ELP201 Report

Uddhav Goel, 2019EE10540

May 2021

1 Synchronous 4-bit Gray-Code Counter

First, we start off by making the state transition table for our 4-bit gray-code counter, that covers all 16 states cyclically.

1.1 State Transition Table

Q_3	Q_2	Q_1	Q_0	Q_3^N	Q_2^N	Q_1^N	Q_0^N
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	1
0	0	1	1	0	0	1	0
0	0	1	0	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	0	1	0	1
0	1	0	1	0	1	0	0
0	1	0	0	1	1	0	0
1	1	0	0	1	1	0	1
1	1	0	1	1	1	1	1
1	1	1	1	1	1	1	0
1	1	1	0	1	0	1	0
1	0	1	0	1	0	1	1
1	0	1	1	1	0	0	1
1	0	0	1	1	0	0	0
1	0	0	0	0	0	0	0

Table 1: State transition table

1.2 Required Flip-Flops

Since we are building a 4-bit gray counter which has a total of 16 states
$\implies$ Minimum number of Flip-Flops required = $\lceil \log_2 16 \rceil = 4$

Since we need an SR-flip flop to represent each bit and its transitions, we need **4 SR- Flip flops** for our 4-bit gray code counter.

1.3 Values of SR Flip-Flops

According to the state transition table shown in section 1.1, we get the following values that need to be assigned to the 4 SR-FFs, in order to implement the gray code counter.

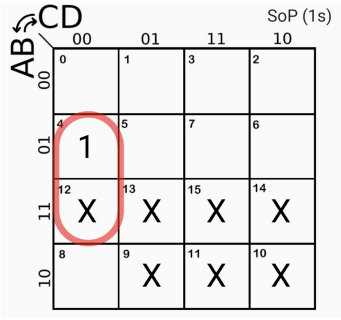
S_3	R_3	S_2	R_2	S_1	R_1	S_0	R_0
0	x	0	x	0	x	1	0
0	x	0	x	1	0	x	0
0	x	0	x	x	0	0	1
0	x	1	0	x	0	0	x
0	x	x	0	x	0	1	0
0	x	x	0	0	1	x	0
0	x	x	0	0	x	0	1
1	0	x	0	0	x	0	x
x	0	x	0	0	x	1	0
x	0	x	0	1	0	x	0
x	0	x	0	x	0	0	1
x	0	0	1	x	0	0	x
x	0	0	x	x	0	1	0
x	0	0	x	0	1	x	0
x	0	0	x	0	x	0	1
0	1	0	x	0	x	0	x

Table 2: Values of S,R Flip-Flop inputs

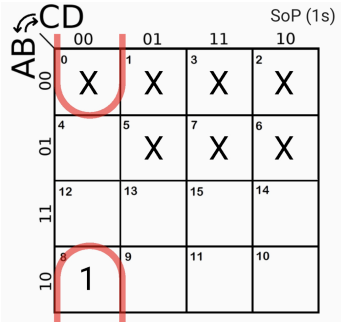
1.4 Simplified expressions for inputs using K-maps

Using K-maps, we get simplified expressions for S_i and R_i $\forall i \in \{1,2,3,4\}$ from the values obtained in Table 2, in terms of the inputs Q_i from Table 1.

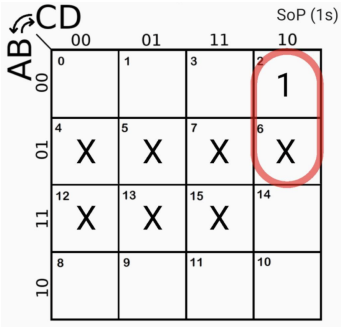
NOTE : In the K-maps below, $A = Q_3$, $B = Q_2$, $C = Q_1$ and $D = Q_0$.



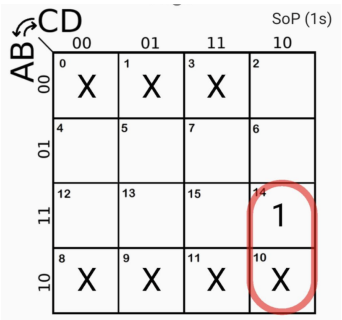
(a) K-map for S_3



(b) K-map for R_3



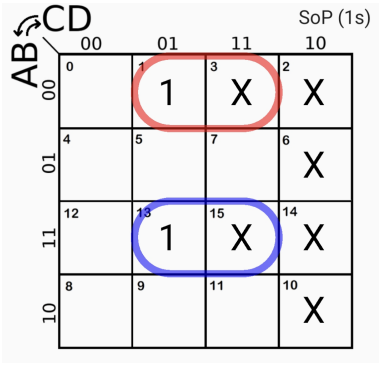
(c) K-map for S_2



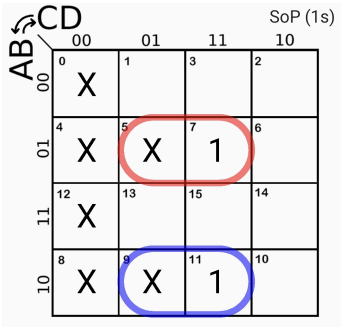
(d) K-map for R_2

The simplified expressions obtained from the K-maps drawn above are as follows:

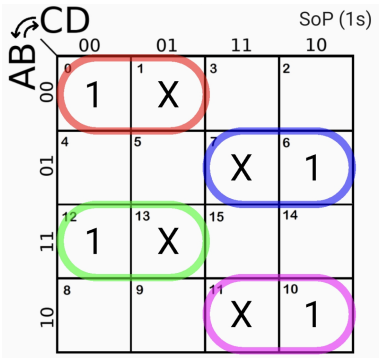
- $S_3 = Q_2\bar{Q}_1\bar{Q}_0$
- $R_3 = \bar{Q}_2\bar{Q}_1\bar{Q}_0$
- $S_2 = \bar{Q}_3Q_1\bar{Q}_0$
- $R_2 = Q_3Q_1\bar{Q}_0$



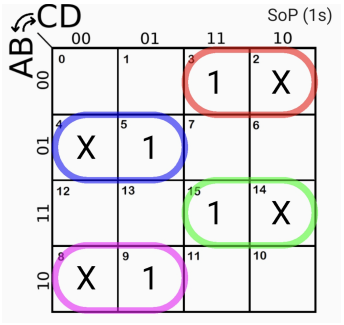
(a) K-map for S_1



(b) K-map for R_1



(c) K-map for S_0



(d) K-map for R_0

The simplified expressions obtained from the K-maps drawn above are as follows:

- $S_1 = Q_0\,(\overline{Q_3 \oplus Q_2})$
- $R_1 = Q_0\,(Q_3 \oplus Q_2)$
- $S_0 = \overline{Q_3 \oplus Q_2 \oplus Q_1}$
- $R_0 = Q_3 \oplus Q_2 \oplus Q_1$

1.5 Verilog Code

After getting the minimised expressions for all the inputs to SR-FFs, we write the following verilog code defining a module for gray code counter.

```
1 module srff ( q, qbar, qcur, s, r, clk, reset);
2     input s, r, clk, qcur, reset;
3     output reg q, qbar;
4     always @(reset == 1) begin  // resets the output to 0
5         q = 0; qbar = 1;
6     end
```

```
7       always @(posedge clk) begin //positive edge triggered clock
8           if(s == 1 & r == 0) begin
9               q = 1; qbar = 0;
10          end
11          else if (s == 0 & r == 1) begin
12              q = 0; qbar = 1;
13          end
14          else if (s == 0 & r == 0) begin
15              q = qcur; qbar = ~qcur;
16          end
17      end
18  endmodule
19
20  module grayctr (
21      input [3 : 0] q,   //Takes in the 4-bit current state
22      input clk,
23      input reset,
24      output [3 : 0] qn   //Counter outputs the next state
25  );
26      wire qdash[0 : 3], s[0 : 3], r[0 : 3];
27      //Now we assign the inputs to SR-FFs in terms of the input
            q.
28      assign s[3] = q[2] & (~q[1]) & (~q[0]);
29      assign r[3] = (~q[2]) & (~q[1]) & (~q[0]);
30      assign s[2] = (~q[3]) & q[1] & (~q[0]);
31      assign r[2] = q[3] & q[1] & (~q[0]);
32      assign s[1] = q[0] & (~(q[2] ^ q[3]));
33      assign r[1] = q[0] & (q[2] ^ q[3]);
34      assign s[0] = (~(q[3] ^ q[2] ^ q[1]));
35      assign r[0] = (q[3] ^ q[2] ^ q[1]);
36      //To implement the gray code counter, we call 4 SR-FF units
            as follows :
37      srff FF1(qn[3], qdash[3], q[3], s[3], r[3], clk, reset);
38      srff FF2(qn[2], qdash[2], q[2], s[2], r[2], clk, reset);
39      srff FF3(qn[1], qdash[1], q[1], s[1], r[1], clk, reset);
40      srff FF4(qn[0], qdash[0], q[0], s[0], r[0], clk, reset);
41  endmodule
```

Here is the code for the test bench to simulate the 4-bit gray code counter:

```
module tb_grayctr;
    reg [3 : 0] q;
    reg clk;
    reg reset;
    wire [3 : 0] qn;
    grayctr DUT( q, clk, reset, qn); // initialise the device
       under test
    always #5 clk = ~clk;             // and the clock

    initial begin
        $dumpfile("grayctr.vcd");
        $dumpvars(0, tb_grayctr);
        $monitor($time, " Current state = %b  Next state = %b",
            q, qn);
        clk <= 0;
        reset <=0;
        q <= 4'b0000;
        repeat(1) @(posedge clk) begin //reset the input and
           the output
            reset <=0;
        end
        repeat(31) @(posedge clk) begin //executes a full cycle
            of the counter
            q <= qn;
        end
        $finish;
    end
endmodule
```

1.6 Output states and Waveform

Here is the output cycle of states generated using the test-bench:

```
Uddhavs-MacBook-Pro:ELL201 uddhavgoel$ iverilog -o grayctrsim grayctr.v tb_grayctr.v
Uddhavs-MacBook-Pro:ELL201 uddhavgoel$ vvp grayctrsim
VCD info: dumpfile grayctr.vcd opened for output.
                   0 Current state = 0000  Next state = 0000
                   5 Current state = 0000  Next state = 0001
                  15 Current state = 0001  Next state = 0001
                  25 Current state = 0001  Next state = 0011
                  35 Current state = 0011  Next state = 0011
                  45 Current state = 0011  Next state = 0010
                  55 Current state = 0010  Next state = 0010
                  65 Current state = 0010  Next state = 0110
                  75 Current state = 0110  Next state = 0110
                  85 Current state = 0110  Next state = 0111
                  95 Current state = 0111  Next state = 0111
                 105 Current state = 0111  Next state = 0101
                 115 Current state = 0101  Next state = 0101
                 125 Current state = 0101  Next state = 0100
                 135 Current state = 0100  Next state = 0100
                 145 Current state = 0100  Next state = 1100
                 155 Current state = 1100  Next state = 1100
                 165 Current state = 1100  Next state = 1101
                 175 Current state = 1101  Next state = 1101
                 185 Current state = 1101  Next state = 1111
                 195 Current state = 1111  Next state = 1111
                 205 Current state = 1111  Next state = 1110
                 215 Current state = 1110  Next state = 1110
                 225 Current state = 1110  Next state = 1010
                 235 Current state = 1010  Next state = 1010
                 245 Current state = 1010  Next state = 1011
                 255 Current state = 1011  Next state = 1011
                 265 Current state = 1011  Next state = 1001
                 275 Current state = 1001  Next state = 1001
                 285 Current state = 1001  Next state = 1000
                 295 Current state = 1000  Next state = 1000
                 305 Current state = 1000  Next state = 0000
                 315 Current state = 0000  Next state = 0000
```

Figure 3: Outputs generated by the test bench

The figure below shows the waveform generated by the simulation of our 4-bit gray code counter :

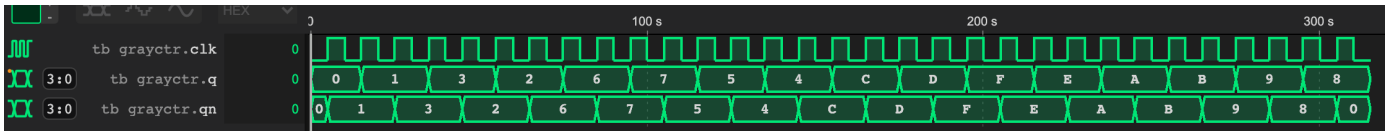


Figure 4: Wave forms generated by the test bench

2 Synchronous Ring Counter

2.1 State Transition Table

Here is the state transition table for the synchronous ring counter given in the question:

Q3	Q2	Q1	Q0	D3	D2	D1	D0
0	0	0	1	1	0	0	0
1	0	0	0	0	1	0	0
0	1	0	0	0	0	1	0
0	0	1	0	1	0	0	1
1	0	0	1	1	1	0	0
1	1	0	0	0	1	1	0
0	1	1	0	1	0	1	1
1	0	1	1	0	1	0	1
0	1	0	1	1	0	1	0
1	0	1	0	1	1	0	1
1	1	0	1	1	1	1	0
1	1	1	0	1	1	1	1
1	1	1	1	0	1	1	1
0	1	1	1	0	0	1	1
0	0	1	1	0	0	0	1

Figure 5: State Table for Synchronous Ring Counter

2.2 Required Flip-Flops

From the state table we can clearly see that our ring counter traverses 15 states
$\implies$ Minimum number of Flip-Flops required = $\lceil \log_2 15 \rceil = 4$

Since we need a D-flip flop to represent each bit and its transitions, we need **4 D-Flip flops** for our synchronous ring counter.

The values of the inputs to these D-Flip flops has been mentioned in the state table attached above.

2.3 Purpose of the counter

As we can clearly see from the state table, the counter doesn't cover all 16 states. **State 0000 is a forbidden state.**

The counter has 15 states, which all recur cyclically. Hence the purpose of the ring counter is to act like a **mod 15 counter**. As we can't see a set order in which the states appear, it can also be used as a **pseudo random sequence generator**, which can be used in security of data transmission.

Since we are going in a cycle, **the starting position doesn't matter**. All it does is shift the cycle a little bit.

2.4 Minimised expressions for inputs to D-Flip flops

Since the ring counter behaves like a shift register, the inputs to all the FFs, except the first one, is the output of of the previous flip flop. Therefore, we get the minimised expressions for D_3, D_2, D_1 and D_0. Here, I solve the K-map for D_3 only, as it can not be directly deduced.

2.4.1 $D_2 = Q_3$

2.4.2 $D_1 = Q_2$

2.4.3 $D_0 = Q_1$

2.4.4 $D_3 = Q_0 \oplus Q_1$

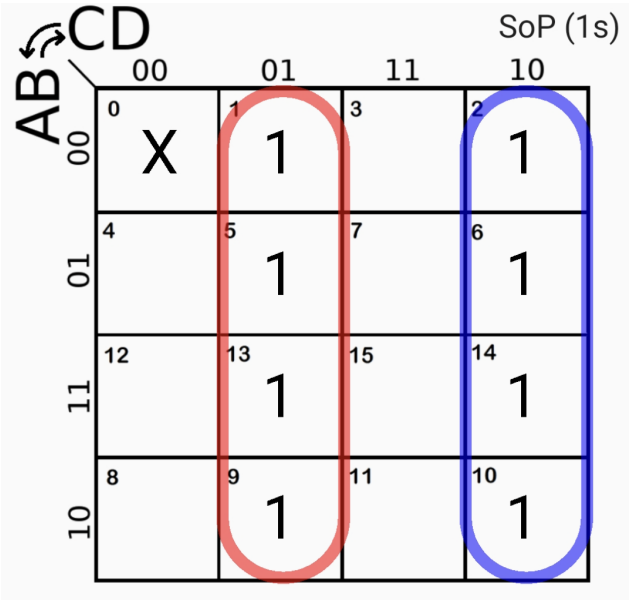


Figure 6: K-map for D_3

NOTE: Here, $A = Q_3$, $B = Q_2$, $C = Q_1$ and $D = Q_0$.

2.5 Verilog code

After getting the minimised expressions for all the inputs to D-FFs, we write the following verilog code defining a module for our synchronous ring counter.

```
module dff ( d, clk, qn, qnbar, reset);
    input d, clk, reset;
    output reg qn, qnbar;
    always @(reset == 1) begin // resets the output to 0
        qn = 0; qnbar = 1;
    end
    always @(posedge clk) begin // positive edge triggered
       clock
        if(d == 0) begin
            qn = 0; qnbar = 1;
        end
        else begin
            qn = 1; qnbar = 0;
        end
    end
endmodule

module ringctr (
    input [3 : 0] q, // Takes in a 4-bit input
    input clk,
    input reset,
    output [3 : 0] qn // Returns the next state (4-bit)
);
    wire [3 : 0] qnbar;
    //Here, we assign the inputs to D-FFs in terms of the
    //input q, and call 4 units of D-FFs as follows :
    dff #1 D1( q[0]^q[1], clk, qn[3], qnbar[3], reset);
    dff #1 D2( q[3], clk, qn[2], qnbar[2], reset);
    dff #1 D3( q[2], clk, qn[1], qnbar[1], reset);
    dff #1 D4( q[1], clk, qn[0], qnbar[0], reset);
endmodule
```

Here is the code for the test bench to simulate our counter:

```
module tb_ringctr;
    reg [3 : 0] q;
    reg clk;
    reg reset;
    wire [3 : 0] qn;
    ringctr DUT( q, clk, reset, qn);
    //initialise the device under test
    //and the clock
    always #5 clk = ~clk;

    initial begin
        $dumpfile("ringctr.vcd");
        $dumpvars(0, tb_ringctr);
        $monitor($time, " Current state = %b  Next state = %b",
            q, qn);
        //print the multiple states as outputs on positive
           clock edges
        clk <= 0;
        reset <= 0;
        q <= 4'b0001;
        //initialise the clk, reset, and input q
        repeat(1) @(posedge clk) begin
            reset <= 1;
        end
        //execute a full cycle of the counter as follows:
        repeat(30) @(posedge clk) begin
            q <= qn;
        end
        $finish;
    end
endmodule
```

2.6 Output states and Waveform

Here is the output cycle of states generated using the test-bench:

```
Uddhavs-MacBook-Pro:ELL201 uddhavgoel$ iverilog -o ringctrsim ringctr.v tb_ringctr.v
Uddhavs-MacBook-Pro:ELL201 uddhavgoel$ vvp ringctrsim
VCD info: dumpfile ringctr.vcd opened for output.
                   0 Current state = 0001  Next state = 0000
                   5 Current state = 0001  Next state = 0000
                  15 Current state = 1000  Next state = 1000
                  25 Current state = 1000  Next state = 0100
                  35 Current state = 0100  Next state = 0100
                  45 Current state = 0100  Next state = 0010
                  55 Current state = 0010  Next state = 0010
                  65 Current state = 0010  Next state = 1001
                  75 Current state = 1001  Next state = 1001
                  85 Current state = 1001  Next state = 1100
                  95 Current state = 1100  Next state = 1100
                 105 Current state = 1100  Next state = 0110
                 115 Current state = 0110  Next state = 0110
                 125 Current state = 0110  Next state = 1011
                 135 Current state = 1011  Next state = 1011
                 145 Current state = 1011  Next state = 0101
                 155 Current state = 0101  Next state = 0101
                 165 Current state = 0101  Next state = 1010
                 175 Current state = 1010  Next state = 1010
                 185 Current state = 1010  Next state = 1101
                 195 Current state = 1101  Next state = 1101
                 205 Current state = 1101  Next state = 1110
                 215 Current state = 1110  Next state = 1110
                 225 Current state = 1110  Next state = 1111
                 235 Current state = 1111  Next state = 1111
                 245 Current state = 1111  Next state = 0111
                 255 Current state = 0111  Next state = 0111
                 265 Current state = 0111  Next state = 0011
                 275 Current state = 0011  Next state = 0011
                 285 Current state = 0011  Next state = 0001
                 295 Current state = 0001  Next state = 0001
                 305 Current state = 0001  Next state = 1000
```

Figure 7: Outputs generated by the test bench

The figure below shows the waveform generated by the simulation of our synchronous ring counter :

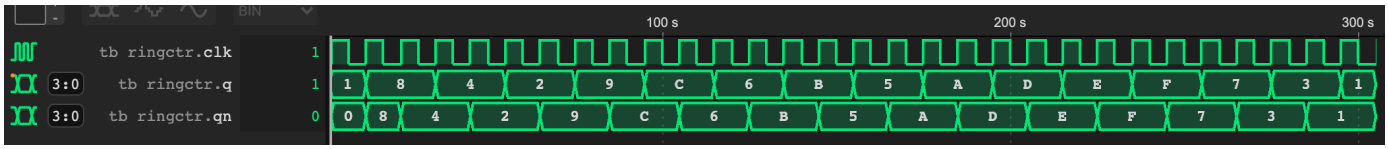


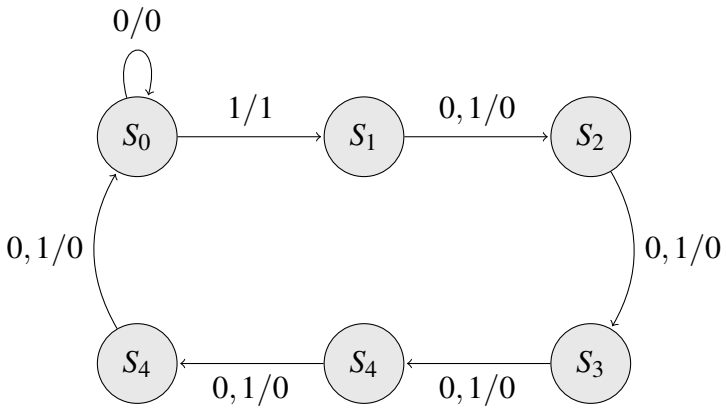
Figure 8: Wave forms generated by the test bench

3 Sequence Generator

Since my entry number is 2019EE10540, the output sequence that I need to generate is:

$$\{\text{3-bit binary of (4\%8), 3-bit binary of (0\%8)}\} = \{1,0,0,0,0,0\}$$

3.1 State Transition Diagram



Here, S_0 represents the idle state. The system starts in the idle state and on seeing the input 1, produces the output sequence Y, as mentioned above, irrespective of the input in the next 5 cycles.

3.2 State Transition Table

Now, we encode our states of the FSM in binary form and obtain the following transition table:

Q_2	Q_1	Q_0	X	Q_2^N	Q_1^N	Q_0^N	Y
0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	1
0	0	1	0	0	1	0	0
0	0	1	1	0	1	0	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	0
0	1	1	0	1	0	0	0
0	1	1	1	1	0	0	0
1	0	0	0	1	0	1	0
1	0	0	1	1	0	1	0
1	0	1	0	0	0	0	0
1	0	1	1	0	0	0	0

Table 3: State Transition Table for the Mealy machine given above

In the table given above, X represents the input, Y represents the output, $Q_2Q_1Q_0$ represents the current state in 3-bit binary form, and finally $Q_2^NQ_1^NQ_0^N$ represents the next state in 3-bit binary form.

NOTE: State S_0 maps to 000, S_1 maps to 001, and so on.

3.3 Flip-Flops required

The encoding of the states is done as follows:

- S_0 maps to 000
- S_1 maps to 001
- S_2 maps to 010
- S_3 maps to 011
- S_4 maps to 100
- S_5 maps to 101

Hence, we see that all the states can be encoded using 3 bits. Now, we also need 1 bit to represent the output, Y.

We need a total of **3 D Flip flops** to implement the given FSM, 1 for each bit to be encoded. The output Y can be implemented using simple combinational logic.

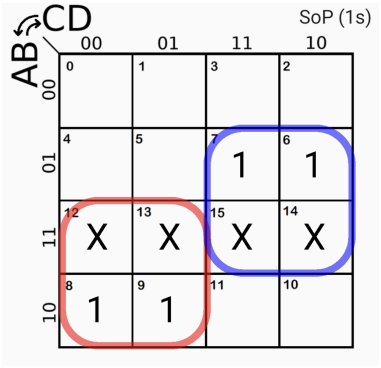
3.4 Minimised Expressions

Since we are using D-FFs, the bits used to represent the next state and the output are the same as the inputs to the D-FFs, i.e,

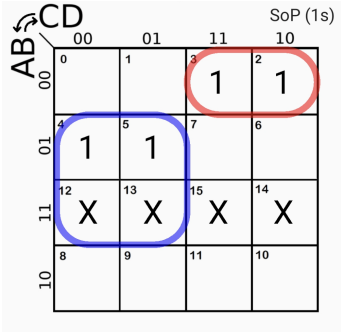
- $Q_2^N = D_2$
- $Q_1^N = D_1$
- $Q_1^N = D_0$

Using the K-maps shown below, we find the minimised expressions for D_2, D_1, D_0 and Y in terms of the current state variables: Q_2, Q_1, Q_0, and the input X.

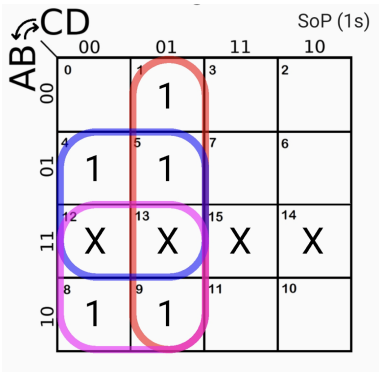
NOTE : In the K-maps below, $A = Q_2$, $B = Q_1$, $C = Q_0$ and $D = X$.



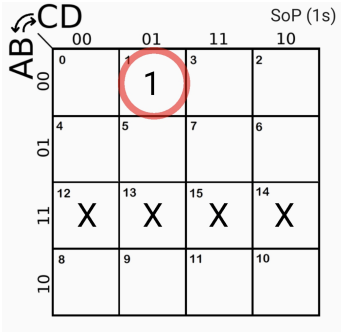
(a) K-map for D_2



(b) K-map for D_1



(c) K-map for D_0



(d) K-map for Y

The simplified expressions obtained from the K-maps drawn above are as follows:

- $D_2 = Q_2\bar{Q_0} + Q_1Q_0$
- $D_1 = Q_1\bar{Q_0} + \bar{Q_2}\bar{Q_1}Q_0$
- $D_0 = \bar{Q_0}\ (\ Q_2 + Q_1 + X\ )$
- $Y = \bar{Q_2}\bar{Q_1}\bar{Q_0}X$

3.5 Verilog code

After getting the minimised expressions for all the inputs to D-FFs, we write the following verilog code defining a module for our synchronous ring counter.

```
1 module dff ( d, clk, qn, qnbar, reset);
2     input d, clk, reset;
3     output reg qn, qnbar;
4     always @(reset == 1) begin // resets the output to 0
5         qn = 0; qnbar = 1;
6     end
```

```
7       always @(posedge clk) begin // positive edge triggered
            clock
8           if(d == 0) begin
9               qn = 0; qnbar = 1;
10          end
11          else begin
12              qn = 1; qnbar = 0;
13          end
14      end
15  endmodule
16
17  module fsm (
18      input [2 : 0] q, // Takes in a 3-bit current state
19      input x_in, // Takes the input
20      input clk,
21      input reset,
22      output [2 : 0] qn, // Returns the next state (3-bit)
23      output y_out // Returns the output
24  );
25      wire qnbar [2 : 0], dd [2 : 0];
26      //Here, we assign the inputs to D-FFs in terms of the
27      //input x and the current state q :
28      assign dd[2] = (q[2] & (~q[0])) | (q[1] & q[0]);
29      assign dd[1] = (q[1] & (~q[0])) | ((~q[2]) & (~q[1]) & q
            [0]);
30      assign dd[0] = (~q[0]) & (q[2] | q[1] | x_in);
31      assign y_out = (~q[2]) & (~q[1]) & (~q[0]) & x_in;
32      //We call the 3 D-FF units as following to implement the
            FSM:
33      dff #1 D1( dd[2], clk, qn[2], qnbar[2], reset);
34      dff #1 D2( dd[1], clk, qn[1], qnbar[1], reset);
35      dff #1 D3( dd[0], clk, qn[0], qnbar[0], reset);
36  endmodule
```

Here is the code for the test bench to simulate our counter:

```
module tb_fsm;
    reg [2 : 0] q;
    reg x_in;
    reg clk;
    reg reset;
    wire y_out;
    wire [2 : 0] qn;
    fsm DUT( q, x_in, clk, reset, qn, y_out);
    //initialise the device under test
    //and the clock
    always #5 clk = ~clk;
    always #40 x_in = ~x_in;

    initial begin
        $dumpfile("fsm.vcd");
        $dumpvars(0, tb_fsm);
        $monitor($time, " Current state = %b  Next state = %b",
            q, qn);
        //print the multiple states as outputs on positive
           clock edges
        clk <= 0;
        x_in <= 1;
        reset <= 0;
        q <= 3'b000;
        //initialise the clk, reset, and input q
        repeat(30) @(posedge clk) begin
            q <= qn;
        end
        $finish;
    end
endmodule
```

3.6 Output states and Waveform

Here is the output cycle of states generated using the test-bench:

```
Uddhavs-MacBook-Pro:ELL201 uddhavgoel$ iverilog -o fsmsim fsm.v tb_fsm.v
Uddhavs-MacBook-Pro:ELL201 uddhavgoel$ vvp fsmsim
VCD info: dumpfile fsm.vcd opened for output.
                   0 Current state = 000  Next state = 000
                   5 Current state = 000  Next state = 001
                  15 Current state = 001  Next state = 001
                  25 Current state = 001  Next state = 010
                  35 Current state = 010  Next state = 010
                  45 Current state = 010  Next state = 011
                  55 Current state = 011  Next state = 011
                  65 Current state = 011  Next state = 100
                  75 Current state = 100  Next state = 100
                  85 Current state = 100  Next state = 101
                  95 Current state = 101  Next state = 101
                 105 Current state = 101  Next state = 000
                 115 Current state = 000  Next state = 000
                 165 Current state = 000  Next state = 001
                 175 Current state = 001  Next state = 001
                 185 Current state = 001  Next state = 010
                 195 Current state = 010  Next state = 010
                 205 Current state = 010  Next state = 011
                 215 Current state = 011  Next state = 011
                 225 Current state = 011  Next state = 100
                 235 Current state = 100  Next state = 100
                 245 Current state = 100  Next state = 101
                 255 Current state = 101  Next state = 101
                 265 Current state = 101  Next state = 000
                 275 Current state = 000  Next state = 000
```

Figure 10: Outputs generated by the test bench

The figure below shows the waveform generated by the simulation of our Mealy Machine :

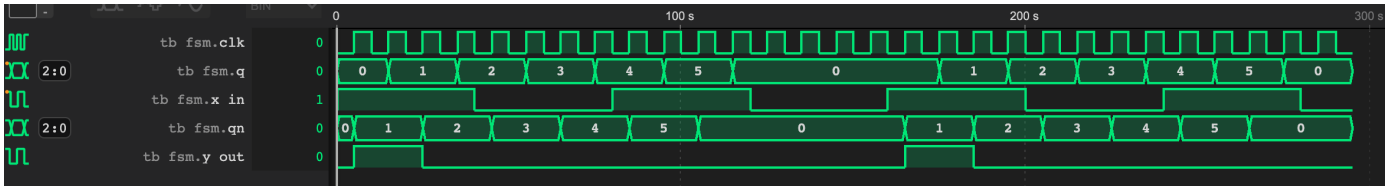


Figure 11: Wave forms generated by the test bench

Here we can see clearly, that when the FSM in the idle state, if the input is 0, then it stays in the idle state. If the input is 1, it starts generating the desired output sequence, irrespective of the input in the next 5 clock cycles.