3'b001: nextstate = 3'b010;
3'b010: nextstate = 3'b011;
3'b011: nextstate = 3'b100;
3'b100: nextstate = 3'b101;
3'b101: nextstate = 3'b110;
3'b110: nextstate = 3'b111;
3'b111: nextstate = 3'b000;
endcase
end
always @(state)
begin
case(state)
3'b000: {r,y,g} = 3'b100;
3'b001: {r,y,g} = 3'b100;
3'b010: {r,y,g} = 3'b100;
3'b011: {r,y,g} = 3'b110;
3'b100: {r,y,g} = 3'b001;
3'b101: {r,y,g} = 3'b001;
3'b110: {r,y,g} = 3'b001;
3'b111: {r,y,g} = 3'b010;
endcase
end
always @(posedge clk)
begin
state <= nextstate;
end
clk_divider divider1 (.clk(clk),.clk_50(clk_50));
endmodule
```

Traffic Signal State Verilog Code 1

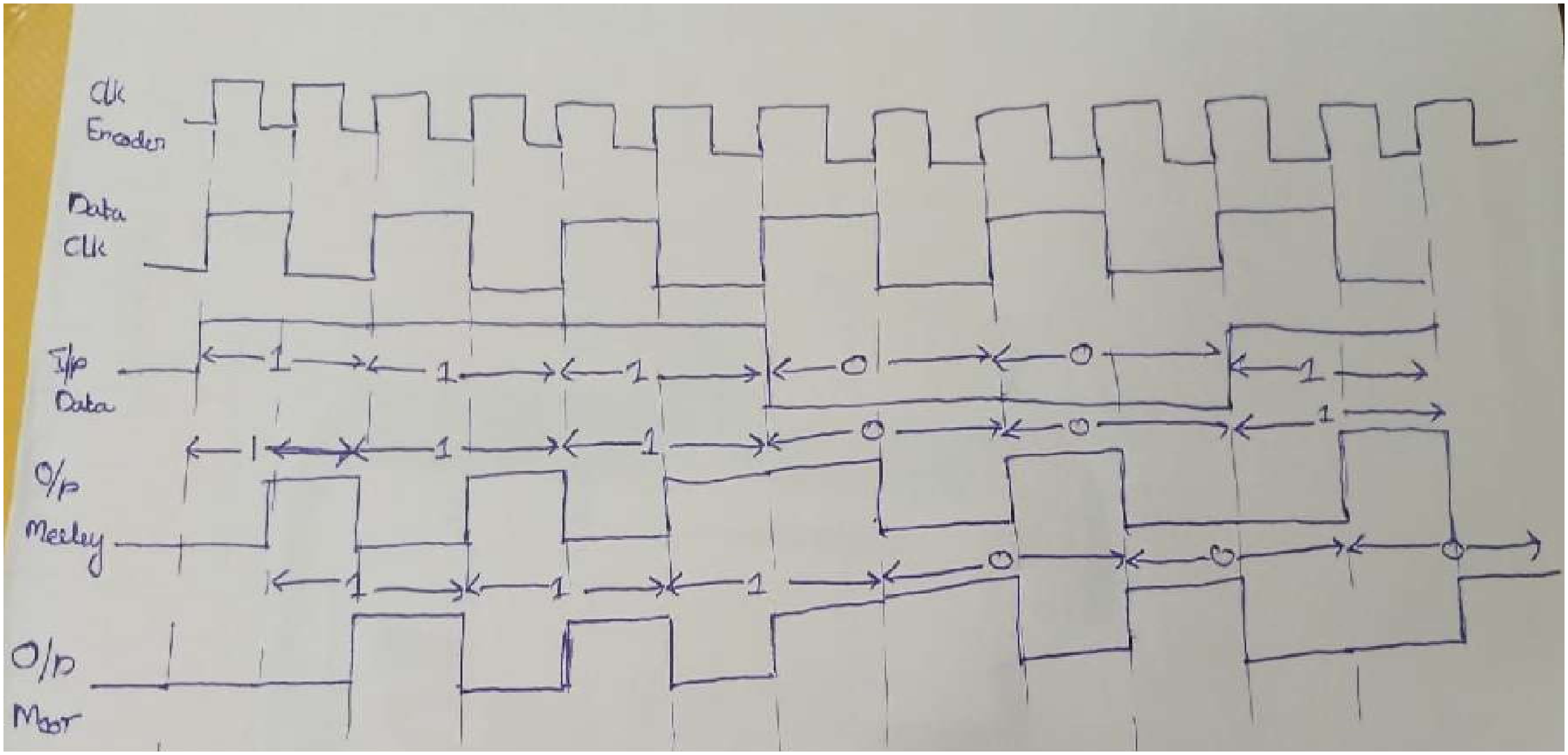
Clock Divider Verilog Code

```
module clk_divider(clk, clk_50);
input clk_50;
output reg clk;
reg [31:0] count;
initial
count = 32'h00000000;
//reg count;
always@(posedge (clk_50))
begin
if (count >= 32'd50000000)
count <= 32'h00000000;
else
begin
if (count<=32'd25000000)
clk <= 1;
else
clk <= 0;
count <= count + 1;
end
end
endmodule
```

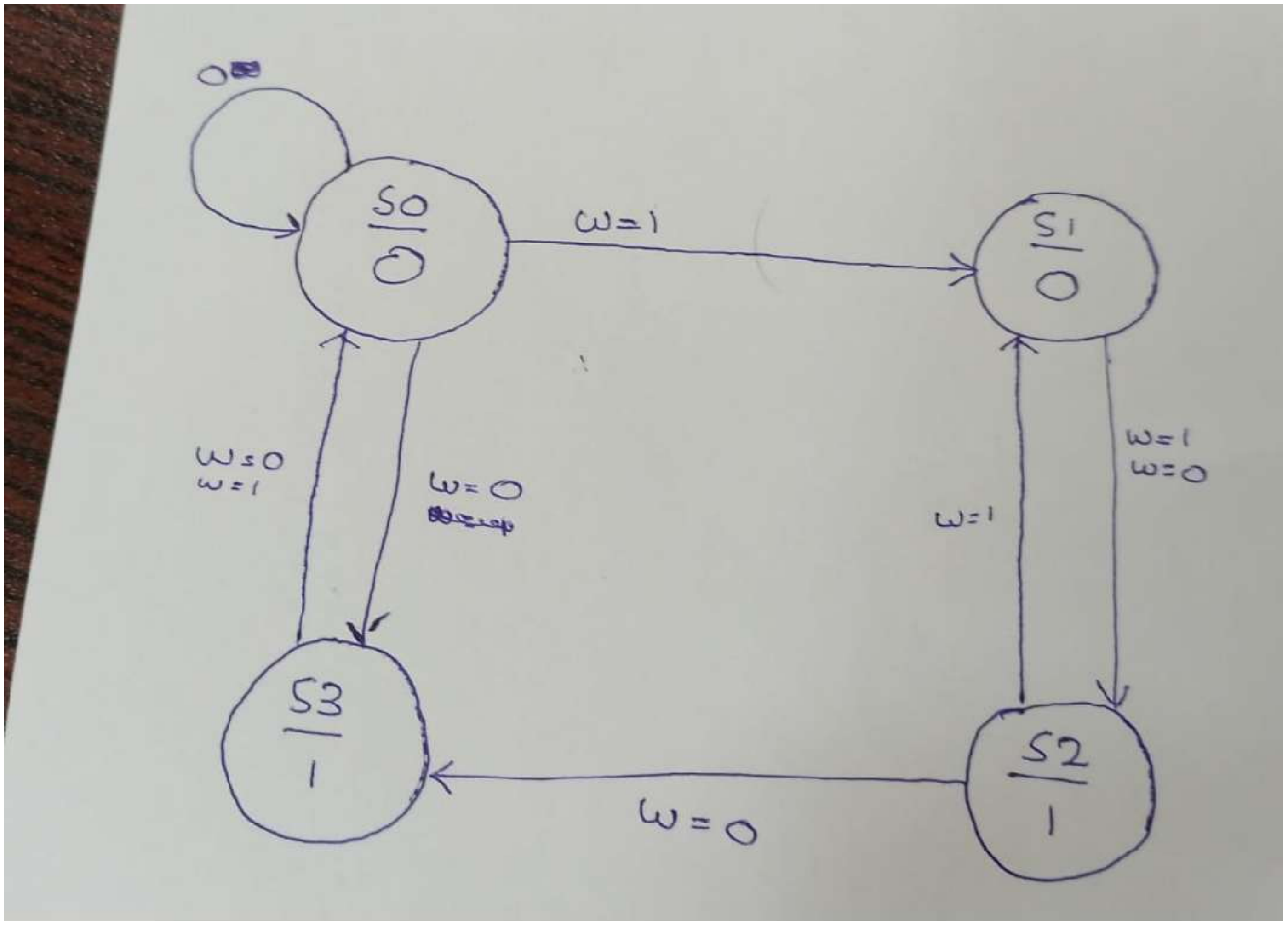
Traffic Signal State Verilog Code 2

```
module traffic_signal_struct(r,y,g,clk_50);
output logic r,y,g;
input clk_50;
logic clk;
logic [2:0] cs;
logic [2:0] ns;
clk_divider divider1 (.clk(clk),.clk_50(clk_50));
// Combinational circuit for output generation
assign r = (~cs[2]&~cs[1]&~cs[0]) | (~cs[2]&~cs[1]&cs[0]) | (~cs[2]&cs[1]&~cs[0]) | (~cs[2]&cs[1]&cs[0]);
assign y =(~cs[2]&cs[1]&cs[0]) | (cs[2]&cs[1]&cs[0]);
assign g =(cs[2]&~cs[1]&~cs[0]) | (cs[2]&~cs[1]&cs[0]) | (cs[2]&cs[1]&~cs[0]);
// Combinational circuit for next state
assign ns[2] = (~cs[2]&cs[1]&cs[0]) | (cs[2]&~cs[1]&~cs[0]) | (cs[2]&~cs[1]&cs[0]) | (cs[2]&cs[1]&~cs[0]);
assign ns[1] = (~cs[2]&~cs[1]&cs[0]) | (~cs[2]&cs[1]&~cs[0]) | (cs[2]&~cs[1]&cs[0]) | (cs[2]&cs[1]&~cs[0]);
assign ns[0] = (~cs[2]&~cs[1]&~cs[0]) | (~cs[2]&cs[1]&~cs[0]) | (cs[2]&~cs[1]&~cs[0]) |
       (cs[2]&cs[1]&~cs[0]);
// State Register
dflipflop dff0(cs[0],ns[0],clk);
dflipflop dff1(cs[1],ns[1],clk);
dflipflop dff2(cs[2],ns[2],clk);
endmodule
```

Manchester Encoder



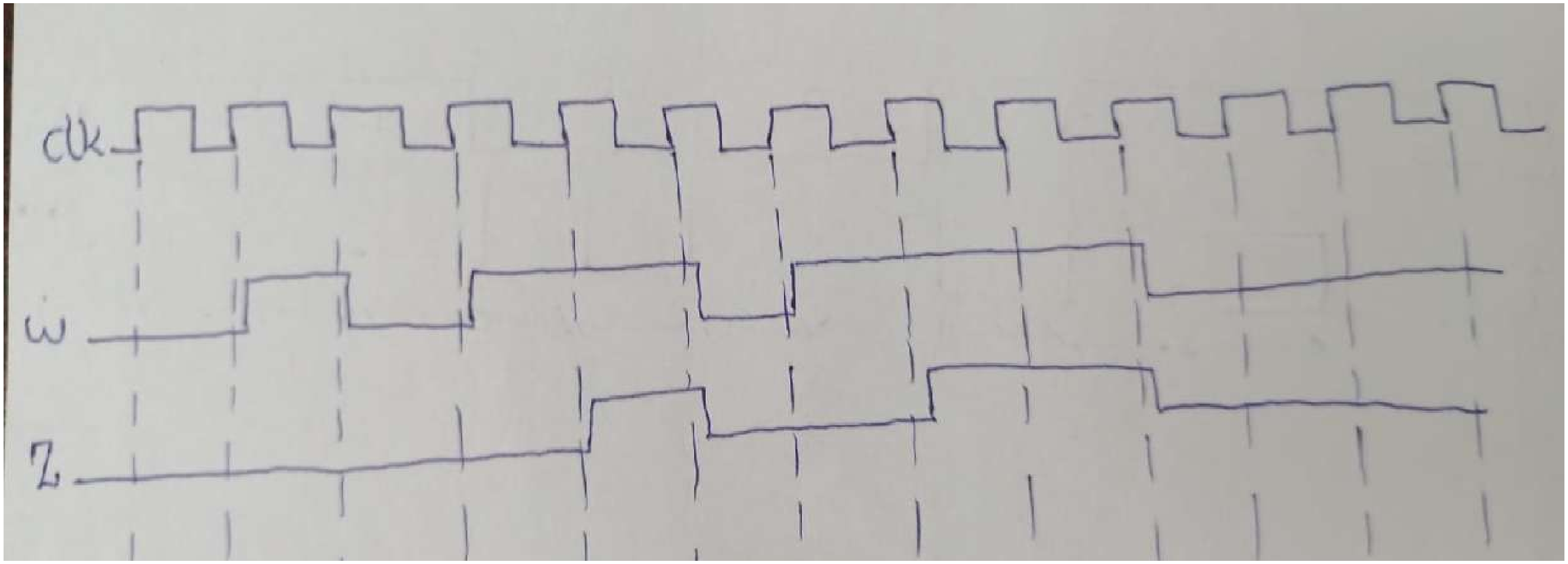
Manchester Encoder State Diagram



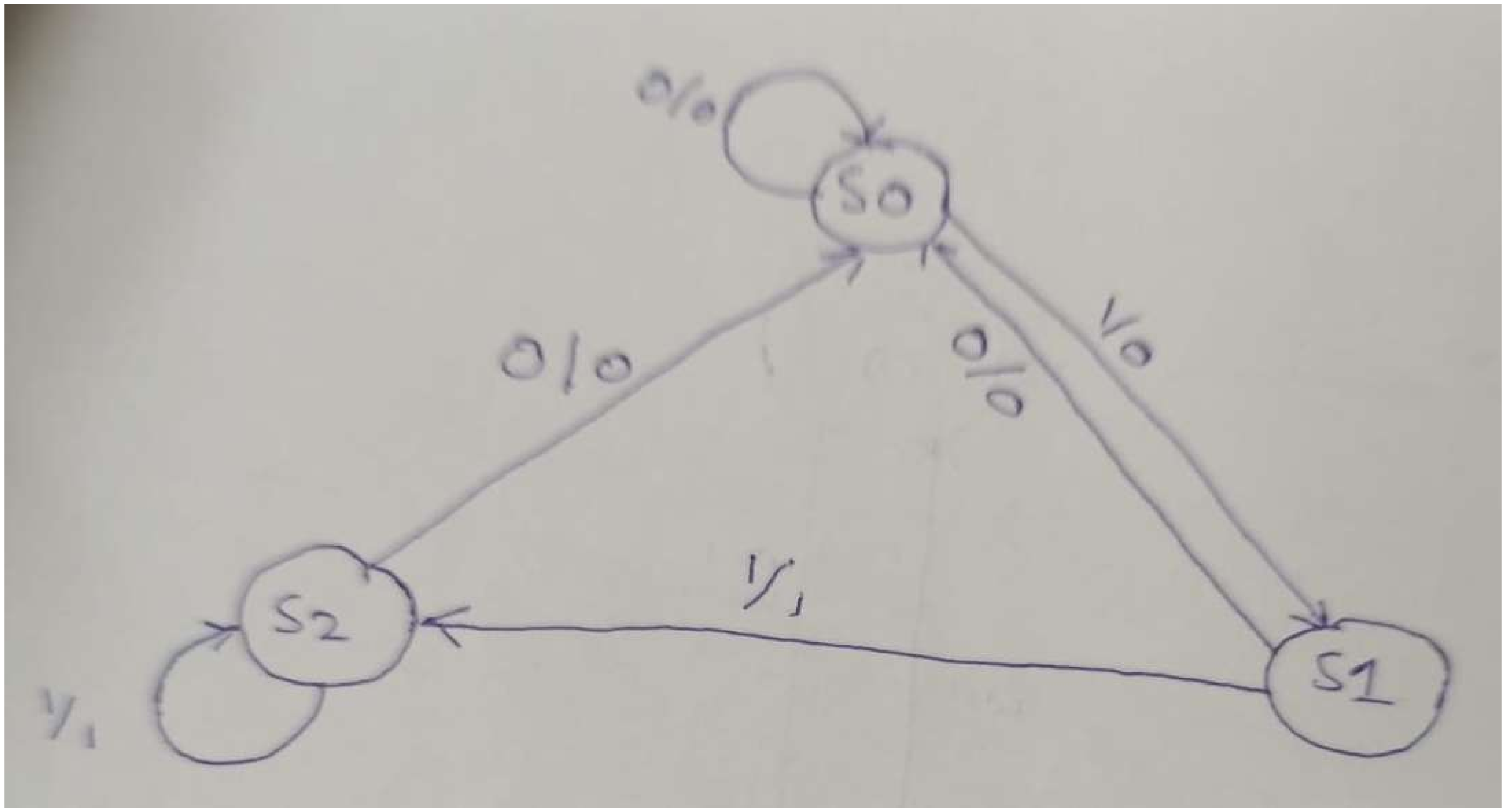
Manchester Encoder Verilog Code

```
module manchester_moore(z,w,clk);
output z;
input w, clk;
reg z;
reg [1:0] state, nextstate;
initial
begin
state = 2'b00; nextstate = 2'b00;
end
always @(state,w)
begin
case(state)
2'b00:begin
if (w===0)
nextstate = 2'b11;
else if(w === 1)
nextstate = 2'b01;
else
nextstate = 2'b00;
z = 0;
end
2'b01:begin
nextstate = 2'b10;
z = 0;
end
2'b10:begin
if (w===0)
nextstate = 2'b11;
else if (w === 1)
nextstate = 2'b01;
else if (w===0)
nextstate = 2’b11;
else
nextstate = 2’b00
z = 1;
end
2'b11:begin
nextstate = 2'b00;
z = 1;
end
default:
begin
state = 2'b00;
nextstate = 2'b00;
end
endcase
end
always @(posedge clk)
begin
state <= nextstate;
end
endmodule
```

Sequence Detector



Sequence Detector



Sequence Detector Verilog Code 1

```
module seq_detector(z,w,clk);
output reg z; input w,clk;
reg [1:0] state, nextstate;
initial
begin
state = 2'b00; nextstate = 2'b00;
end
always @(state,w)
begin
case(state)
//state 0
2'b00:begin
if(w === 1)
begin
nextstate = 2'b01; z = 0;
end
else
begin
nextstate = 2'b00;
z = 0;
end
end
//State 1
2'b01:begin
if(w === 1)
begin
nextstate = 2'b10;
z = 1;
end
else
begin
nextstate = 2'b00;
z = 0;
end
end
//state 2
2'b10:begin
if(w === 1)
begin
nextstate = 2'b10; z = 1;
end
else
begin
nextstate = 2'b00;
z = 0;
end
end
endcase
end
//state register
always@(posedge clk)
begin
state <= nextstate;
end

endmodule
```

Sequence Detector Verilog Code 2

```
module seq_det_struct(z,w,clk);
output z;
input w,clk;
reg [1:0] c;
reg [1:0] n;
assign z = (c[1]&~c[0]&w);
assign n[1] = (~c[1]&c[0]&w) | (c[1]&~c[0]&w);
assign n[0] = ~c[1]&~c[0]&w;
dflipflop d0(c[0],n[0],clk);
dflipflop d1(c[1],n[1],clk);
endmodule
```

```
module dflipflop(q,d,clk);
output reg q;
input d,clk;
initial
begin
q = 0;
end
always@(posedge clk)
begin
q <= d;
end
endmodule
```