**Learning Objectives:**
i) **Introduction to Sequential circuits**
ii) **Blocking and Non-Blocking Assignments**
iii) **Sequential and Parallel blocks**
iv) **Finite State Machine Implementation (Mealy Machine)**
v) **4-bit Shift Register Implementation**

Introduction:

In the first two labs, we have learned how to simulate digital circuits using different types of modeling (i.e. Gate level, Data flow, Behavioral) on combinational circuit simulation in VeriLog. In this lab, we will learn how to simulate digital circuits using different types of modeling on sequential circuit simulation in VeriLog.

A Sequential circuit is a circuit made up by combining logic gates such that the required logic at the output(s) depends not only on the current input logic conditions but also on the past inputs, outputs and sequences.
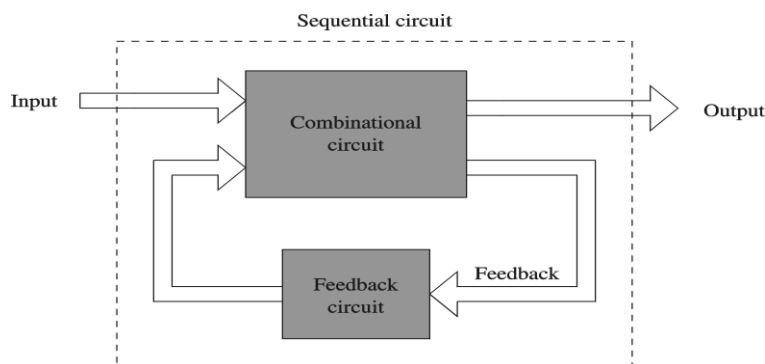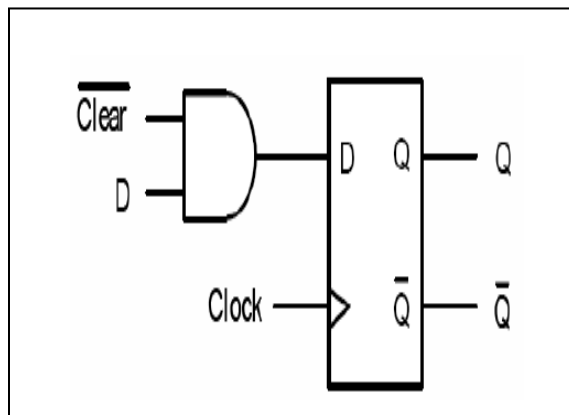

Sequential Circuits has a feedback of the output(s) from a stage to the input of either that stage or any previous stage.

Problem 1:

```
module dff_sync_clear(d, clearb,
clock, q);

input d, clearb, clock;
output q;
reg q;

always @ (posedge clock)
begin
if (!clearb) q <= 1'b0;
else q <= d;
end
endmodule
```
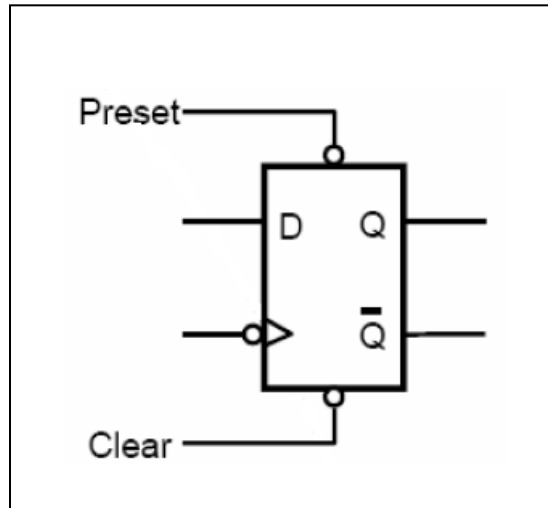
```verilog
module dff_async_clear(d
, clearb, clock, q);

input d, clearb, clock;
output q;
reg q;

always @ (negedge cle
arb or posedge clock)
begin
if (!clearb) q <= 1'b0;
else q <= d;
end
endmodule
```


// Test Bench

```verilog
module Testing;

reg d, clk, rst;
wire q;

dff_sync_clear dff (d, clk, rst, q); // Or dff_async_clear dff (d, clk,
rst, q);

//Always at rising edge of clock display the signals
always @(posedge clk)begin
$display("d=%b, clk=%b, rst=%b, q=%b\n", d, clk, rst, q);
end

//Module to generate clock with period 10 time units
initial begin
forever begin
clk=0;
#5
clk=1;
#5
clk=0;
end
end
initial begin
d=0; rst=1;
#4
d=1; rst=0;
#50
d=1; rst=1;
#20
d=0; rst=0;
end
endmodule
```

Blocking and Non-Blocking Assignments:

In Blocking assignment, evaluation and assignment are immediate.

```
always @ (a or b or c)
begin
x = a | b;                 1.Evaluate a|b, assign result to x.
y = a ^ b ^ c;             2.Evaluate a^b^c, assign result to y.
z = b & ~c;                3.Evaluate b& ~c, assign result to z.
end
```

In Nonblocking assignment, all assignments deferred until all right-hand sides have been evaluated (end of simulation timestep).

```
always @ (a or b or c)
begin
x <= a | b;        1.Evaluate a|b, but differ assignment of x.
y <= a ^ b ^ c;    2.Evaluate a^b^c, but differ assignment of y.
z <= b & ~c;       3.Evaluate b& ~c, but differ assignment of z.
end                4.Assign x, y, and z with their new values
```
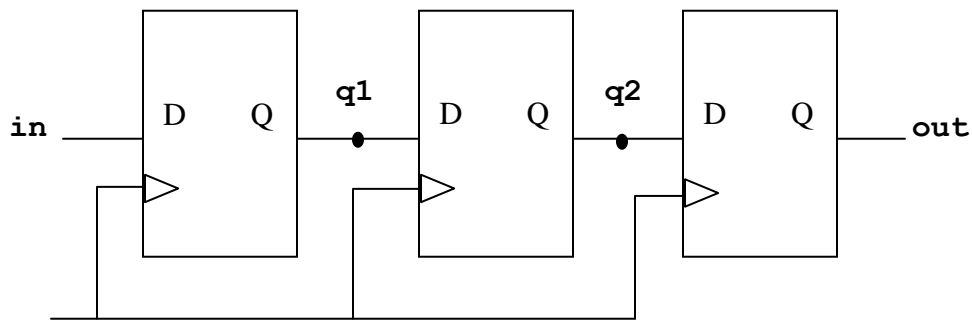


```
module nonblocking(in, clk,
out);

input in, clk;
output out;
reg q1, q2, out;

always @ (posedge clk)
begin
q1 <= in;
q2 <= q1;
out <= q2;
end
endmodule
```

```
module blocking(in, clk,
out);

input in, clk;
output out;
reg q1, q2, out;

always @ (posedge clk)
begin
q1 = in;
q2 = q1;
out = q2;
end
endmodule
```

In verilog, we have two types of block – sequential blocks and parallel blocks.

Sequential Block

In the sequential blocks, **begin** and **end** keywords are used to group the statements, All the statement in this group executes sequentially. (this rule is not applicable for nonblocking assignments). If the statements are given with some timing/delays then the given delays get added into. It would be clearer with following examples.

<table>
<tr>
<td>

Example - 1:

```
reg a,b,c;
initial
begin
     a = 1'b1;
     b = 1'b0;
     c = 1'b1;
end
```

</td>
<td>

Example - 2:

```
reg a,b,c;
initial
begin
     #5 a = 1'b1;
     #10 b = 1'b0;
     #15 c = 1'b1;
end
```

</td>
</tr>
</table>

The **Example - 1** is showing the sequential block without delays, All the statements written inside the begin-end will execute sequentially and after the execution of initial block, final values are a=1, b=0 and c=1

The **Example - 2** is showing the sequential block with delays, In this case, the same statements are given with some delays, Since All the statements execute sequentially, the a will get value 1 after 5 time unit, b gets value after 15 time unit and c will take value 1 after 30 time unit

Parallel Block:

The statements are written inside the parallel block, execute parallel, If the sequencing is required then it can be given by providing some delays before the statements. In parallel blocks, all the statements occur within **fork and join**

<table>
<tr>
<td>

Example - 1:

```
reg a,b,c;
initial
fork
   #5 a = 1'b1;
   #10 b = 1'b0;
   #15 c = 1'b1;
join
```

</td>
<td>

Example - 2:

```
reg a,b,c,d;
initial
begin
fork
   #5 a = 1'b1;
   #10 b = 1'b0;
   #15 c = 1'b1;
join
#30 d = 1'b0;
end
```

</td>
</tr>
</table>

In **Example-1**, all the statements written inside the fork and join, executes parallel, it means the c with have value '1' after 15 time unit.

In **Example-2**, the initial block contains begin-end and fork-join both. In this case c takes value after 15 time unit, and d takes the value after 30 time unit.

Mealy Machine:

A Mealy machine is a finite state transducer that generates an output based on its current state and input. This means that the state diagram will include both an input and output signal for each transition edge.
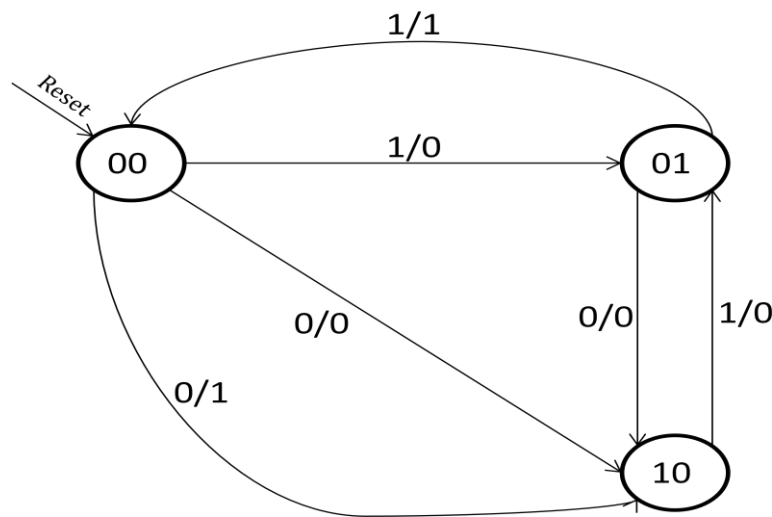
Solution:

```
module mealy( clk, rst, inp, outp);

input clk, rst, inp;
output outp;
reg [1:0] state;
reg outp;

always @( posedge clk, posedge rst ) begin
if( rst ) begin
state <= 2'b00;
outp <= 0;
end

else  begin
case( state )
2'b00: begin
if( inp ) begin
state <= 2'b01;
outp  <= 0;
end
```

```verilog
                else begin
                state <= 2'b10;
                outp <= 0;
                end
                end

        2'b01: begin
        if( inp ) begin
        state <= 2'b00;
        outp  <= 1;
        end
        else  begin
        state <= 2'b10;
        outp <= 0;
        end
        end

        2'b10: begin
        if( inp ) begin
        state <= 2'b01;
        outp  <= 0;
        end
        else begin
        state <= 2'b00;
        outp <= 1;
        end
        end

        default: begin
        state <= 2'b00;
        outp <= 0;
        end
        endcase

        end
        end
        endmodule

// Test Bench

module mealy_test;

reg  clk, rst, inp;
wire outp;
reg[15:0] sequence;
integer i;

mealy duty( clk, rst, inp, outp);

initial
begin
clk = 0;
```

```
rst = 1;
sequence = 16'b0101_0111_0111_0010;
#5 rst = 0;

for( i = 0; i <= 15; i = i + 1)
begin
inp = sequence[i];
#2 clk = 1;
#2 clk = 0;
$display("State = ", duty.state, " Input = ", inp, ", Output = ",
outp);
end
testing;
end

task testing;
for( i = 0; i <= 15; i = i + 1)
begin
inp = $random % 2;
 #2 clk = 1;
 #2 clk = 0;
 $display("State = ", duty.state, " Input = ", inp, ", Output =
", outp);
end
endtask
endmodule
```

Problem 2: Implementing a 4-bit Shift Register

An n-bit shift register is generally comprised of a set of n flip-flops which provide n bits of storage. The flip-flops are connected in such a way as to produce a shifting action of the bits stored in the individual flip-flops. The bits shift at the active portion of the system clock which is usually a clock edge.

Following figure shows a 4-bit shift register that uses D flip-flops for the individual storage elements.
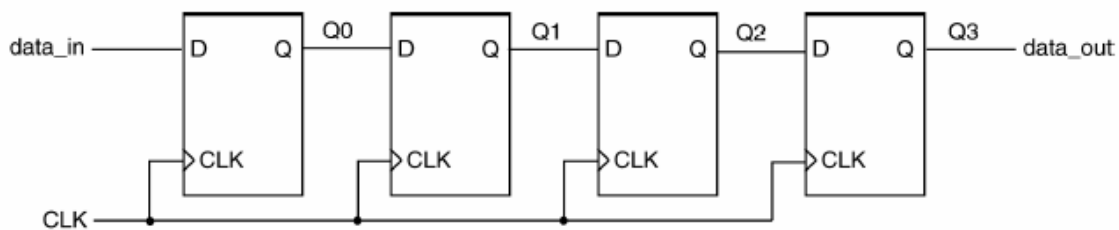
Figure 1: Diagram for a 4-bit shift register.

Solution:

```
//VeriLog Code for 4 bit Shift Register.

module shiftreg(EN, in, CLK, Q);
```

```verilog
parameter n = 4;
input EN;
input in;
input CLK;
output [n-1:0] Q;
reg [n-1:0] Q;

initial
Q=4'd10;
always @(posedge CLK)
begin
if (EN)
Q={in,Q[n-1:1]};
end
endmodule

// Test Bench of 4 bit shift register

module shiftregtest;

parameter n= 4;
reg EN,in , CLK;
wire [n-1:0] Q;
//reg [n-1:0] Q;

shiftreg shreg(EN,in,CLK,Q);

initial
begin
CLK=0;
end

always
#2 CLK=~CLK;
initial
$monitor($time,"EN=%b in= %b Q=%b\n",EN,in,Q);
initial
begin
in=0;EN=0;
#4 in=1;EN=1;
#4 in=1;EN=0;
#4 in=0;EN=1;
#5 $finish;
end
endmodule
```
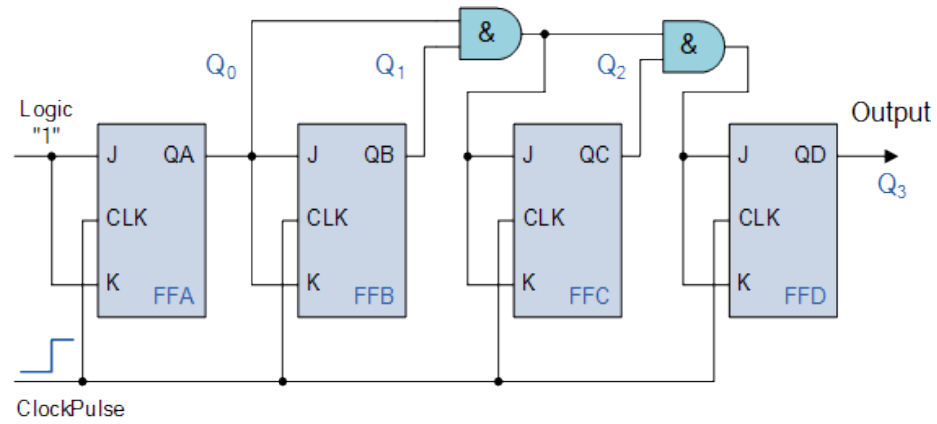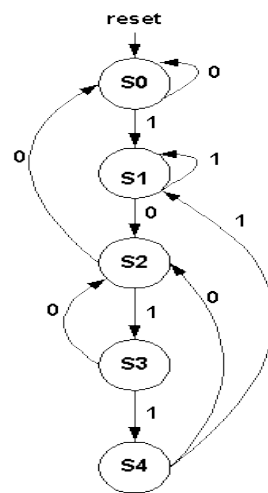
Exercises:

(Q.1) Implement Binary 4-bit Synchronus counter using J-K Flip Flops. Verify the design by writing a testbench module.

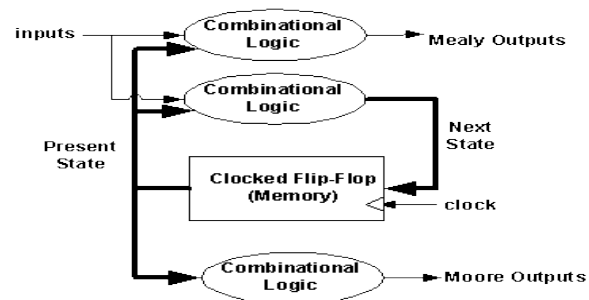**(Q.2.)** Design a FSM (Finite State Machine) to detect a sequence 10110.

State Transition Diagram

FSM Block Diagram

(Q.3.) Design and implement a four-bit serial adder. Make use of four-bit shift register constructed in **Problem 2**. Verify the design by writing a testbench module.