EEE 248/ CNG 232
Logic Design

Project 3

Yıldız Alara Köymen

2453389

TABLE OF CONTENTS

3.1: Introduction

. Page 2

3.2: Preliminary Work

3.2.2: D Flip-Flop with Asynchronous Reset and Synchronous Load

1. Page 3
2. Page 6
3. Page 8

3.2.3: 4-bit Shift Register with Asynchronous Reset and Synchronous Load

1. Page 10
2. Page 12
3. Page 13

3.2.4: 4-bit Synchronous Wrap-Around Up/Down Counter with Synchronous Up/Down Select, Count-Enable and Asynchronous Reset

1. Page 16
2. Page 19
3. Page 19

3.2.5: 3-bit Down Counter Using D Flip-Flop

. Page 21

3.2.6: 3-bit Down Counter using JK Flip-Flop

. Page 23

3.2.7: 4-bit Synchronous Parallel Load Shift Register with Counter and Asynchronous Reset

1. Page 28
2. Page 32
3. Page 33

3.3: Conclusion:

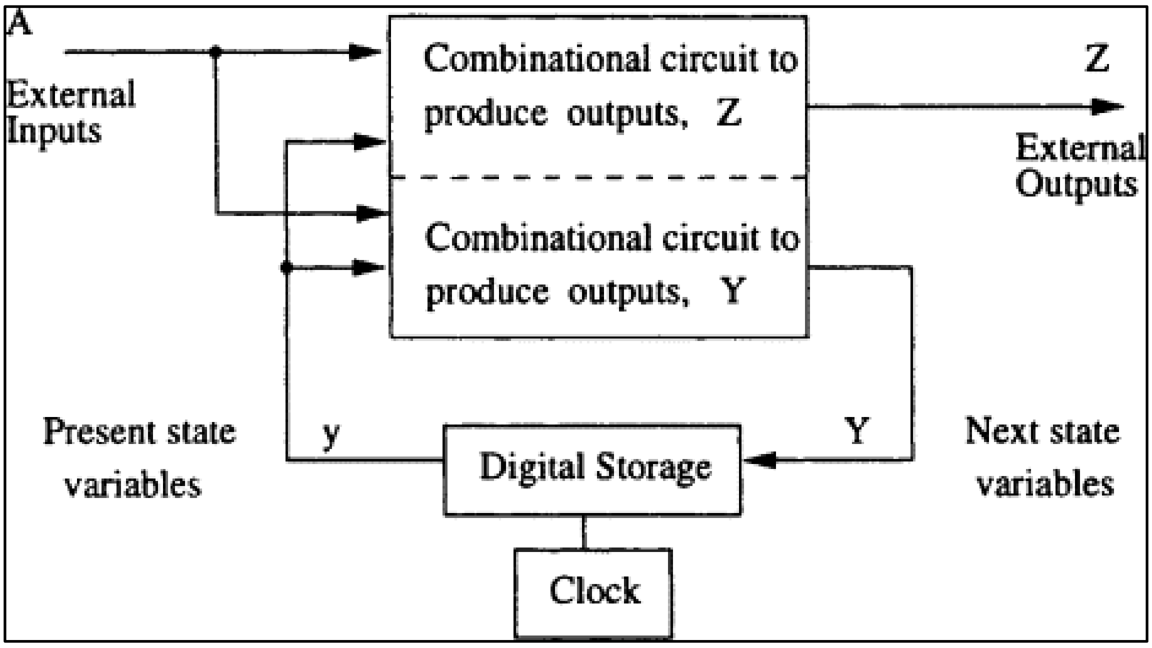
. Page 35

3.1: Introduction:

The objective of this lab is to study basic synchronous sequential circuits, also known as storage elements.

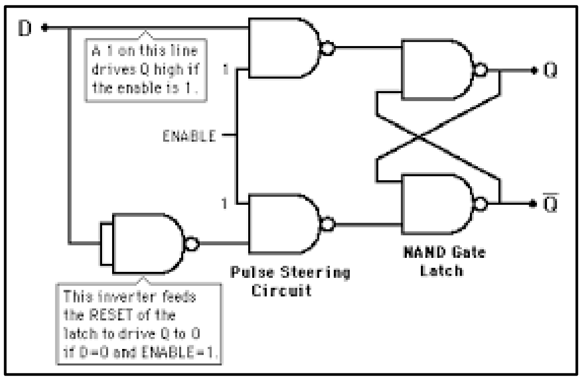
What is a sequential circuit?

In digital electronics, a **synchronous circuit** is known as a digital circuit in which the changes in the state of memory elements are synchronized by a clock signal.

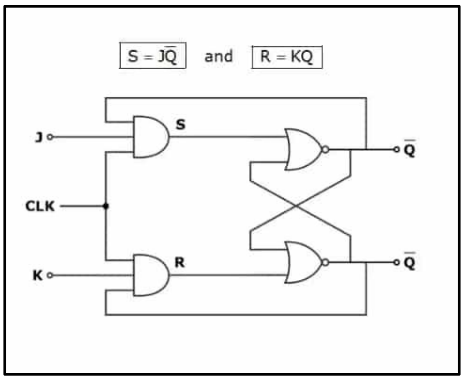


Data is stored in devices known as Flip Flops or latches. We will use D flip flops and JK flip flops for this lab.

D Flip Flop:


D
A 1 on this line drives Q high if the enable is 1.
1
ENABLE
1
Q
$\overline{Q}$
NAND Gate Latch
Pulse Steering Circuit
This inverter feeds the RESET of the latch to drive Q to 0 if D=0 and ENABLE=1.

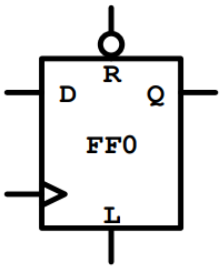
JK Flip Flop:



3.2 Preliminary Work:

3.2.2: D Flip-Flop with Asynchronous Reset and Synchronous Load

1. Draw a schematic to show how you would add combinational logic along with two new inputs (R and L) to a conventional D Flip-Flop to have the Reset and Load functions as shown in Figure 1. Note Load input take effect synchronously on the rising edge of the clock and Reset input is an active low input that takes effect asynchronously.



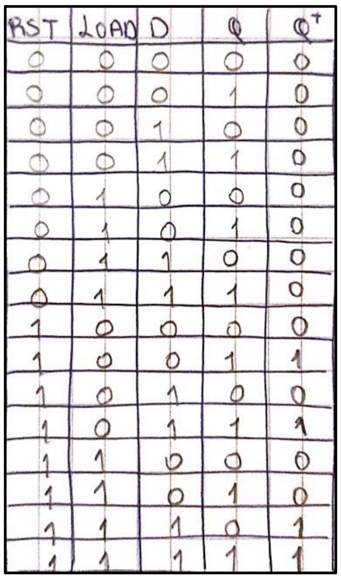
(a)

RST	LOAD	D	Q	CLK	Q^+	Operation
0	X	X	X	X	0	Asynchronous reset
1	1	X	X	↑	D	Load Data
1	0	X	X	↑	Q	Retain Value
1	X	X	X	0	Q	Retain Value
1	X	X	X	1	Q	Retain Value
1	X	X	X	↓	Q	Retain Value

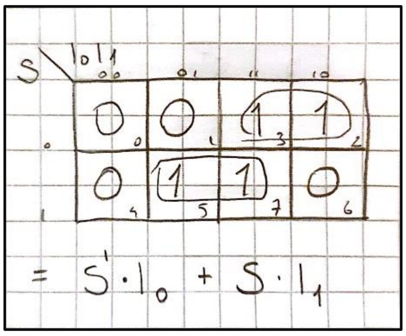
(b)

Figure 1: (a) New D-Flip-Flop Symbol, (b) Condensed Truth-Table for the New Flip-Flop.

We created a truth table based on figure 1. Afterwards we can create a Karnaugh map.



RST	LOAD	D	Q	Q^+
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1



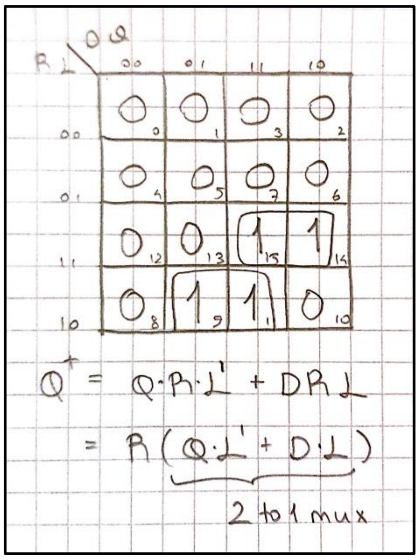
From this we find the formula of Q+, which is:

Q+ = Q*R*L' + DRL

Then we can have:

Q+ = R*(Q*L' + DL)

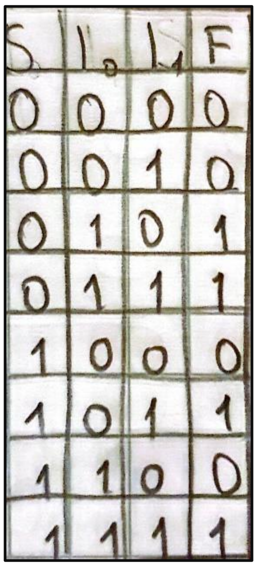
We can see that (Q*L’ + DL) looks like the 2 to 1 mux formula.



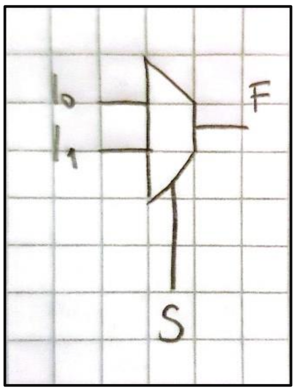
RL \ DQ	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	1	1
10	0	1	1	0

$$Q^+ = Q \cdot R \cdot L' + DRL$$

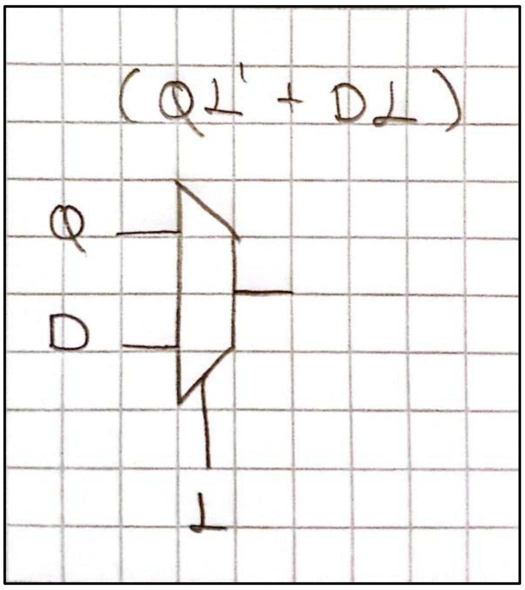
$$= R(\underbrace{Q \cdot L' + D \cdot L}_{\text{2 to 1 mux}})$$



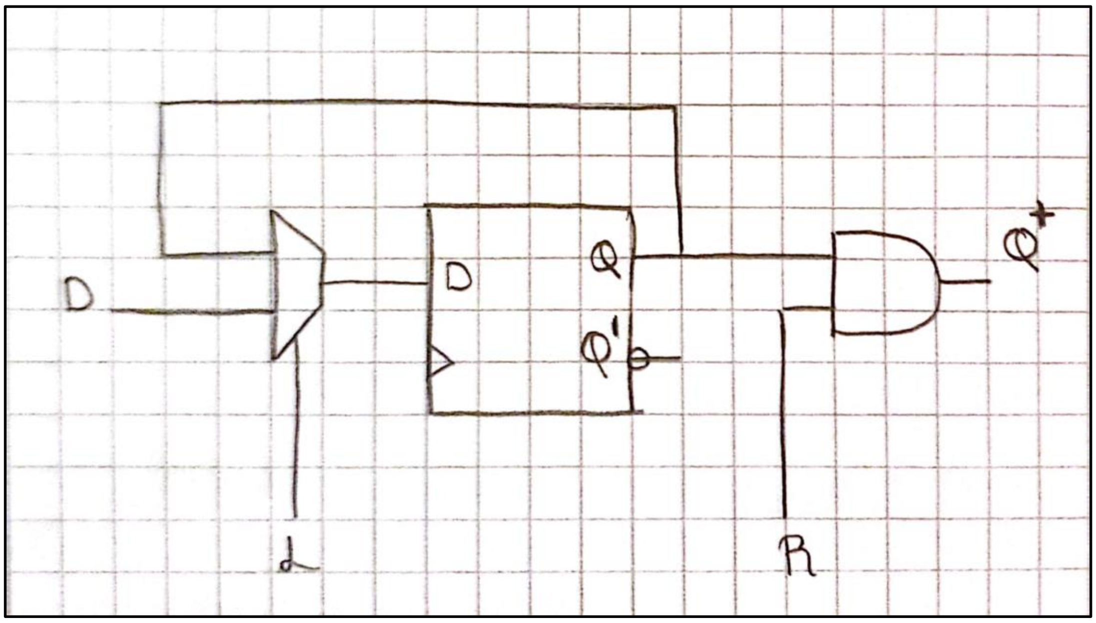
S	I_0	I_1	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



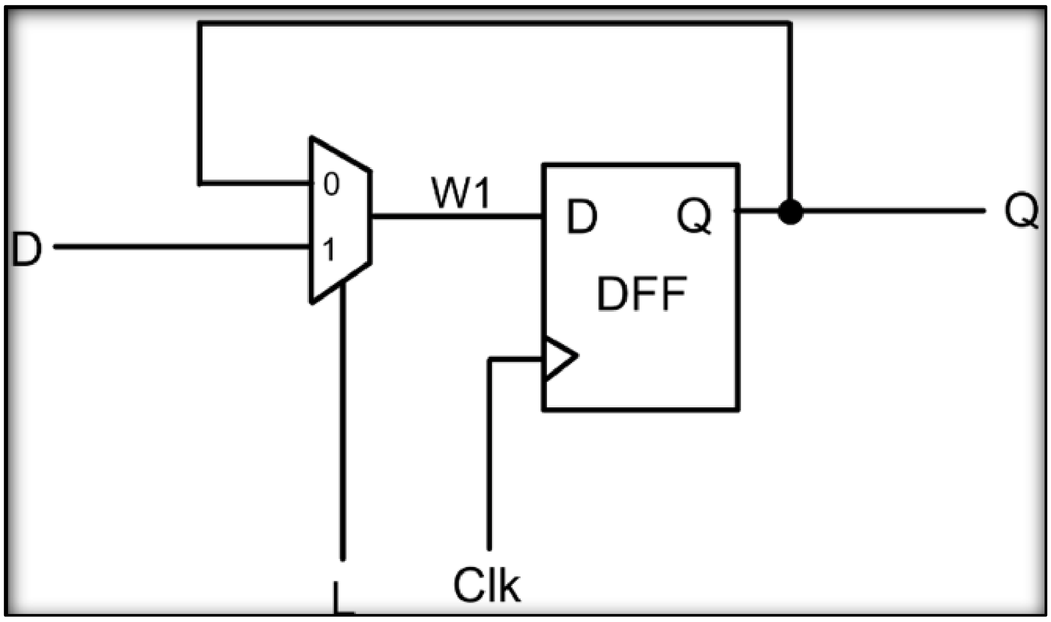
That means we can write:



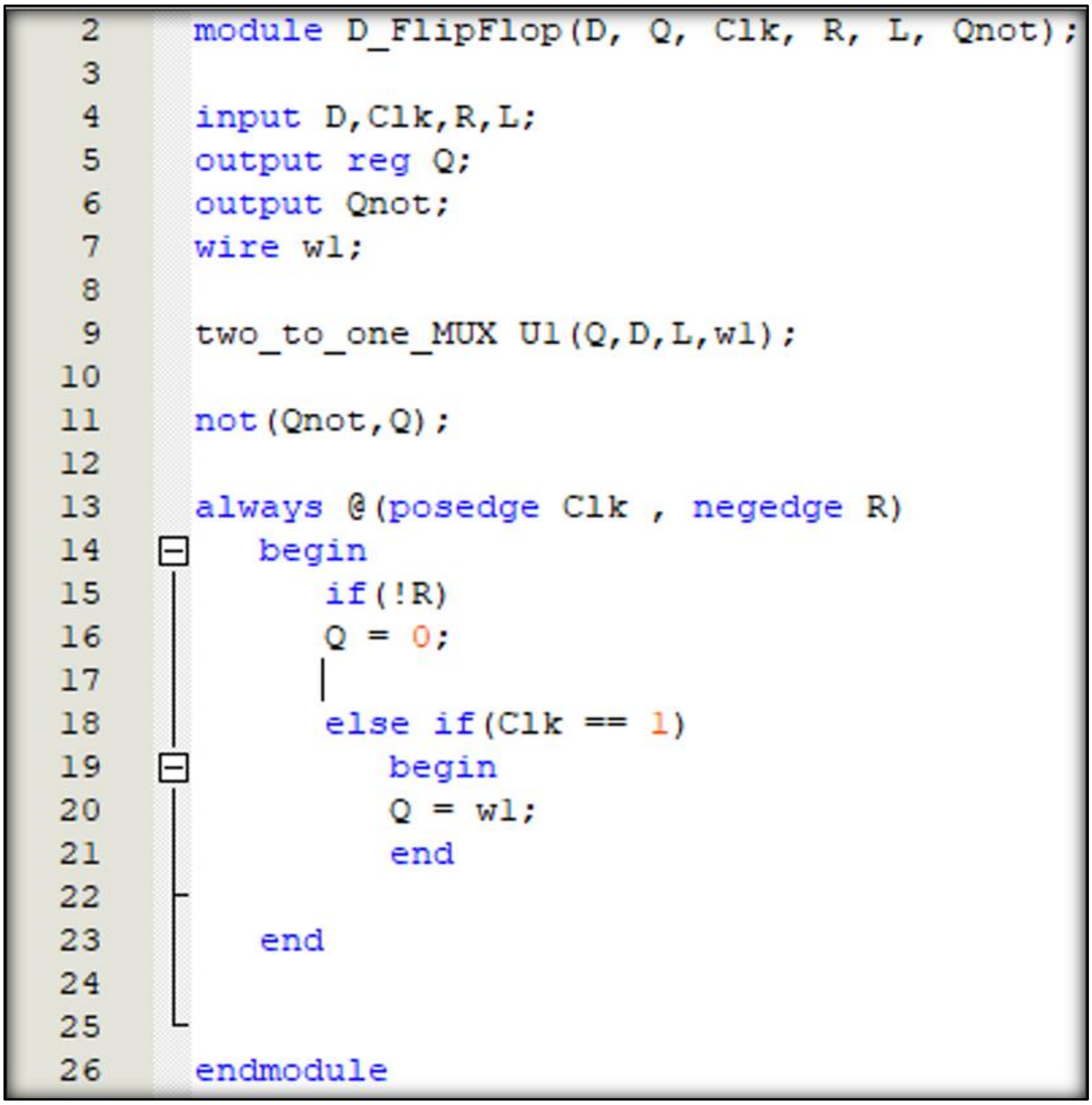
We can connect the mux to the D flip-flop. R is going to be asynchronous so its connected at the output with and and gate. If R is 0 the output will be 0.



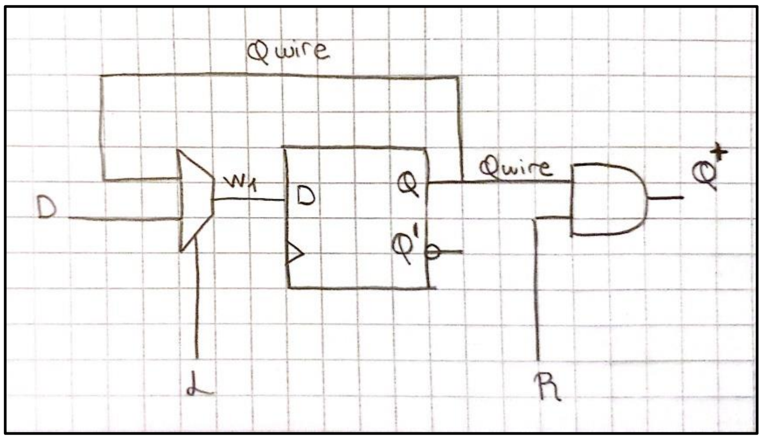
Later we changed the reset operation by using "negedge R" inside of the always block.



2. Use your answer from 3.2.2.1 to modify the Verilog code provided for the D Flip-Flop in section 3.3 to implement the new asynchronous reset and synchronous load functions.



```
 2   module D_FlipFlop(D, Q, Clk, R, L, Qnot);
 3
 4   input D,Clk,R,L;
 5   output reg Q;
 6   output Qnot;
 7   wire w1;
 8
 9   two_to_one_MUX U1(Q,D,L,w1);
10
11   not(Qnot,Q);
12
13   always @(posedge Clk , negedge R)
14      begin
15          if(!R)
16          Q = 0;
17
18          else if(Clk == 1)
19             begin
20             Q = w1;
21             end
22
23      end
24
25
26   endmodule
```



Line 10: Qwire, D and L enter the 2to1mux and output w1.

Line 11: Qwire and R enter and output Q.

Line 15 - Line 20: When Clk is on posedge, Qwire wll get the value of w1.

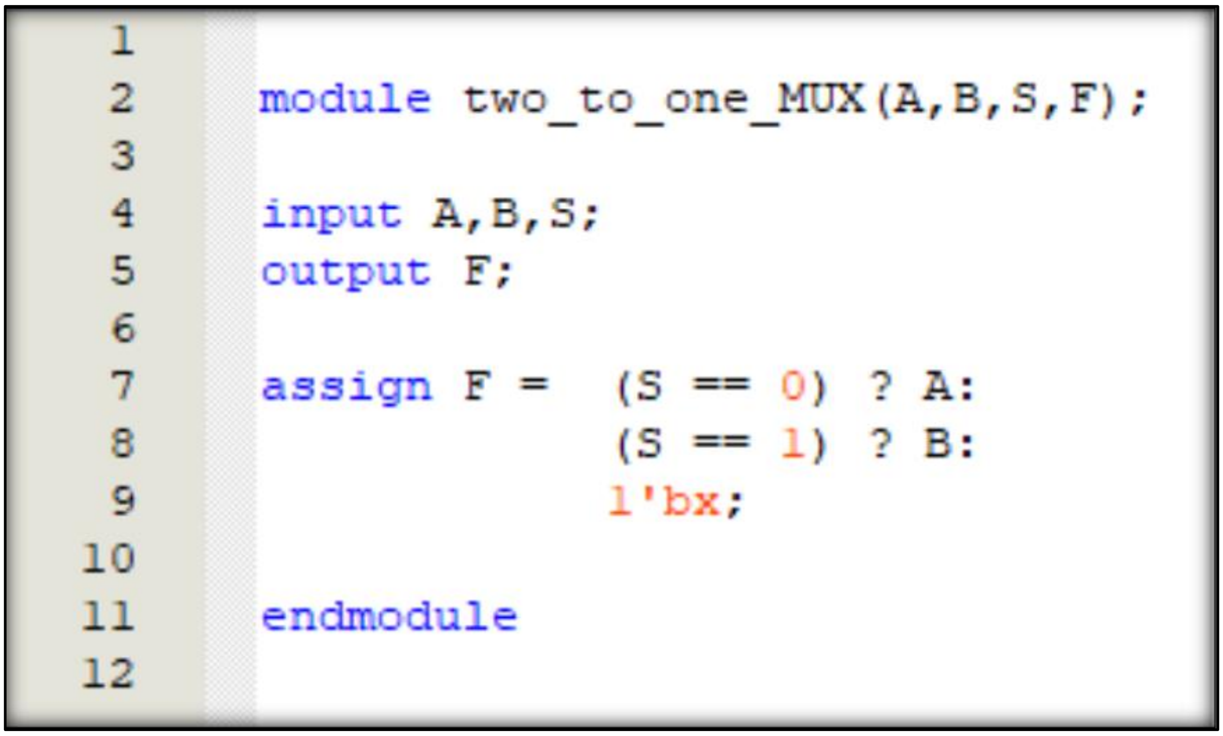
2to1 mux used for the D flip flop.

A would be Qwire,

B would be D,

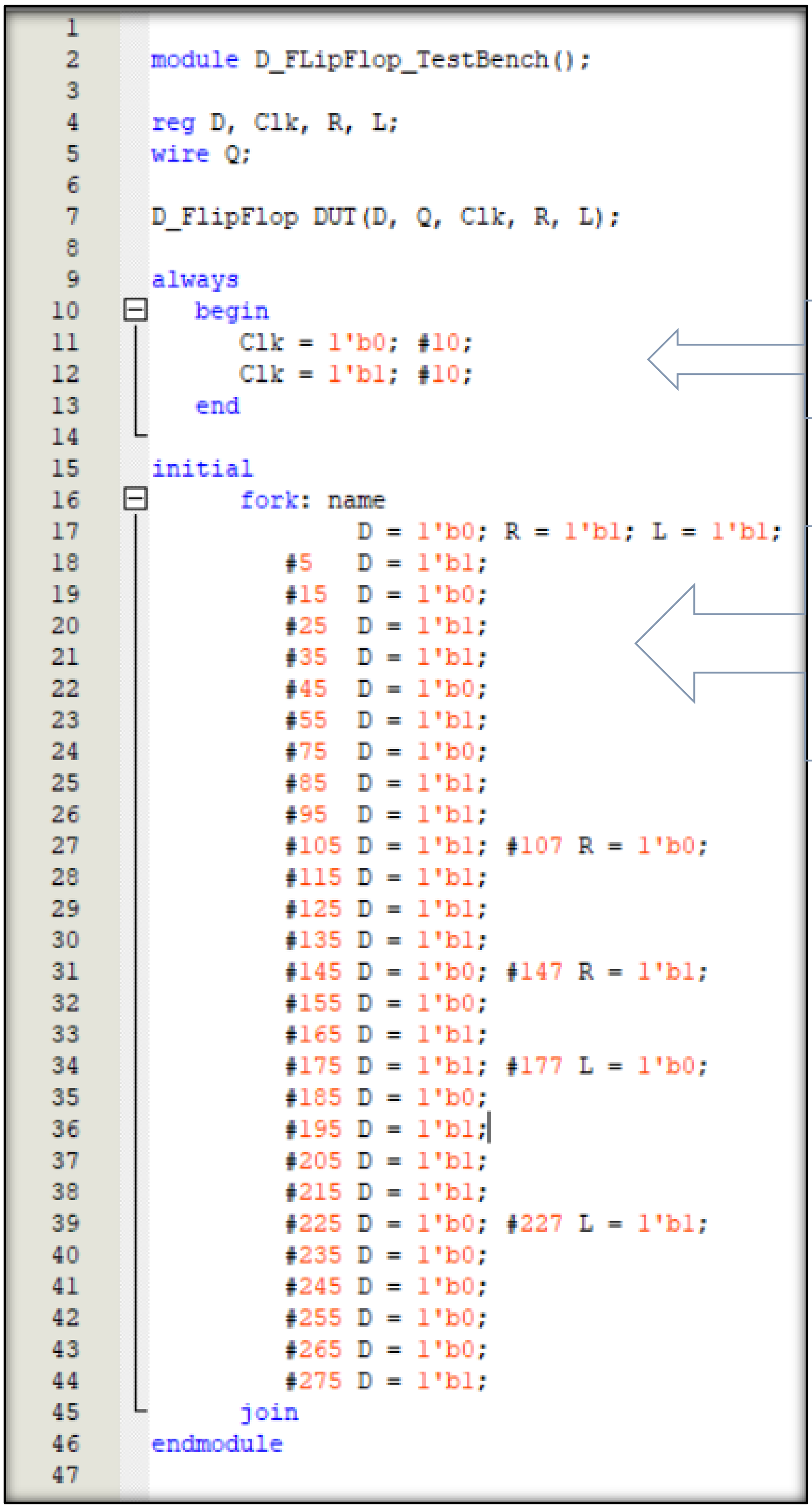
S would be L,

F would be w1.



```

module two_to_one_MUX(A,B,S,F);

input A,B,S;
output F;

assign F =   (S == 0) ? A:
             (S == 1) ? B:
             1'bx;


endmodule

```

3. Write a Verilog Testbench code to simulate and verify the functionality in ModelSim®.

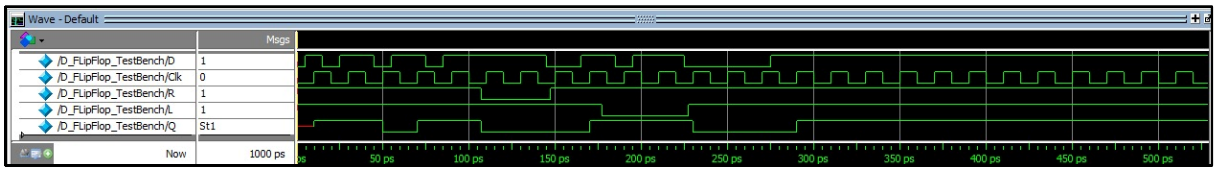


```verilog

module D_FLipFlop_TestBench();

reg D, Clk, R, L;
wire Q;

D_FlipFlop DUT(D, Q, Clk, R, L);

always
    begin
        Clk = 1'b0; #10;
        Clk = 1'b1; #10;
    end

initial
        fork: name
                D = 1'b0; R = 1'b1; L = 1'b1;
            #5  D = 1'b1;
            #15 D = 1'b0;
            #25 D = 1'b1;
            #35 D = 1'b1;
            #45 D = 1'b0;
            #55 D = 1'b1;
            #75 D = 1'b0;
            #85 D = 1'b1;
            #95 D = 1'b1;
            #105 D = 1'b1; #107 R = 1'b0;
            #115 D = 1'b1;
            #125 D = 1'b1;
            #135 D = 1'b1;
            #145 D = 1'b0; #147 R = 1'b1;
            #155 D = 1'b0;
            #165 D = 1'b1;
            #175 D = 1'b1; #177 L = 1'b0;
            #185 D = 1'b0;
            #195 D = 1'b1;
            #205 D = 1'b1;
            #215 D = 1'b1;
            #225 D = 1'b0; #227 L = 1'b1;
            #235 D = 1'b0;
            #245 D = 1'b0;
            #255 D = 1'b0;
            #265 D = 1'b0;
            #275 D = 1'b1;
        join
endmodule

```

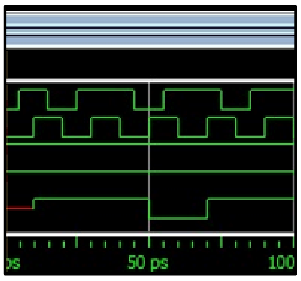
Clock would change every 10 pico seconds.

D would change every 5 pico seconds.

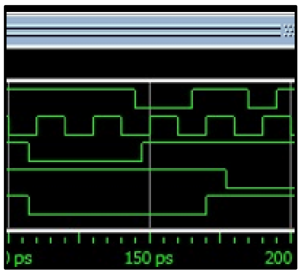
R and L are changed at given random values


Wave - Default

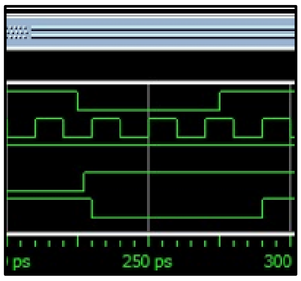
	Msgs
/D_FLipFlop_TestBench/D	1
/D_FLipFlop_TestBench/Clk	0
/D_FLipFlop_TestBench/R	1
/D_FLipFlop_TestBench/L	1
/D_FLipFlop_TestBench/Q	St1
Now	1000 ps



D flip flop won’t have a value till 10 ps. It will load the frist value when it reaches the first posedge.



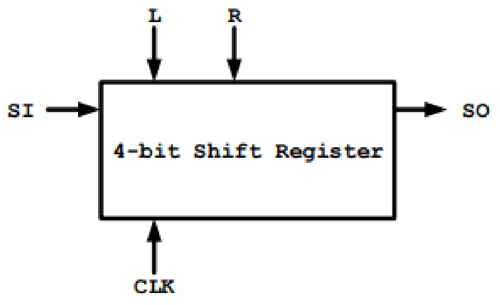
There is asynchronous reset when R = 0. Q becomes 0.



There is a synchronous load when L = 0. Q retains value. It waits for the clock since its synchronous.

3. 2. 3: 4-bit Shift Register with Asynchronous Reset and Synchronous Load

1. Use the new D Flip-Flop in Figure 1 and design a 4-bit scalable bit-sliced synchronous shift register (with SI input and SO output) with synchronous load and asynchronous reset. The symbol and a table that describes the operation of the register are depicted in Figure 2.

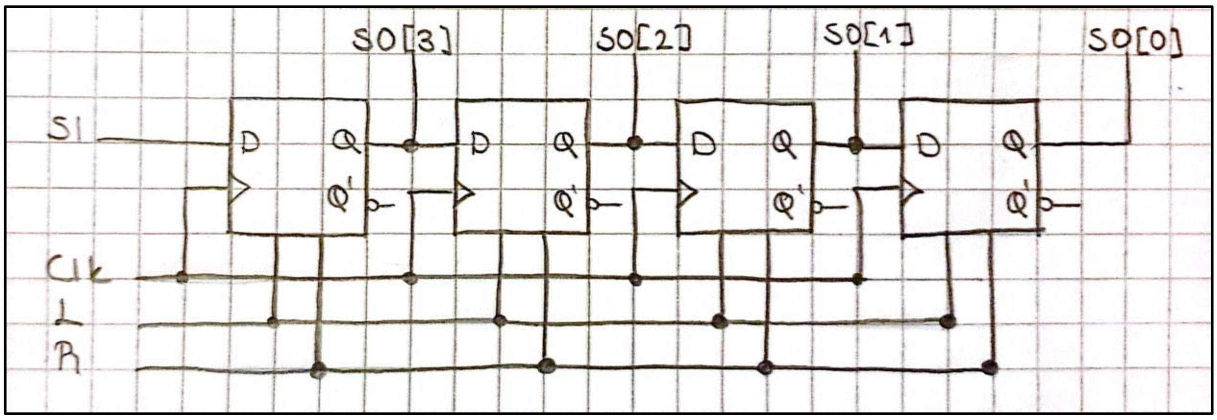


(a)

R	SI	SO
1	X	0000
0	D_1	D_1000
0	D_2	D_2D_100
0	D_3	$D_3D_2D_10$
0	D_4	$D_4D_3D_2D_1$

(b)

Figure 2: (a) Symbol, (b) Table to Describe the 4-bit Shift Register Operation.

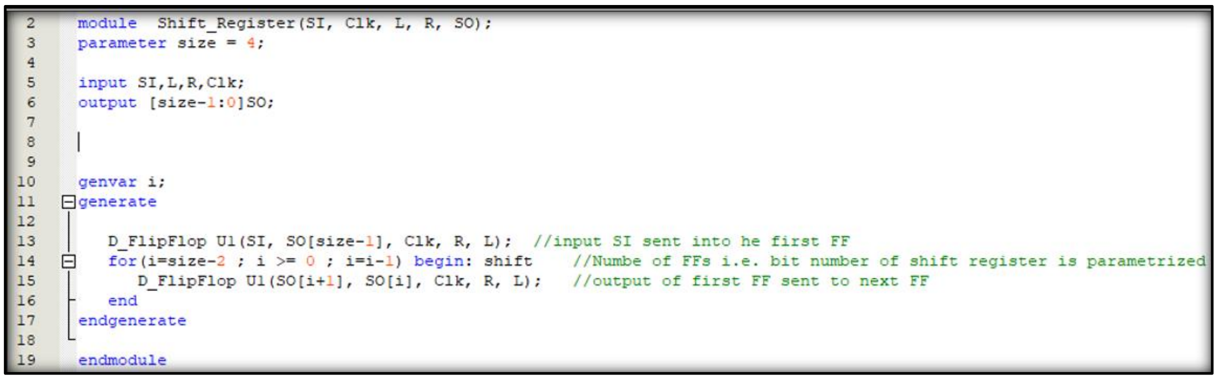


When R is 1 there is a asynchronous reset. When R is 0 SI would either be D1, D2, D3, D4.

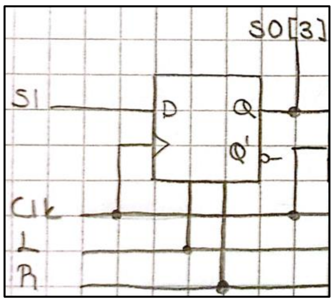
When SI is 1:

With each clock posedge the value would enter the next register. It would shift with each clock.

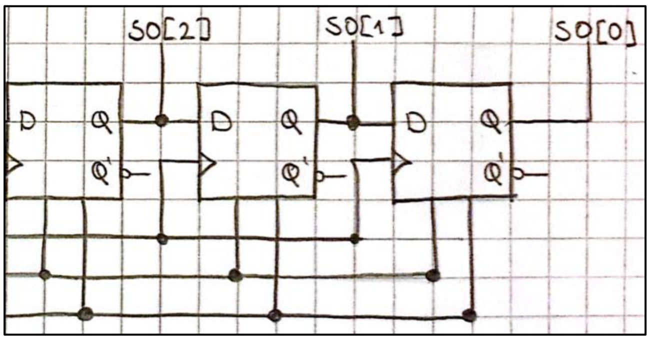
2. The register has four inputs and one output. Write a structural (hierarchical) Verilog HDL code by explicitly declaring the D Flip-Flop created in 3.2.2. **Do not use if statement (behavioural) approach.**



```verilog
2   module  Shift_Register(SI, Clk, L, R, SO);
3   parameter size = 4;
4
5   input SI,L,R,Clk;
6   output [size-1:0]SO;
7
8
9
10  genvar i;
11  generate
12
13     D_FlipFlop U1(SI, SO[size-1], Clk, R, L);  //input SI sent into he first FF
14     for(i=size-2 ; i >= 0 ; i=i-1) begin: shift    //Numbe of FFs i.e. bit number of shift register is parametrized
15        D_FlipFlop U1(SO[i+1], SO[i], Clk, R, L);   //output of first FF sent to next FF
16     end
17  endgenerate
18
19  endmodule
```



For line 13: The first register is entered.

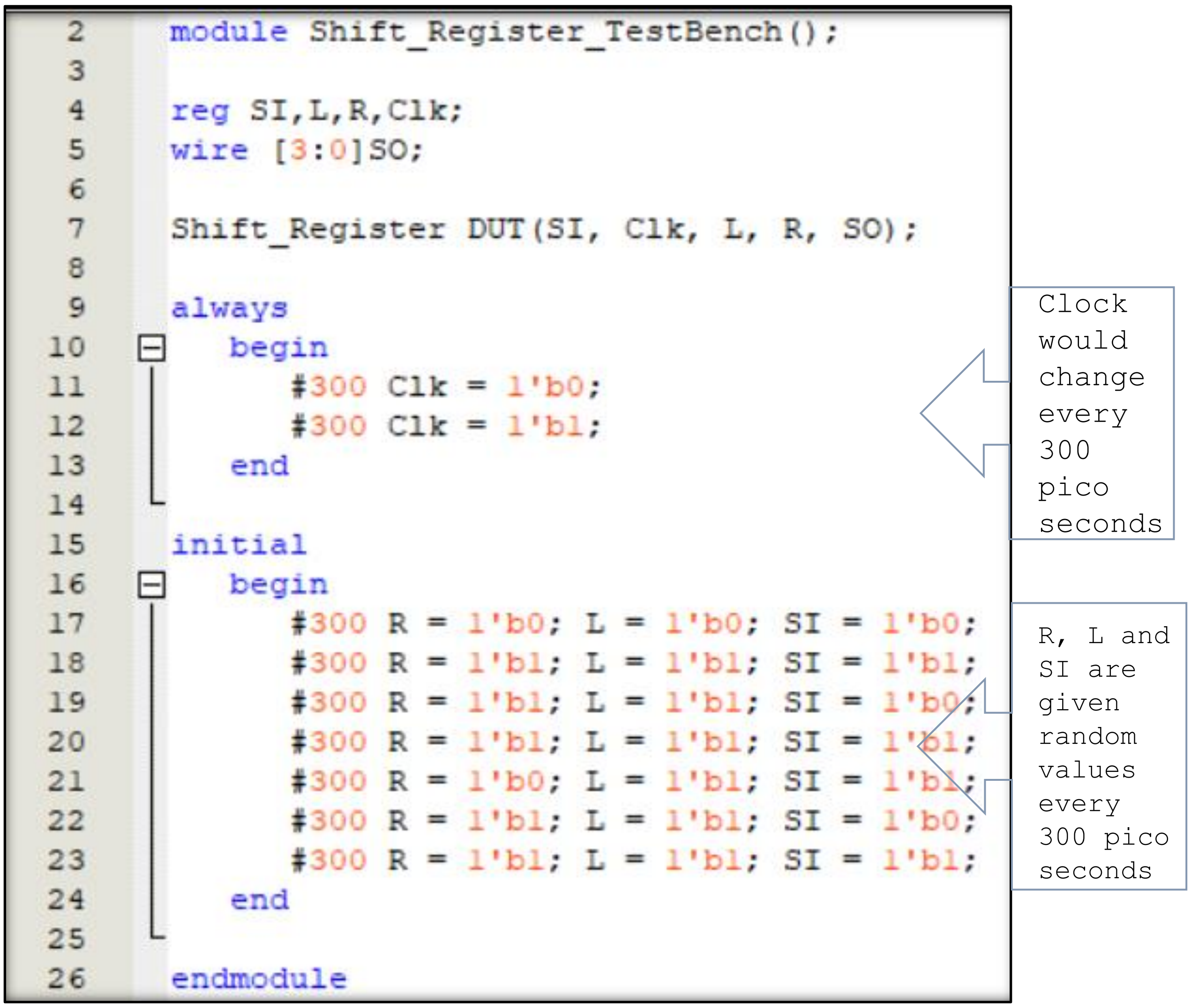


For line 14-16: The middle and last registers are entered.

3. Use the following test patterns with the clock periods provided in Table 1. Write a Verilog Testbench code to simulate and verify the functionality in ModelSim®.

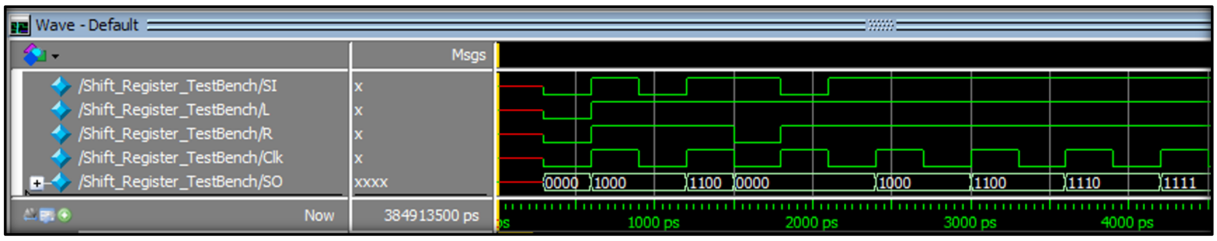
Table 1: Testbench Stimuli

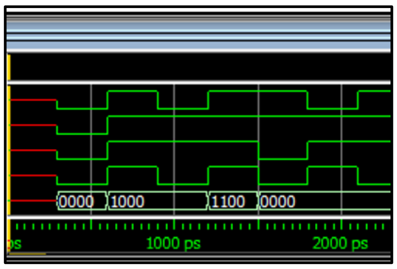
R	L	SI	Clock Period
0	0	0	300 ns
1	1	1	300 ns
1	1	0	300 ns
1	1	1	300 ns
0	1	1	300 ns
1	1	0	300 ns
1	1	1	300 ns



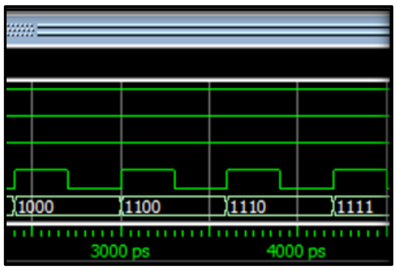
```verilog
module Shift_Register_TestBench();

reg SI,L,R,Clk;
wire [3:0]SO;

Shift_Register DUT(SI, Clk, L, R, SO);

always
   begin
      #300 Clk = 1'b0;
      #300 Clk = 1'b1;
   end

initial
   begin
      #300 R = 1'b0; L = 1'b0; SI = 1'b0;
      #300 R = 1'b1; L = 1'b1; SI = 1'b1;
      #300 R = 1'b1; L = 1'b1; SI = 1'b0;
      #300 R = 1'b1; L = 1'b1; SI = 1'b1;
      #300 R = 1'b0; L = 1'b1; SI = 1'b1;
      #300 R = 1'b1; L = 1'b1; SI = 1'b0;
      #300 R = 1'b1; L = 1'b1; SI = 1'b1;
   end

endmodule
```

Clock would change every 300 pico seconds
R, L and SI are given random values every 300 pico seconds


Wave - Default
Msgs
/Shift_Register_TestBench/SI
/Shift_Register_TestBench/L
/Shift_Register_TestBench/R
/Shift_Register_TestBench/Clk
/Shift_Register_TestBench/SO
x
x
x
x
xxxx
Now
384913500 ps
0000
1000
1100
0000
1000
1100
1110
1111
1000 ps
2000 ps
3000 ps
4000 ps



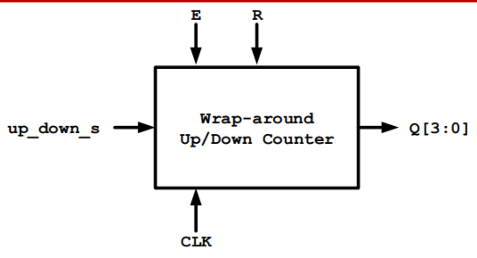
First the data needs to be loaded. After it was loaded the shift register starts with 0000. Then it shifts to 1000 then to 1100. Goes back to 0000 with the Reset.



The reigsters keep shifting as Reset and Load stay at 1.

3. 2. 4: 4-bit Synchronous Wrap-Around Up/Down Counter with Synchronous Up/Down Select, Count-Enable and Asynchronous Reset

1. Write a structural (hierarchical) Verilog HDL code by explicitly declaring the D Flip-Flop to create a scalable design of a 4-bit synchronous wrap-around up/down counter with synchronous up/down select and count-enable controls, and asynchronous reset as depicted in Figure 3. Do not use if statement (behavioural) approach.



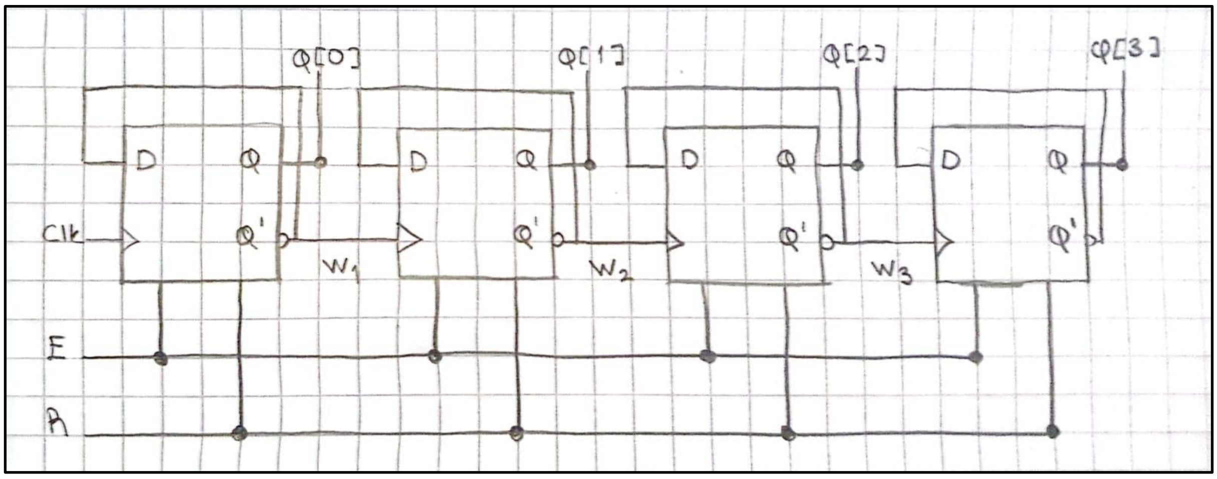
(a)

R	up_down_s	E	Q[3:0]
1	X	X	0000
0	X	0	Q = Q
0	1	1	Q = Q+1
0	0	1	Q = Q-1

(b)

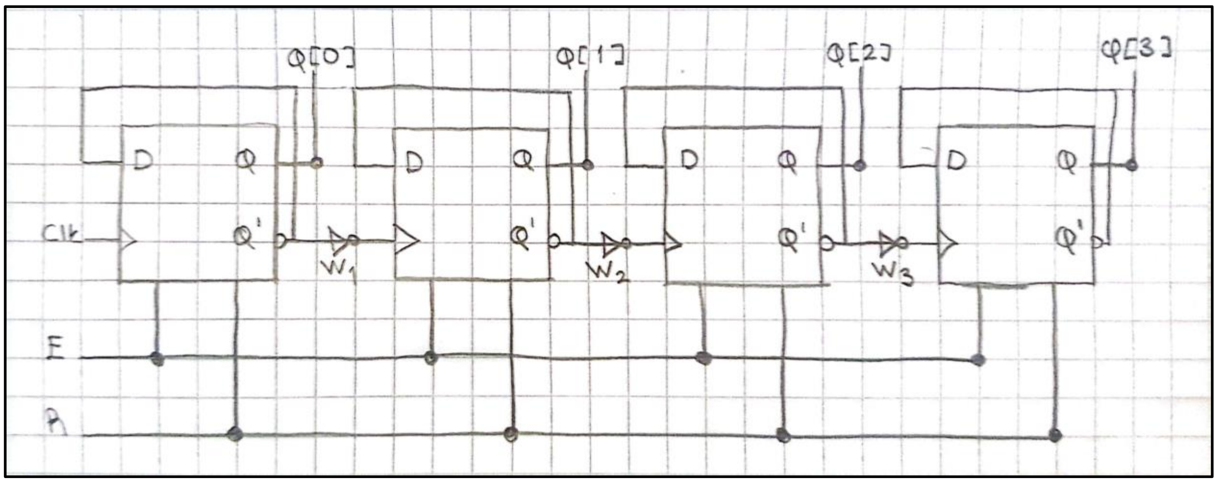
Figure 3: (a) Wrap-around Up/Down Counter Symbol, (b) Table to Describe the Wrap-around Up/Down Counter Operation.

Up counter:

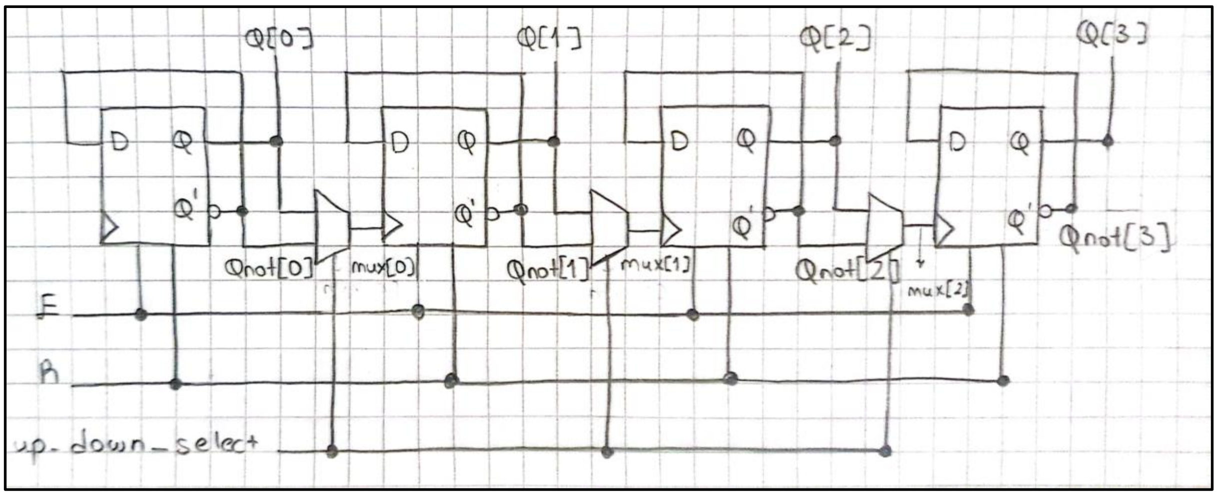


E would function as the Load this time. This time the reset would happen when R is 1, not 0.

Down Counter:



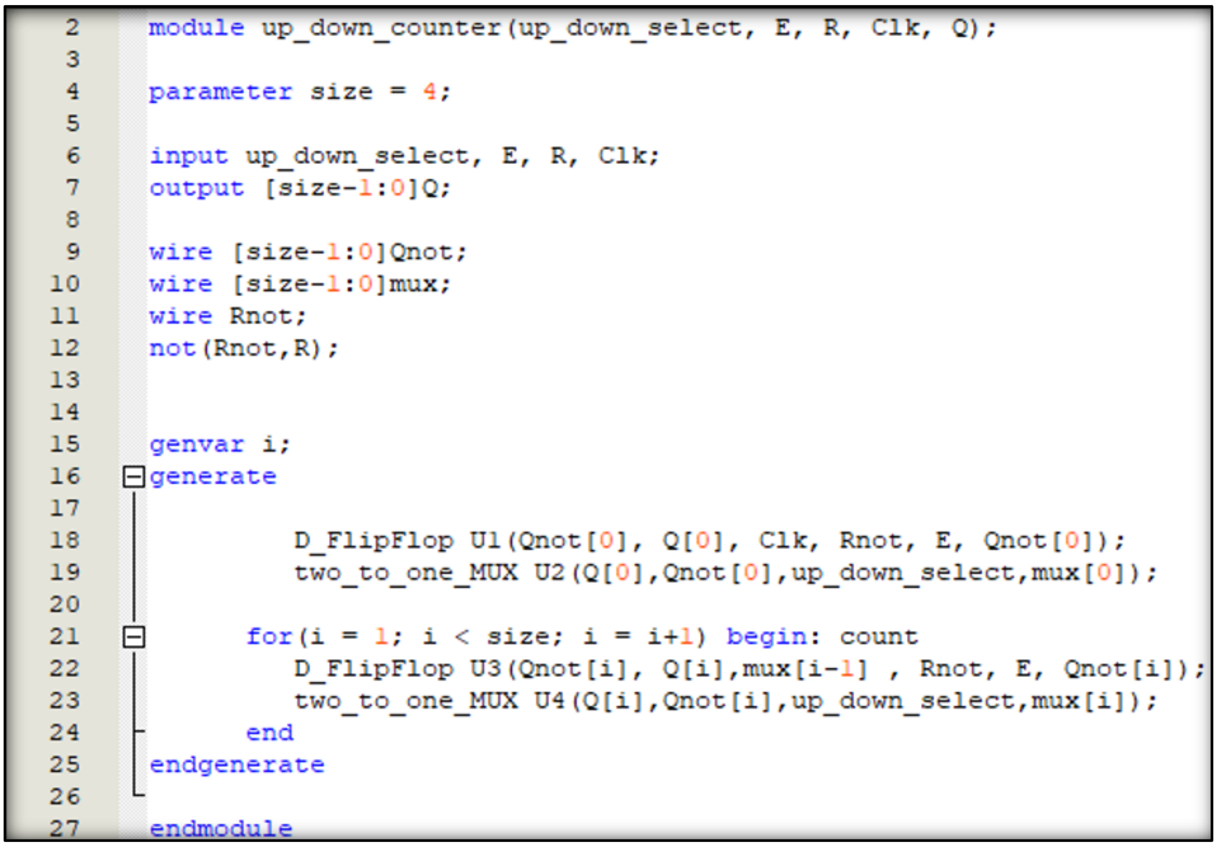
Combine the two:



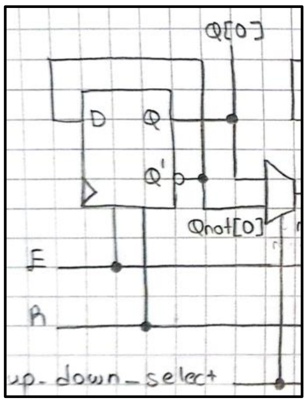
With the multiplexer we can now select a down or up counter with the up_down_select.

```
If R is 1:
      Q = 0000
Else R is 0:
      If E is 0:
            Q is the same
      Else E is 1:
            If up_down_s is 1:
                  Q is Q + 1: count up
            Else up_down_s is 0:
                  Q is Q – 1: count down
```

Verilog implementation:

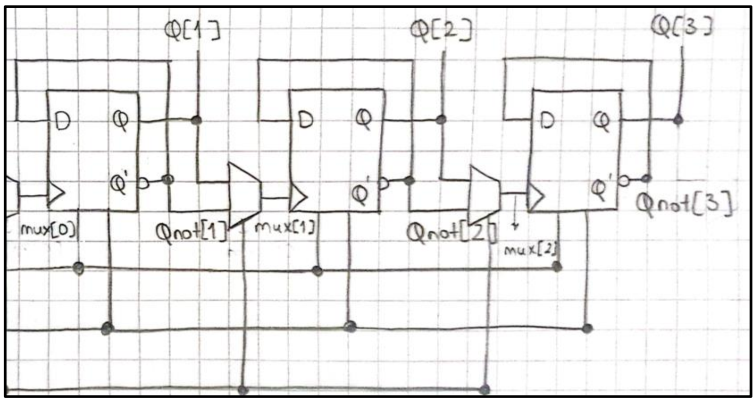


```
2    module up_down_counter(up_down_select, E, R, Clk, Q);
3
4    parameter size = 4;
5
6    input up_down_select, E, R, Clk;
7    output [size-1:0]Q;
8
9    wire [size-1:0]Qnot;
10   wire [size-1:0]mux;
11   wire Rnot;
12   not(Rnot,R);
13
14
15   genvar i;
16   generate
17
18           D_FlipFlop U1(Qnot[0], Q[0], Clk, Rnot, E, Qnot[0]);
19           two_to_one_MUX U2(Q[0],Qnot[0],up_down_select,mux[0]);
20
21       for(i = 1; i < size; i = i+1) begin: count
22           D_FlipFlop U3(Qnot[i], Q[i],mux[i-1] , Rnot, E, Qnot[i]);
23           two_to_one_MUX U4(Q[i],Qnot[i],up_down_select,mux[i]);
24       end
25   endgenerate
26
27   endmodule
```



In Line 18: The first Flip Flop is entered.

In Line 19: The multiplexer is entered.



In Line 21 to 24: The rest of the flip flops and multiplexers are entered.

2. Modify your code to connect a 4-bit output to 7-segment decoder designed and used in LAB 2&3.

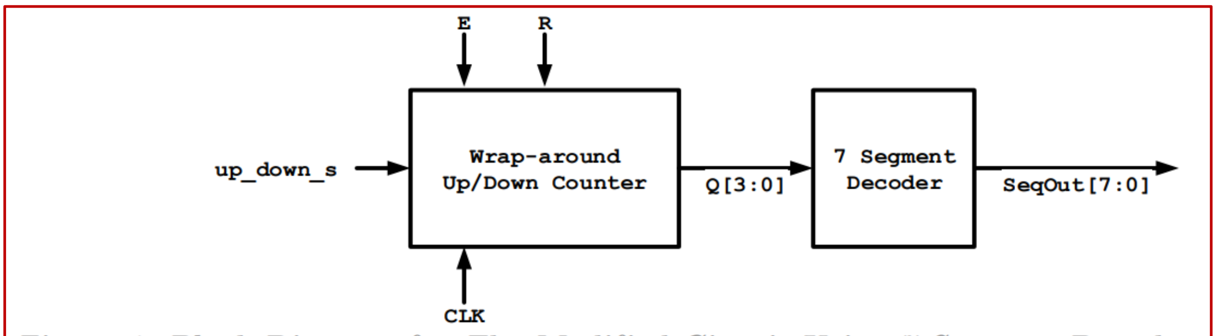
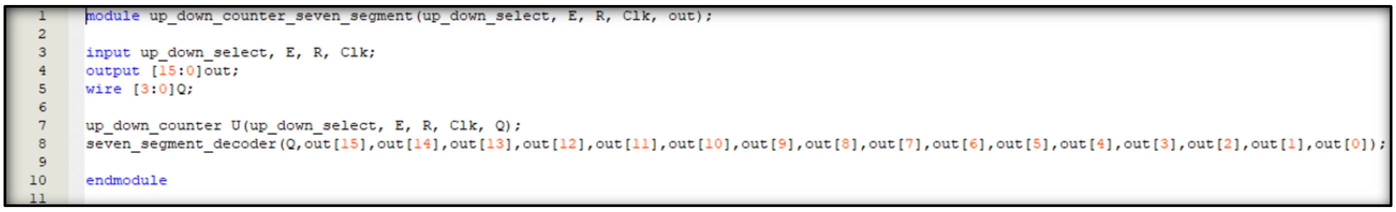


Figure 4: Block Diagram for The Modified Circuit Using 7 Segment Decoder.



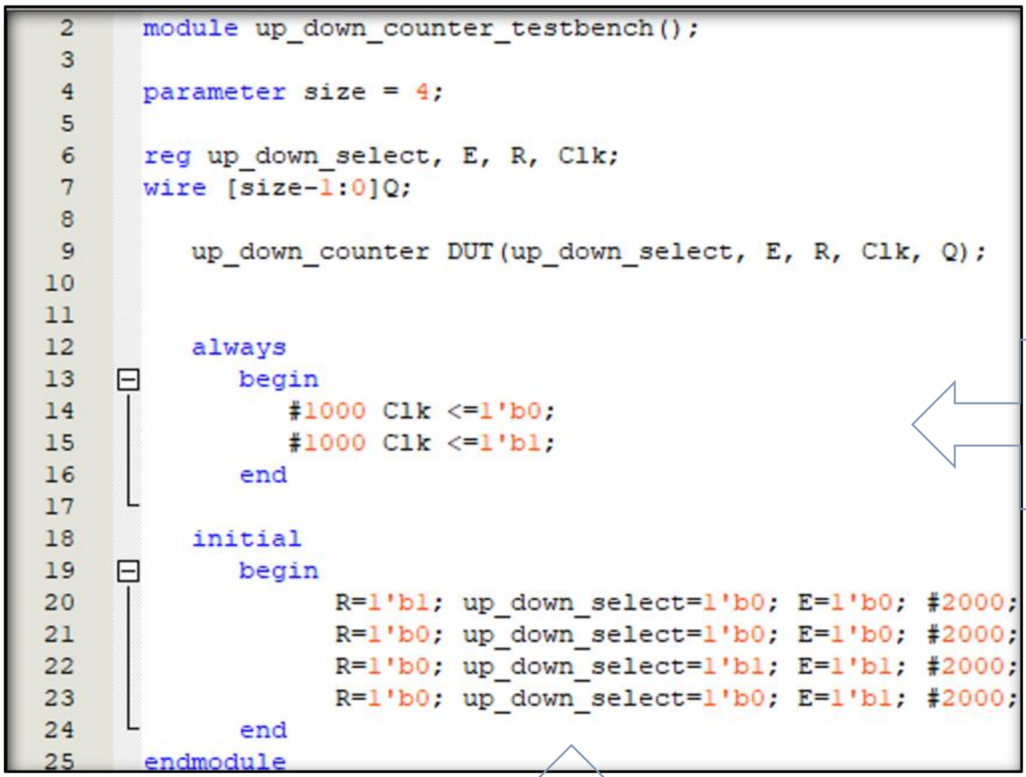
```verilog
module up_down_counter_seven_segment(up_down_select, E, R, Clk, out);

input up_down_select, E, R, Clk;
output [15:0]out;
wire [3:0]Q;

up_down_counter U(up_down_select, E, R, Clk, Q);
seven_segment_decoder(Q,out[15],out[14],out[13],out[12],out[11],out[10],out[9],out[8],out[7],out[6],out[5],out[4],out[3],out[2],out[1],out[0]);

endmodule
```

In this code, 4-bit output of the counter is fed into the 4-bit input of the seven-segment decoder in a top-level code. Therefore output of the counter will be displayed in the seven-segment display of the board thanks to the decoder code written in the previous project.

3. Use the following test patterns with the clock periods provided in Table 2. Write a Verilog Testbench code to simulate and verify the functionality in ModelSim®.

Table 2: Testbench Stimuli

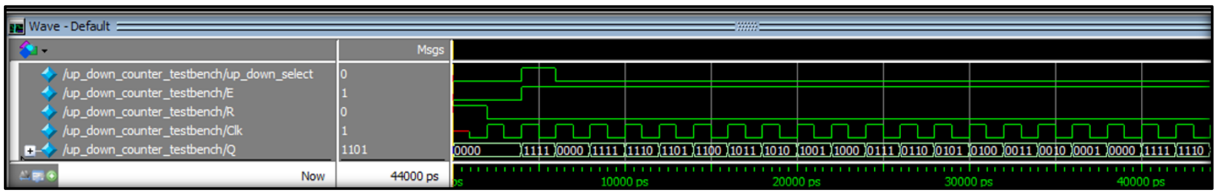
R	up_down_s	E	Clock Period
1	0	0	2000 ns
0	0	0	2000 ns
0	1	1	2000 ns
0	0	1	2000 ns

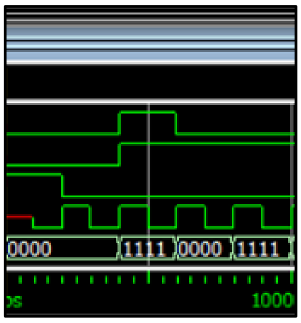


```verilog
module up_down_counter_testbench();

parameter size = 4;

reg up_down_select, E, R, Clk;
wire [size-1:0]Q;

   up_down_counter DUT(up_down_select, E, R, Clk, Q);


   always
      begin
         #1000 Clk <=1'b0;
         #1000 Clk <=1'b1;
      end

   initial
      begin
            R=1'b1; up_down_select=1'b0; E=1'b0; #2000;
            R=1'b0; up_down_select=1'b0; E=1'b0; #2000;
            R=1'b0; up_down_select=1'b1; E=1'b1; #2000;
            R=1'b0; up_down_select=1'b0; E=1'b1; #2000;
      end
endmodule
```

The clock will change every 1000 pico seconds

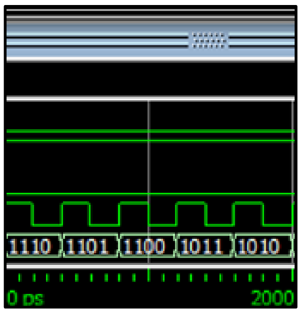
R needs to be 1 first to start the counting operation. R would set the initial value to be 0000. Up_down select and E values are random. R, up_down_select and E change every 2000 pico seconds.


Wave - Default
Msgs
/up_down_counter_testbench/up_down_select 0
/up_down_counter_testbench/E 1
/up_down_counter_testbench/R 0
/up_down_counter_testbench/Clk 1
/up_down_counter_testbench/Q 1101
Now 44000 ps
0000 1111 0000 1111 1110 1101 1100 1011 1010 1001 1000 0111 0110 0101 0100 0011 0010 0001 0000 1111 1110
10000 ps
20000 ps
30000 ps
40000 ps

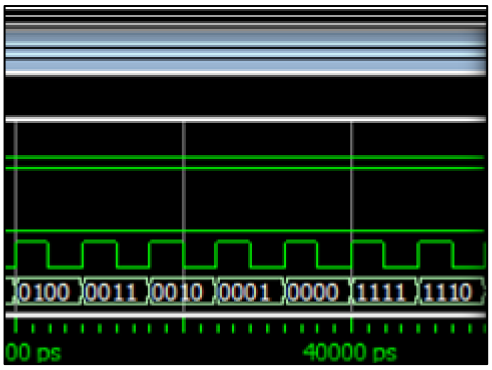


First R is 1 so Q would be 0000. When up_down_select is rising, there is a down count operation 0000 -> 1111.

up_down_select is falling, there is a up count operation 1111->0000.



Continues counting.

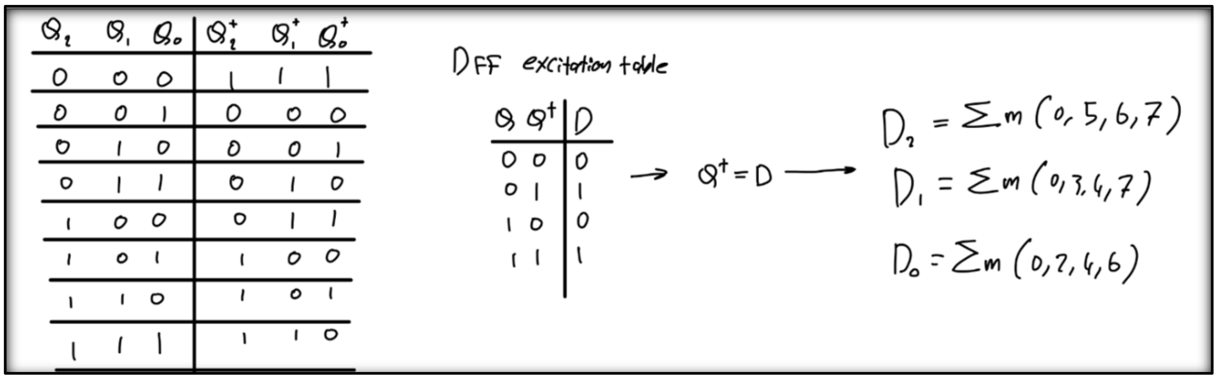


When Q reaches 0000 it goes back to 1111.

3. 2. 5: 3-bit Down Counter Using D Flip-Flop

Design a 3-bit down counter with asynchronous clear using rising-edge triggered D flip-flops. The count should wrap around from "000" to "111"

State table is constructed.



Q_2	Q_1	Q_0	Q_2^+	Q_1^+	Q_0^+
0	0	0	1	1	1
0	0	1	0	0	0
0	1	0	0	0	1
0	1	1	0	1	0
1	0	0	0	1	1
1	0	1	1	0	0
1	1	0	1	0	1
1	1	1	1	1	0

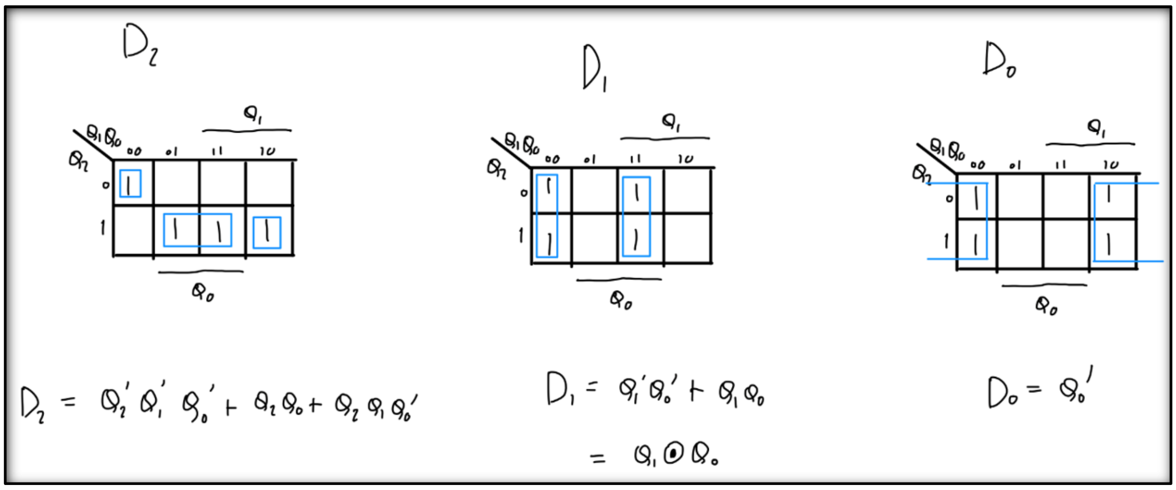
DFF excitation table

Q	Q^+	D
0	0	0
0	1	1
1	0	0
1	1	1

$$\rightarrow Q^+ = D \longrightarrow$$

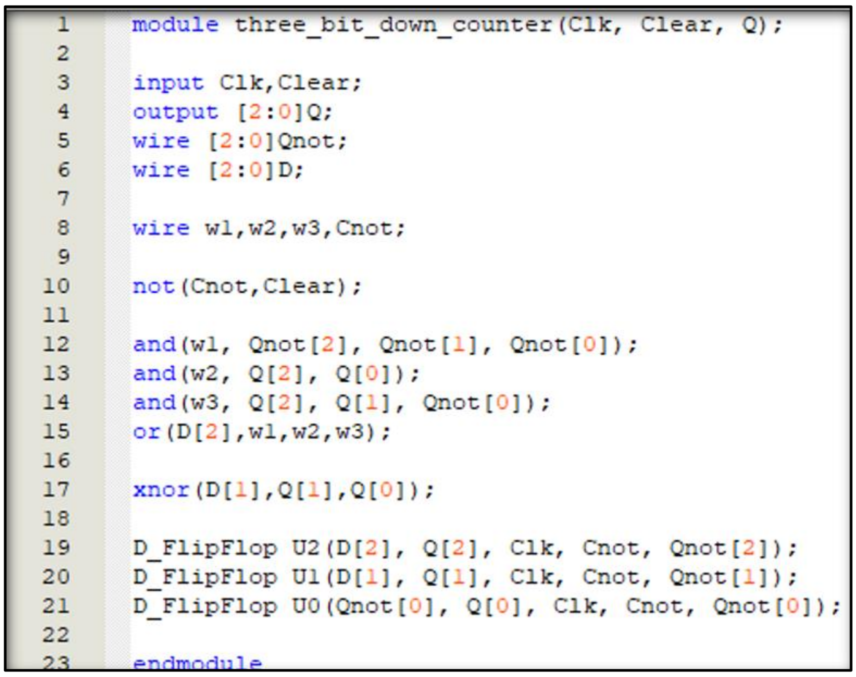
$$D_2 = \Sigma m(0,5,6,7)$$
$$D_1 = \Sigma m(0,3,4,7)$$
$$D_0 = \Sigma m(0,2,4,6)$$

KMAP's are created to find formulas.



D2 = Q2'Q1'Q0' + Q2Q0 + Q2Q1Q0'
D1 = Q1'Q0' + Q1Q0 = Q1 xnor Q0

D0 = Q0'



```
1   module three_bit_down_counter(Clk, Clear, Q);
2
3   input Clk,Clear;
4   output [2:0]Q;
5   wire [2:0]Qnot;
6   wire [2:0]D;
7
8   wire w1,w2,w3,Cnot;
9
10  not(Cnot,Clear);
11
12  and(w1, Qnot[2], Qnot[1], Qnot[0]);
13  and(w2, Q[2], Q[0]);
14  and(w3, Q[2], Q[1], Qnot[0]);
15  or(D[2],w1,w2,w3);
16
17  xnor(D[1],Q[1],Q[0]);
18
19  D_FlipFlop U2(D[2], Q[2], Clk, Cnot, Qnot[2]);
20  D_FlipFlop U1(D[1], Q[1], Clk, Cnot, Qnot[1]);
21  D_FlipFlop U0(Qnot[0], Q[0], Clk, Cnot, Qnot[0]);
22
23  endmodule
```

Line 12: w1 = Q2′Q1′Q0′

Line 13: w2 = Q2Q0

Line 14: w3 = Q2Q1Q0′

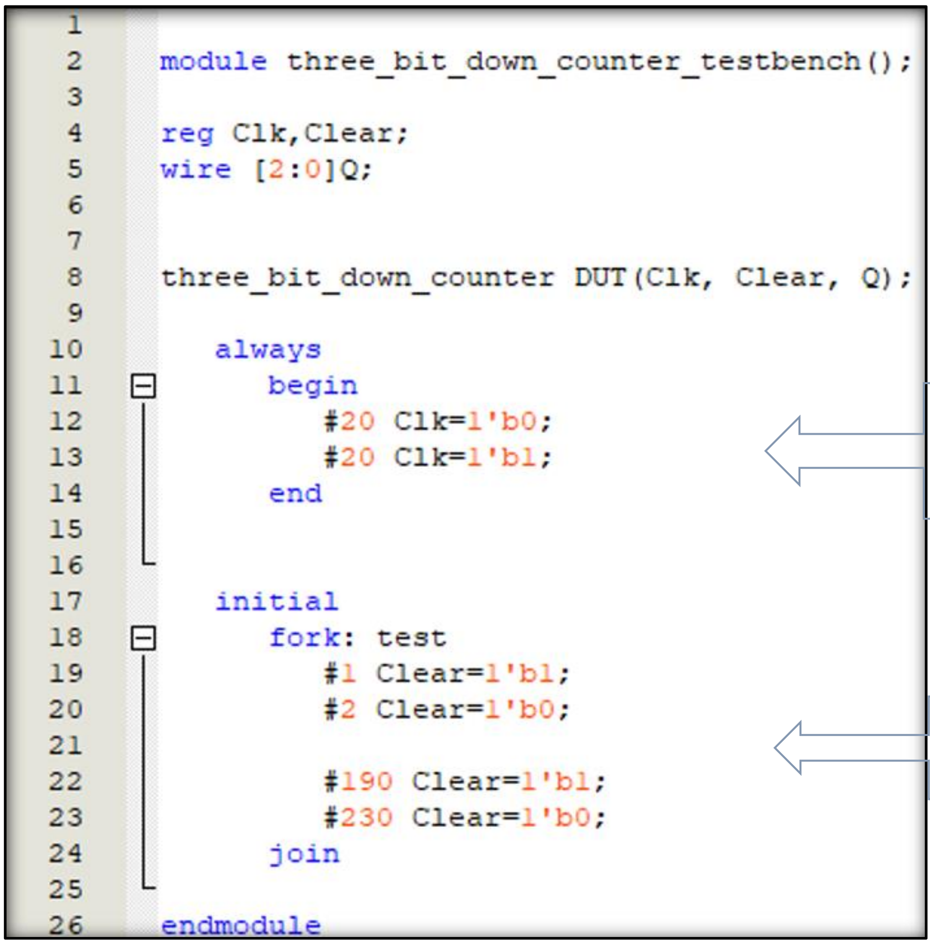
Line 15: D2 = w1 + w2 + w3

Line 17: D1 = Q1 xnor Q0

Line 19: flipflop D2

Line 20: flipflop D1

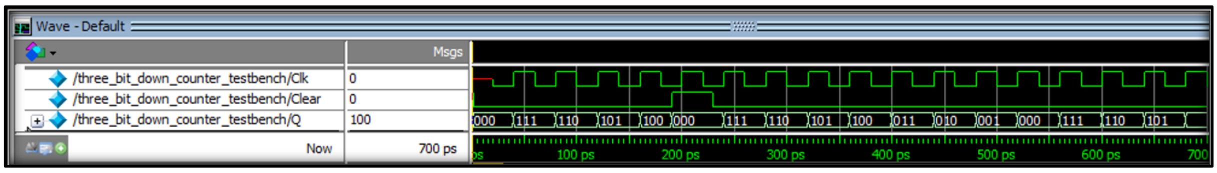
Line 21: flipflop D0



```

module three_bit_down_counter_testbench();

reg Clk,Clear;
wire [2:0]Q;


three_bit_down_counter DUT(Clk, Clear, Q);

   always
      begin
         #20 Clk=1'b0;
         #20 Clk=1'b1;
      end


   initial
      fork: test
         #1 Clear=1'b1;
         #2 Clear=1'b0;

         #190 Clear=1'b1;
         #230 Clear=1'b0;
      join

endmodule
```

Clk changes every 20 pico seconds

Clear changes randomly.



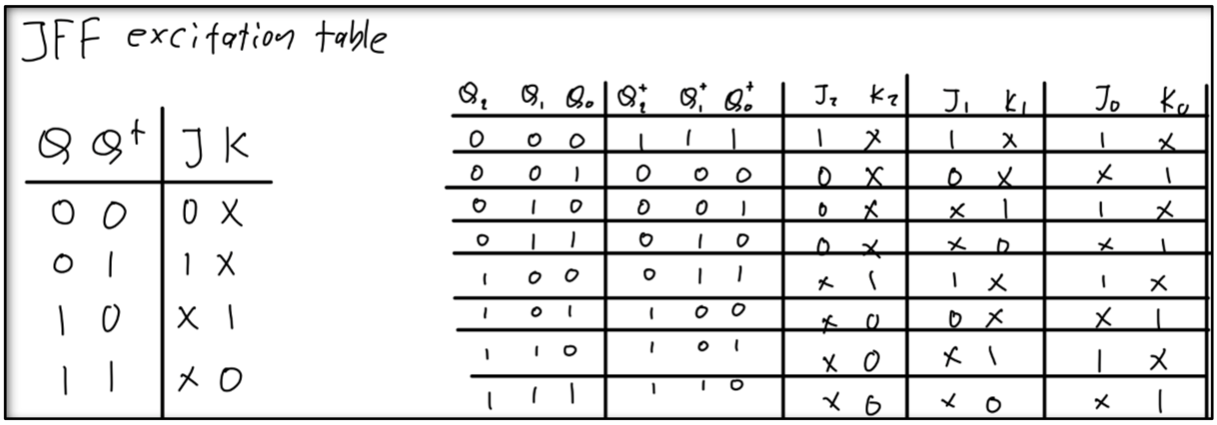
000 -> 111 -> 110 -> 101 and so on.

Between 190-230ps we can see the functionality of asynchronous clear. In this interval, output becomes 000 independent of the clock. After this interval, count down start again from 000.

3. 2. 6: 3-bit Down Counter using JK Flip-Flop

Design a 3-bit down counter with asynchronous clear using negative-edge triggered JK-type flip-flops with asynchronous active low clear.

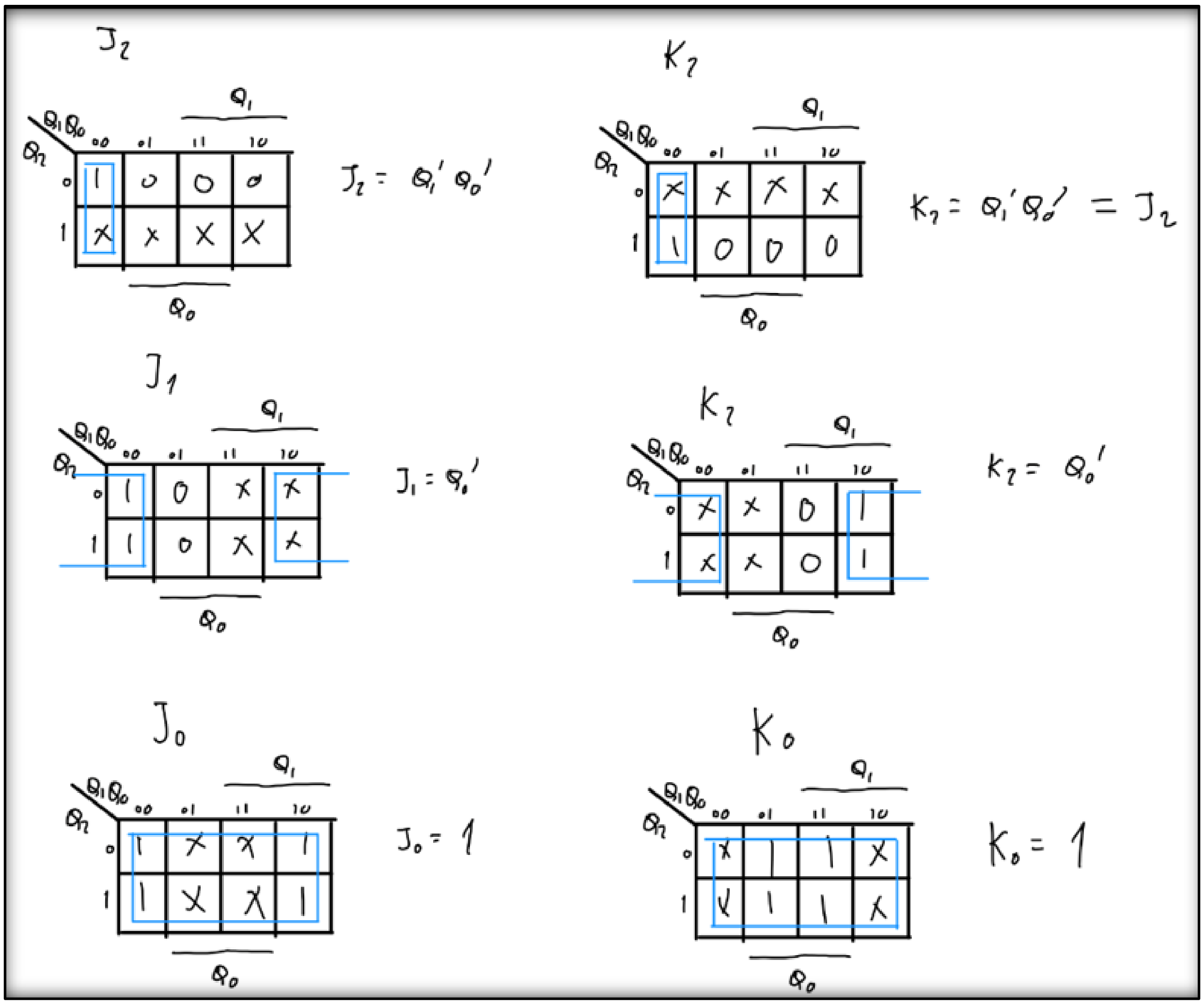
Similarly to 3. 2. 5 we create a table.


JFF excitation table

Q	Q^+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Q_2	Q_1	Q_0	Q_2^+	Q_1^+	Q_0^+	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	1	1	1	1	X	1	X	1	X
0	0	1	0	0	0	0	X	0	X	X	1
0	1	0	0	0	1	0	X	X	1	1	X
0	1	1	0	1	0	0	X	X	0	X	1
1	0	0	0	1	1	X	1	1	X	1	X
1	0	1	1	0	0	X	0	0	X	X	1
1	1	0	1	0	1	X	0	X	1	1	X
1	1	1	1	1	0	X	0	X	0	X	1

Then we create KMAP's to find the formulas.



J2 = Q1'Q0'

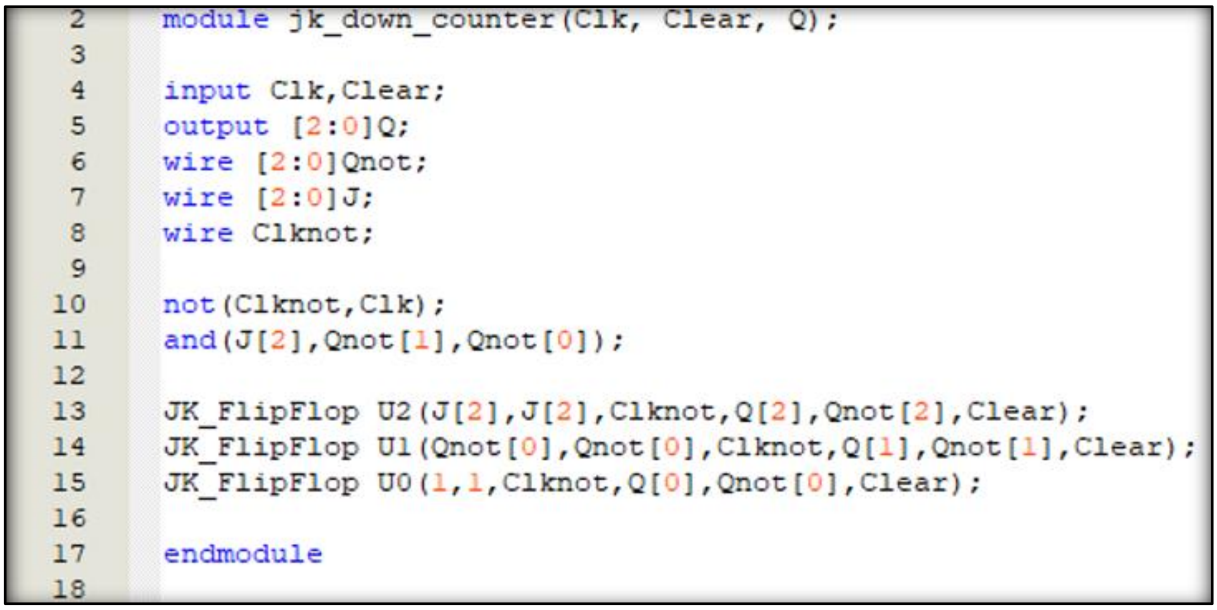
J1 = Q0'

J0 = 1

K2 = Q1'Q0' = J2

K2 = Q0' = J1

K0 = 1 = J0



```
2    module jk_down_counter(Clk, Clear, Q);
3
4    input Clk,Clear;
5    output [2:0]Q;
6    wire [2:0]Qnot;
7    wire [2:0]J;
8    wire Clknot;
9
10   not(Clknot,Clk);
11   and(J[2],Qnot[1],Qnot[0]);
12
13   JK_FlipFlop U2(J[2],J[2],Clknot,Q[2],Qnot[2],Clear);
14   JK_FlipFlop U1(Qnot[0],Qnot[0],Clknot,Q[1],Qnot[1],Clear);
15   JK_FlipFlop U0(1,1,Clknot,Q[0],Qnot[0],Clear);
16
17   endmodule
18
```

Line 11: J2 = Q1’Q0’

Line 13: Flip Flop J2

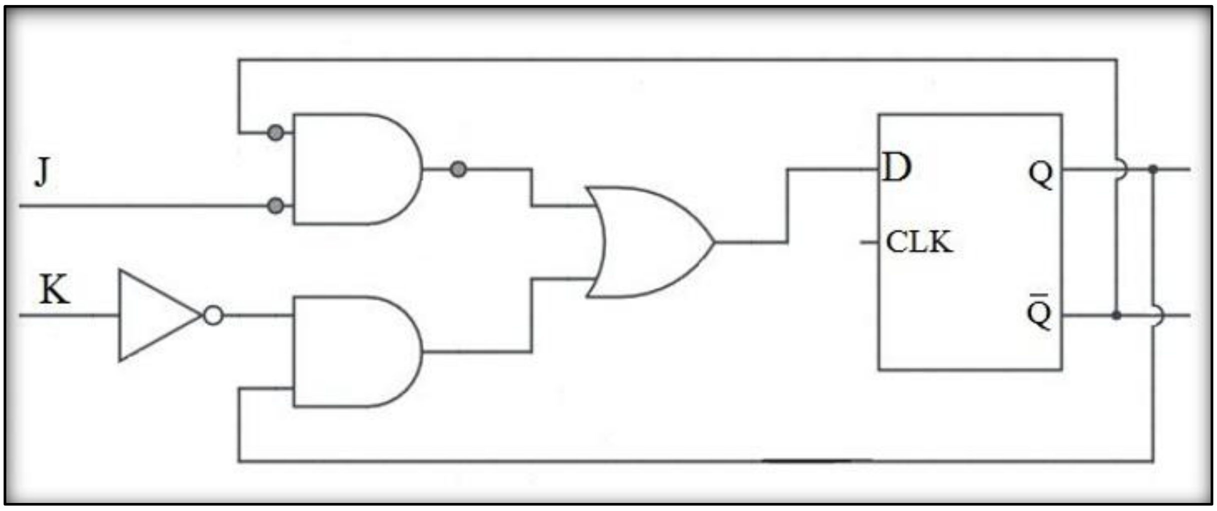
Line 14: Flip Flop J1

Line 15: Flip Flop J0

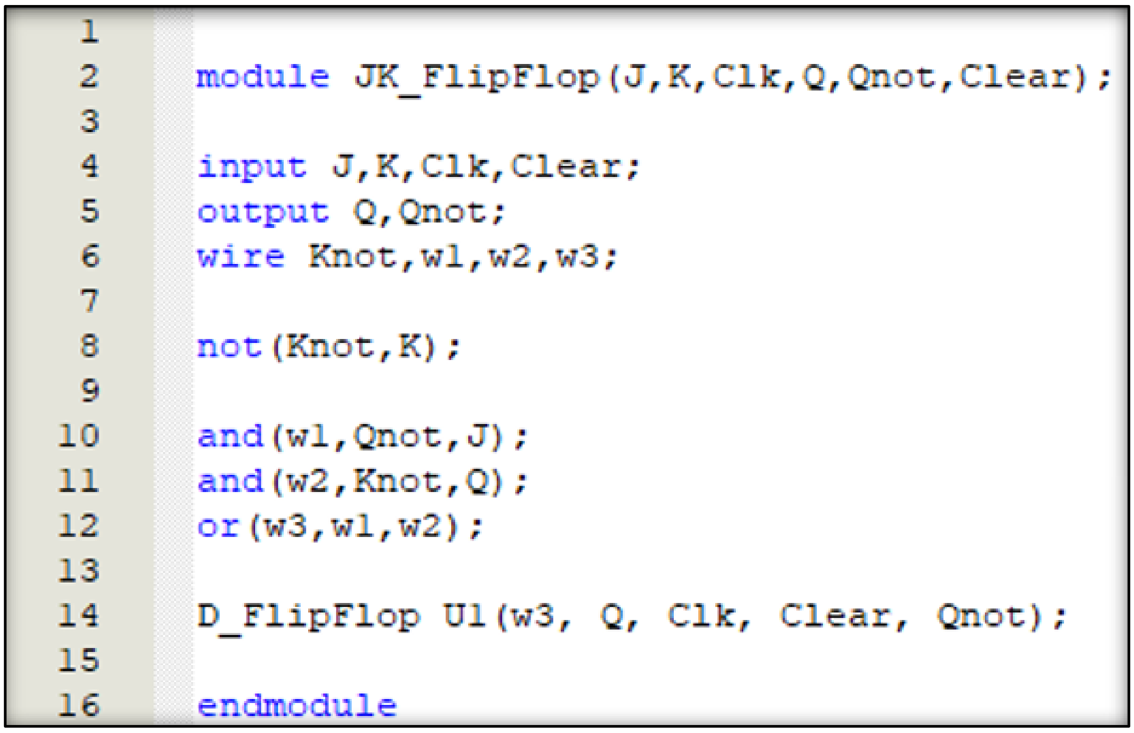
Line 10: Clock is inverted using a NOT gate in order to achieve negative edge JKFF.

Unlike the counter with DFFs, clear isn’t inverted to achieve active low asynchronous clear input.

JK flip flop using D flip flop:



D = J*Q’ + K’*Q (formula of JK flip flop)



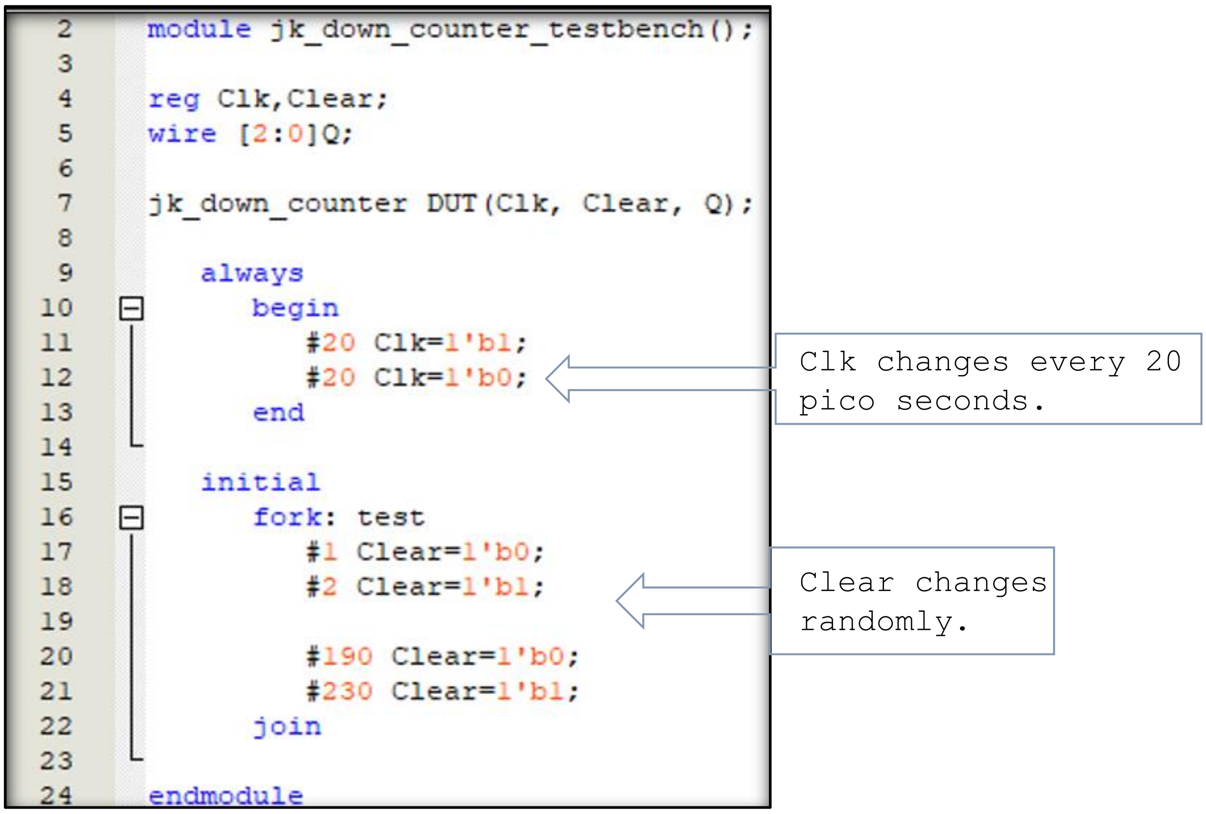
```
 1
 2   module JK_FlipFlop(J,K,Clk,Q,Qnot,Clear);
 3
 4   input J,K,Clk,Clear;
 5   output Q,Qnot;
 6   wire Knot,w1,w2,w3;
 7
 8   not(Knot,K);
 9
10   and(w1,Qnot,J);
11   and(w2,Knot,Q);
12   or(w3,w1,w2);
13
14   D_FlipFlop U1(w3, Q, Clk, Clear, Qnot);
15
16   endmodule
```

Line 10: w1 = J*Q’.

Line 11: w2 = K’*Q.

Line 12: w3 = w1 + w2 = J*Q’ + K’*Q.

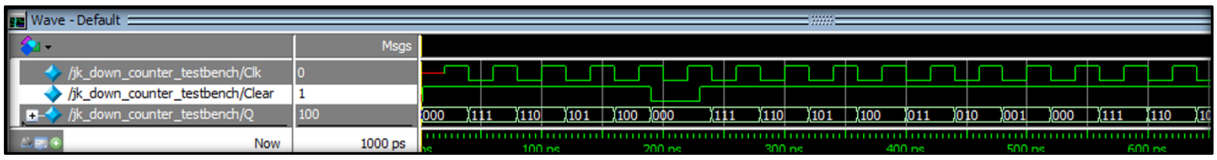
Line 13: D flip flop with input w3. D = w3 = J*Q + K’*Q.



```
module jk_down_counter_testbench();

reg Clk,Clear;
wire [2:0]Q;

jk_down_counter DUT(Clk, Clear, Q);

   always
      begin
         #20 Clk=1'b1;
         #20 Clk=1'b0;
      end

   initial
      fork: test
         #1 Clear=1'b0;
         #2 Clear=1'b1;

         #190 Clear=1'b0;
         #230 Clear=1'b1;
      join

endmodule
```

Clk changes every 20 pico seconds.

Clear changes randomly.

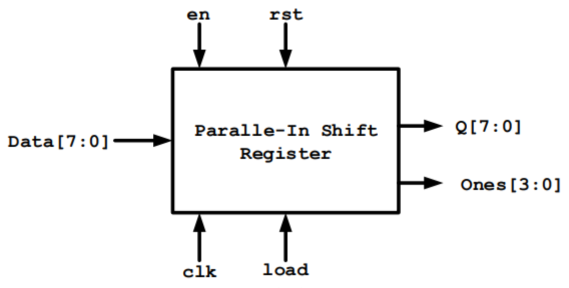


Output starts from 000 and counts down with every falling edge of the clock.

Between 190-230ps we can see the functionality of the active low asynchronous clear. In this interval, output becomes 000 independent of the clock. After this interval, count down start again from 000 and wraps around at 560ps.

3. 2. 7: 4-bit Synchronous Parallel Load Shift Register with Counter and Asynchronous Reset

1. Write a structural (hierarchical) Verilog code to read an 8-bit message from parallel input (user switches) into a shift-register and use the shift function together with a counter to determine the number of '1's in the message. You have to create a top-level design and initialize shift register and counter explicitly. Do not use if statement (behavioural) approach. (no need to design the 7-segment LED decoder here, the decoder will be used during the experimental work.)

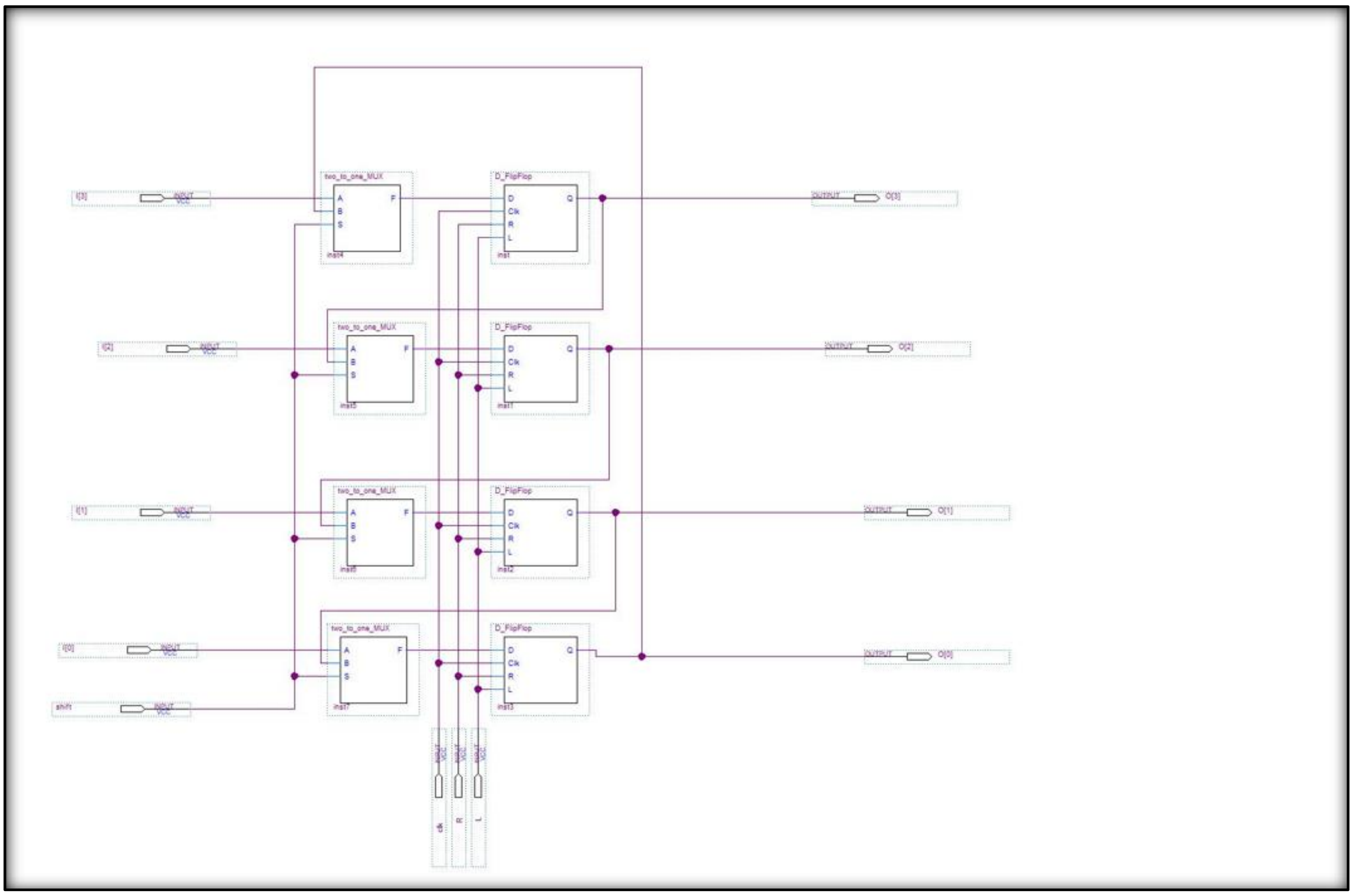


(a)

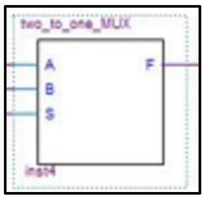
rst	load	en	Q[7:0]	Ones[3:0]
1	X	X	0	0
0	X	0	0	0
0	1	1	Data	0
0	0	1	Data	#ones

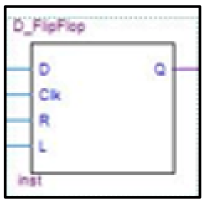
(b)

Figure 5:(a) Parallel Load Shift Register Symbol, (b) table to describe the operation



Circuit in the above schematic is used to implement the parallel load shift register. (the schematic is prepared with 4 DFFs in order to fit the image, actual circuit was prepared with 8 DFFs.)





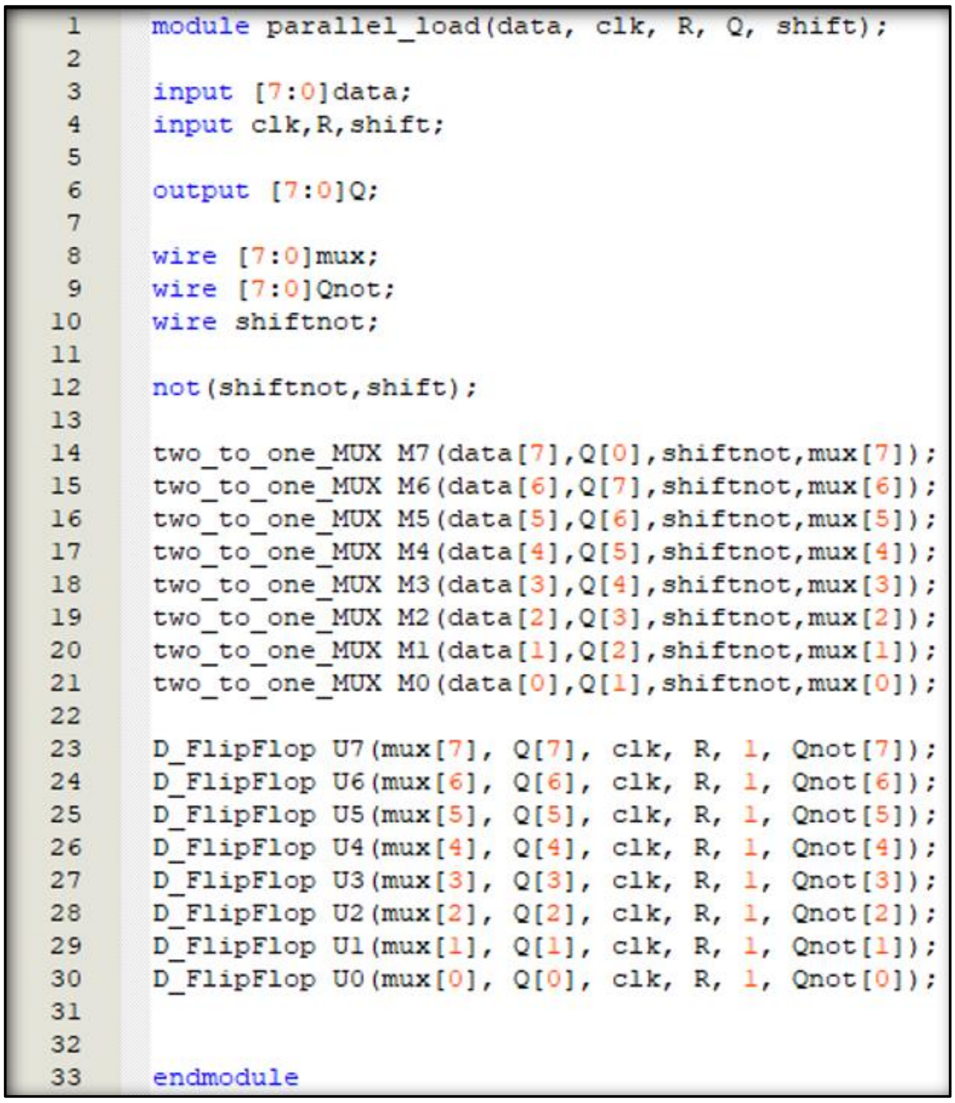
A regular parallel load register is customized by adding MUXs to the inputs.

These MUXs select between input and the above DFF's output. Thus, when the MUX select ("shift" input in this circuit) is 1, values of the DFFs are shifted.

Output of the DFF that holds the least significant bit is then connected to the enable input of an up counter.

This way, when the values in the DFFs are shifted 8 times, 0 and 1 values that are stored in the DFFs are fed to the enable of the up counter one by one. Therefore, up counter counts up the exact number of times it detects "1" in the enable input.

As a result, we achieve the "number of ones" operation.

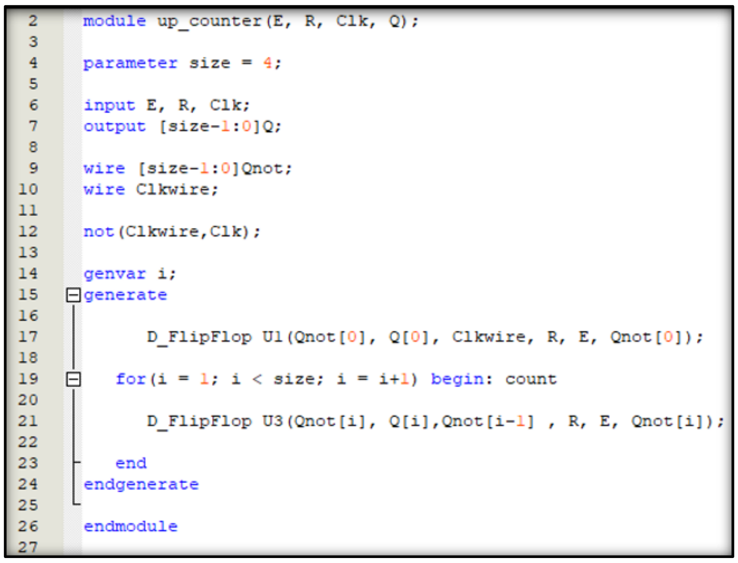


```
 1  module parallel_load(data, clk, R, Q, shift);
 2
 3  input  [7:0]data;
 4  input clk,R,shift;
 5
 6  output [7:0]Q;
 7
 8  wire  [7:0]mux;
 9  wire  [7:0]Qnot;
10  wire shiftnot;
11
12  not(shiftnot,shift);
13
14  two_to_one_MUX M7(data[7],Q[0],shiftnot,mux[7]);
15  two_to_one_MUX M6(data[6],Q[7],shiftnot,mux[6]);
16  two_to_one_MUX M5(data[5],Q[6],shiftnot,mux[5]);
17  two_to_one_MUX M4(data[4],Q[5],shiftnot,mux[4]);
18  two_to_one_MUX M3(data[3],Q[4],shiftnot,mux[3]);
19  two_to_one_MUX M2(data[2],Q[3],shiftnot,mux[2]);
20  two_to_one_MUX M1(data[1],Q[2],shiftnot,mux[1]);
21  two_to_one_MUX M0(data[0],Q[1],shiftnot,mux[0]);
22
23  D_FlipFlop U7(mux[7], Q[7], clk, R, 1, Qnot[7]);
24  D_FlipFlop U6(mux[6], Q[6], clk, R, 1, Qnot[6]);
25  D_FlipFlop U5(mux[5], Q[5], clk, R, 1, Qnot[5]);
26  D_FlipFlop U4(mux[4], Q[4], clk, R, 1, Qnot[4]);
27  D_FlipFlop U3(mux[3], Q[3], clk, R, 1, Qnot[3]);
28  D_FlipFlop U2(mux[2], Q[2], clk, R, 1, Qnot[2]);
29  D_FlipFlop U1(mux[1], Q[1], clk, R, 1, Qnot[1]);
30  D_FlipFlop U0(mux[0], Q[0], clk, R, 1, Qnot[0]);
31
32
33  endmodule
```

Q output of the least significant DFF is fed into the B input of the most significant DFF MUX.

Line 14 to 21: 8 mux's are entered.

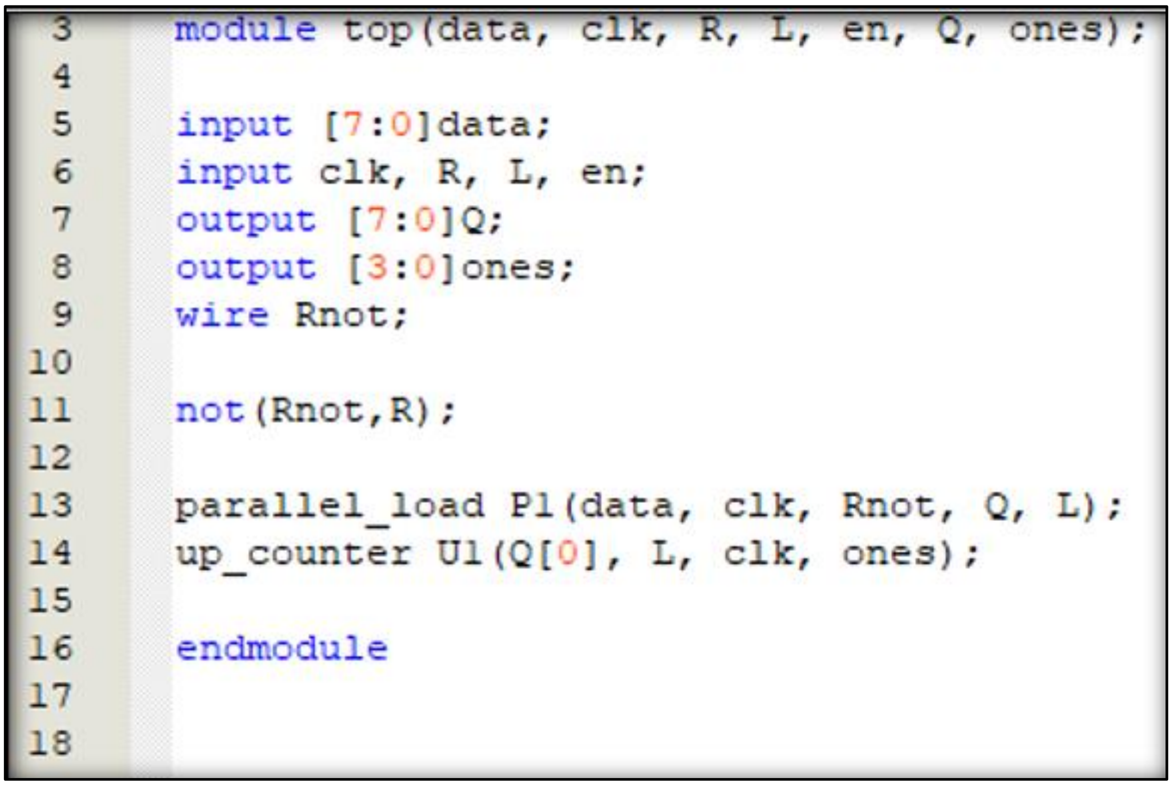
Line 23 to 30: 8 flip flops are entered



```
module up_counter(E, R, Clk, Q);

parameter size = 4;

input E, R, Clk;
output [size-1:0]Q;

wire [size-1:0]Qnot;
wire Clkwire;

not(Clkwire,Clk);

genvar i;
generate

      D_FlipFlop U1(Qnot[0], Q[0], Clkwire, R, E, Qnot[0]);

   for(i = 1; i < size; i = i+1) begin: count

      D_FlipFlop U3(Qnot[i], Q[i],Qnot[i-1] , R, E, Qnot[i]);

   end
endgenerate

endmodule
```

Wrap around Up/Down Counter code is modified since we can only count up.

Top level:



```
 3    module top(data, clk, R, L, en, Q, ones);
 4
 5    input [7:0]data;
 6    input clk, R, L, en;
 7    output [7:0]Q;
 8    output [3:0]ones;
 9    wire Rnot;
10
11    not(Rnot,R);
12
13    parallel_load P1(data, clk, Rnot, Q, L);
14    up_counter U1(Q[0], L, clk, ones);
15
16    endmodule
17
18
```

Line 13: parallel_load module is entered.

Line 14: up_counter module is entered.

2. Modify your code to connect a 4-bit output to 7-segment decoder designed and used in LAB 2&3.

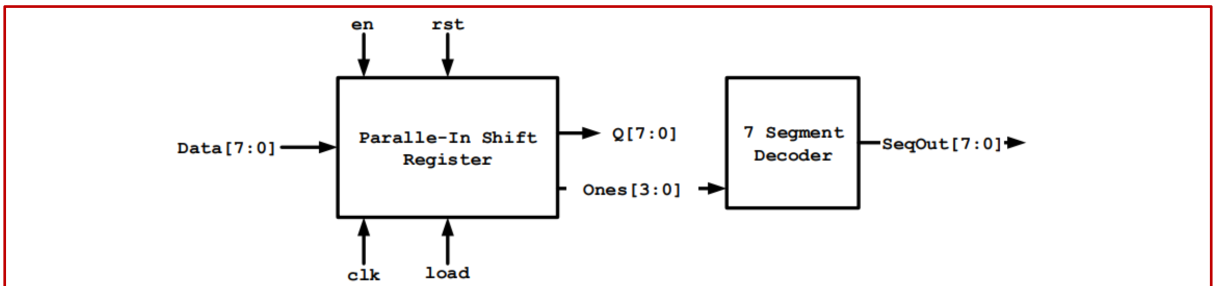
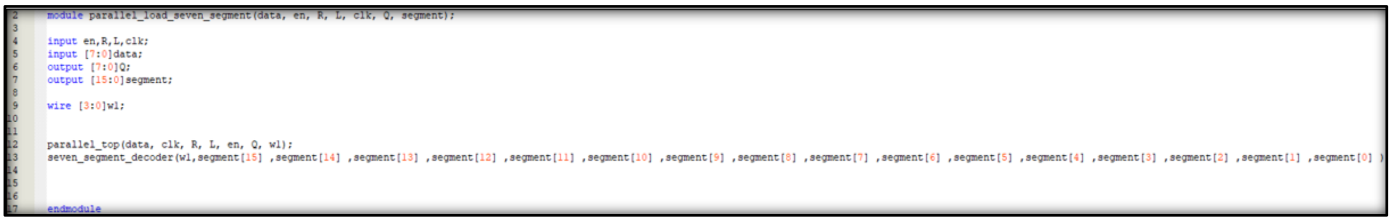


Figure 6: The Block Diagram for The Modified Circuit using 7-Segment Decoder.


```
module parallel_load_seven_segment(data, en, R, L, clk, Q, segment);

input en,R,L,clk;
input [7:0]data;
output [7:0]Q;
output [15:0]segment;

wire [3:0]wl;


parallel_top(data, clk, R, L, en, Q, wl);
seven_segment_decoder(wl,segment[15] ,segment[14] ,segment[13] ,segment[12] ,segment[11] ,segment[10] ,segment[9] ,segment[8] ,segment[7] ,segment[6] ,segment[5] ,segment[4] ,segment[3] ,segment[2] ,segment[1] ,segment[0] )



endmodule
```

Seven segment decoder and the top level code of parallel shift register with counter is combined in another top level code.

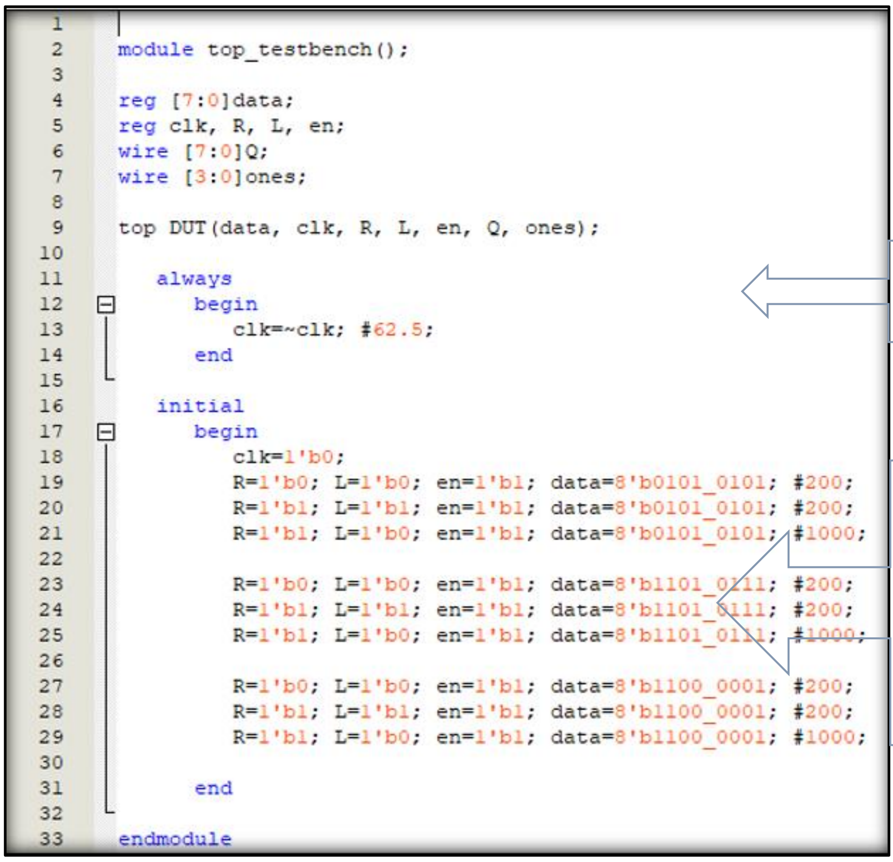
Decoder with n inputs would have 2^n outputs. We have 4 inputs and 16 outputs.

All the previous inputs and outputs are present here. However, "ones" output of parallel_top is fed into the 4-bit input of the seven segment decoder, output of the seven segment decoder is fed into the appropriate 16-bit output in order to drive the seven segment on the board.

3. Use the following test patterns with the clock periods provided in Table 3. Write a Verilog Testbench code to simulate and verify the functionality in ModelSim®.

Table 3: Testbench Stimuli

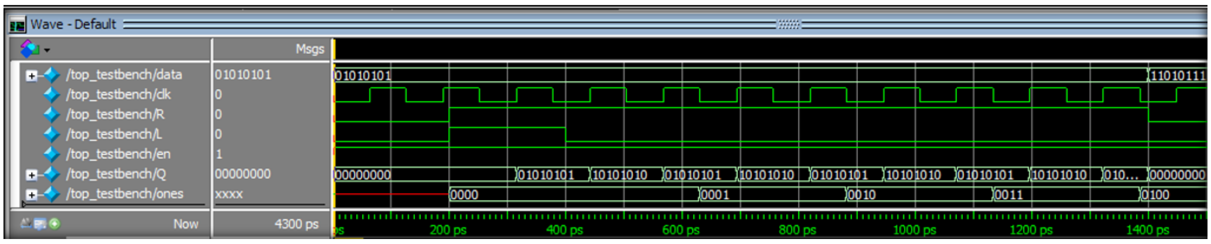
rst	load	en	Data	Delay
0	0	0	01010101	200 ns
1	1	1	01010101	200 ns
1	0	1	01010101	1000 ns
0	0	0	11010111	200 ns
1	1	1	11010111	200 ns
1	0	1	11010111	1000 ns
0	0	0	11000001	200 ns
1	1	1	11000001	200 ns
1	0	1	11000001	1000 ns



```

module top_testbench();

reg [7:0]data;
reg clk, R, L, en;
wire [7:0]Q;
wire [3:0]ones;

top DUT(data, clk, R, L, en, Q, ones);

   always
      begin
         clk=~clk; #62.5;
      end

   initial
      begin
         clk=1'b0;
         R=1'b0; L=1'b0; en=1'b1; data=8'b0101_0101; #200;
         R=1'b1; L=1'b1; en=1'b1; data=8'b0101_0101; #200;
         R=1'b1; L=1'b0; en=1'b1; data=8'b0101_0101; #1000;

         R=1'b0; L=1'b0; en=1'b1; data=8'b1101_0111; #200;
         R=1'b1; L=1'b1; en=1'b1; data=8'b1101_0111; #200;
         R=1'b1; L=1'b0; en=1'b1; data=8'b1101_0111; #1000;

         R=1'b0; L=1'b0; en=1'b1; data=8'b1100_0001; #200;
         R=1'b1; L=1'b1; en=1'b1; data=8'b1100_0001; #200;
         R=1'b1; L=1'b0; en=1'b1; data=8'b1100_0001; #1000;

      end

endmodule
```

Clk is changed every 62.5 pico seconds.

R, L, en, data changes 200 pico seconds 2 times then 1000 pico seconds one time. This is done three times as given in the table.

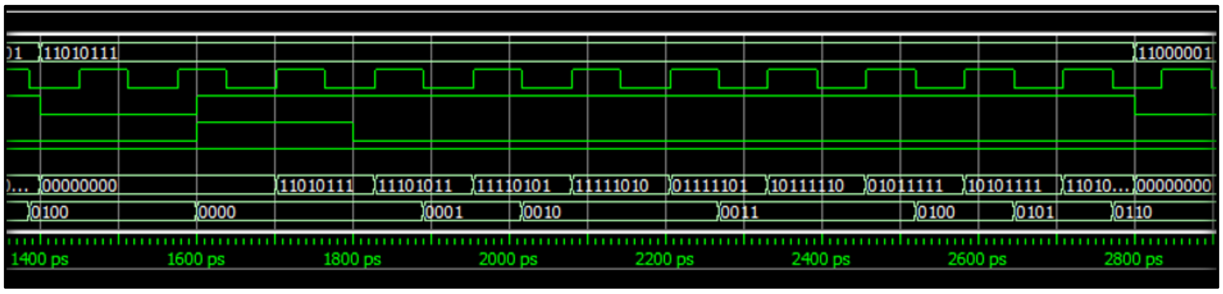


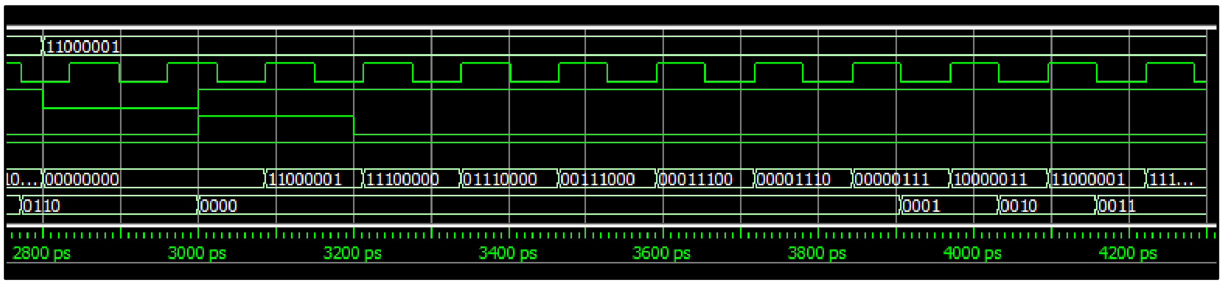
```
200-400ps: R = 1, L = 1. Since the load value is 1, rising
clock in this interval causes DFFs to take the data values.
Therefore Q=01010101.
```

```
400-1400ps: in this interval we see 8 rising clocks, which
means the values inside DFFs are shifter 8 times, and
meanwhile, according to what the least significant value of Q
is, counter count up at 1 values.
```

```
See that at the rising clock at 440ps, "ones" value stays at
0000. Because the least significant bit of the Q is 0 at that
point.
```

```
In contrast, at around 1400ps, just before the DFFs are
resetted, "ones" value becomes 0100 which is 4 and indeed
01010101 has 4 1's.
```





This exact behavior can be seen in 1400-2800ps and 2800-4400ps intervals.

see the "ones" value at 2800ps is 0110 which is 6, 11010111 has 6 ones.

Same with 4200ps, 0011 = 3, 11000001 has 3 ones.

3.3: Conclusion:

In conclusion, we tried to implement the flip flops best we could. However, it has been a challenge. Needing to learn chapter 6 before seeing it during the lectures has been an issue. A lot of compilation errors and inconsistencies between Modelsim and Quartus had prevailed. What compiled in Quartus didn’t in Modelsim. What simulated in Quartus didn’t open in Modelism. However even with all these challenges we understood memory storage and flip flops in a much deeper level. What we learned during chapter 5 helped us with creating tables and getting the formulas we needed. While simulating we understood how the clock worked exactly, when the module should load or reset. As much as this lab was difficult, it was also a great learning experience.