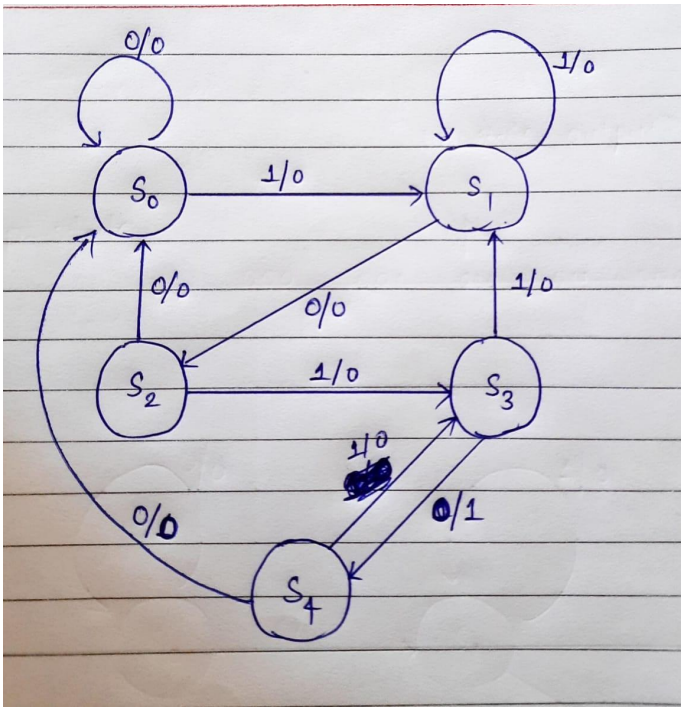
1010 SEQUENCE DETECTOR

- **State Diagram**



We have used four states for our FSM namely S_0, S_1, S_2, and S_3, where the binary encodings for the states are as follows:
S_0 = 000
S_1 = 001
S_2 = 010
S_3 = 011
S_4 = 100

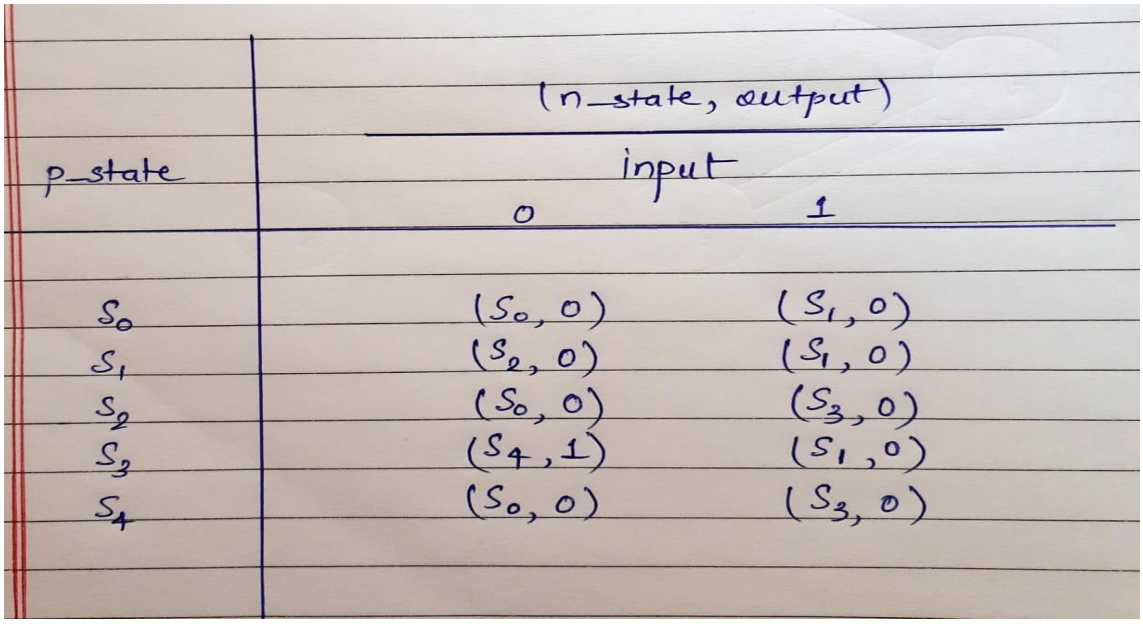
- **Approach:-**

Initially, the state is equal to S_0. At this stage, if the next input is 1 then we move to next state, i.e. S_1, else we remain in S_0 as long as the input = 0.
In S_1, if we get input = 0, we move to the next state i.e. S_2 and the current sequence is 10. Otherwise, we stay in S_1.
In S_2, if we get the input = 1, we move to the next state i.e. S_3, else, we move to S_0 since the sequence becomes 100 in this case which is not the desired subsequence for the sequence we want(1010). Hence, we start over in the latter case.
In S_3, if we get input = 1, we move to S1, since the sequence is distorted again(1011). If we get input=0, we get the desired sequence(1010), and we move to S_4.
In S_4, if we get 0, we move to S_0 and start over. Else, if we get input = 1, we move to S_3 since we are considering overlapping as valid in this problem.

- **Implementation details/ Assumptions**

As mentioned in the question, we have provided 15 different 1-bit inputs at a time delay of 5 units. The output is equal to 1 whenever we encounter 1,0,1,0 in successive input values. In all other cases, the output is 0. We have also taken care of the overlapping condition and the sequence used in our test bench also tests the same.
When reset is true, our state = S_0 and hence, the sequence 1010 encountered with reset = 1 doesn't produce output = 1.

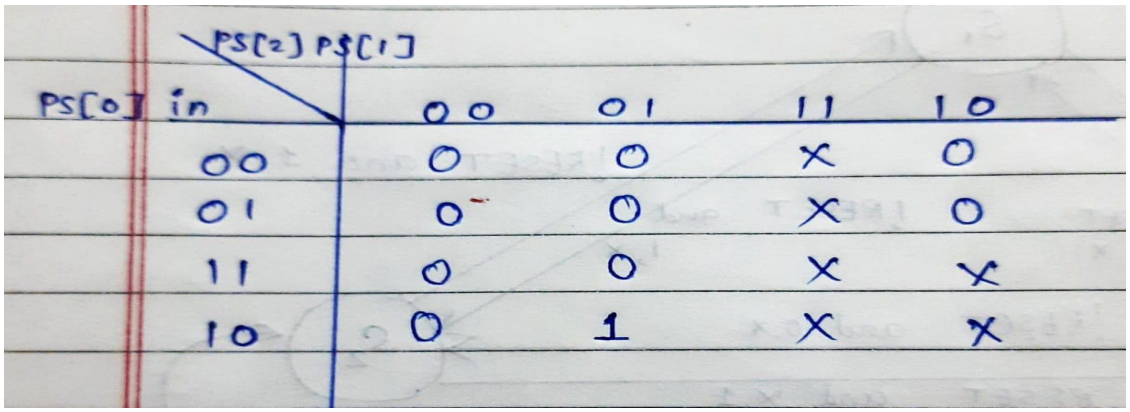
- **Transition and Output Table**



p_state	(n_state, output) input 0	input 1
S_0	$(S_0, 0)$	$(S_1, 0)$
S_1	$(S_2, 0)$	$(S_1, 0)$
S_2	$(S_0, 0)$	$(S_3, 0)$
S_3	$(S_4, 1)$	$(S_1, 0)$
S_4	$(S_0, 0)$	$(S_3, 0)$

Where, p_state represents the present state and n_state represents the next state.

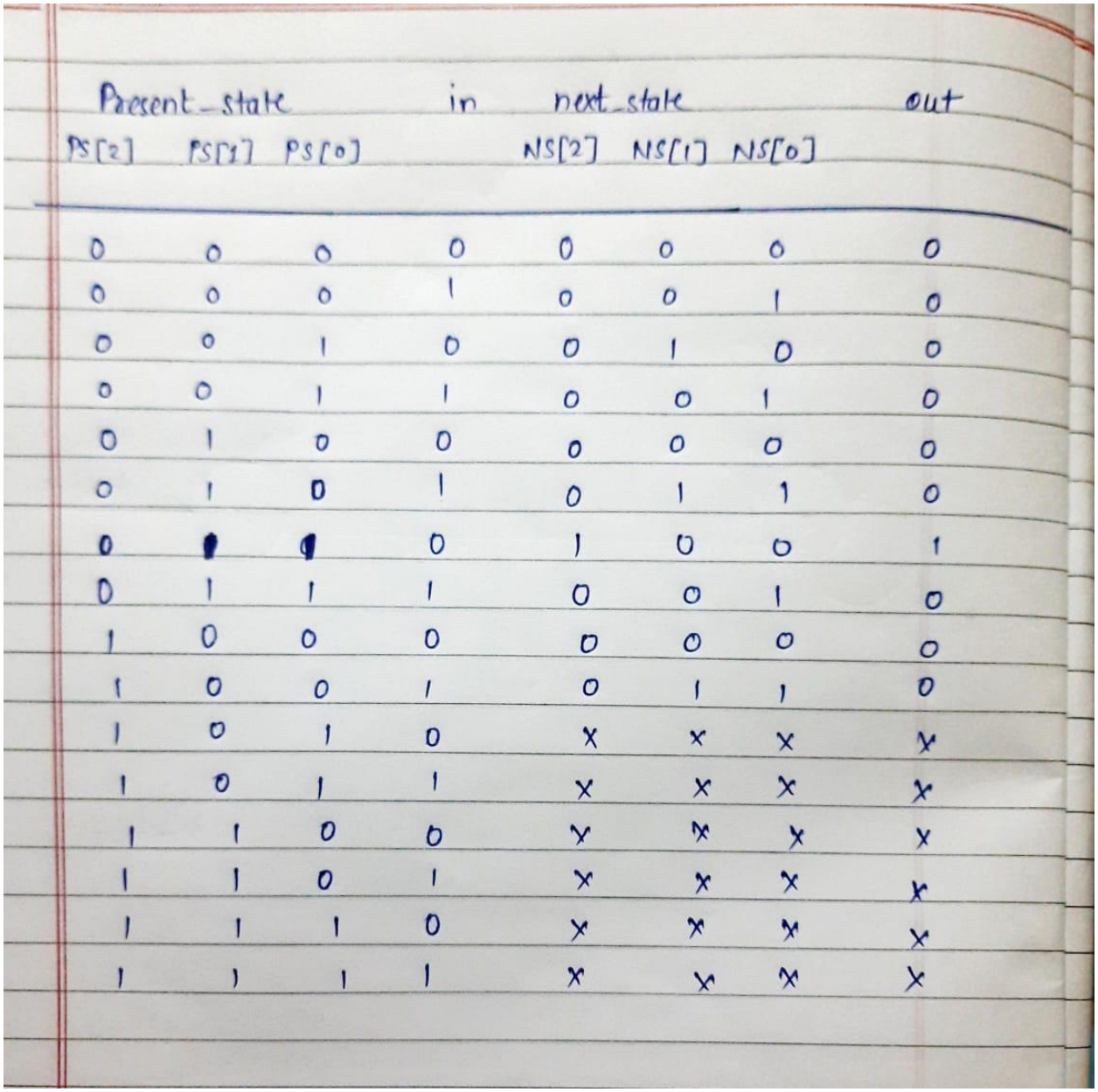
- **K-maps**



PS[0] in \ PS[2] PS[1]	00	01	11	10
00	0	0	X	0
01	0	0	X	0
11	0	0	X	X
10	0	1	X	X

Where,"in" is the input bit; PS[2] PS[1] PS[2] denotes the bits of the present state and NS[2] NS[1] NS[2] denotes the bits of the next state.
The values for states used correspond to the encodings defined earlier.

- **Excitation Table**

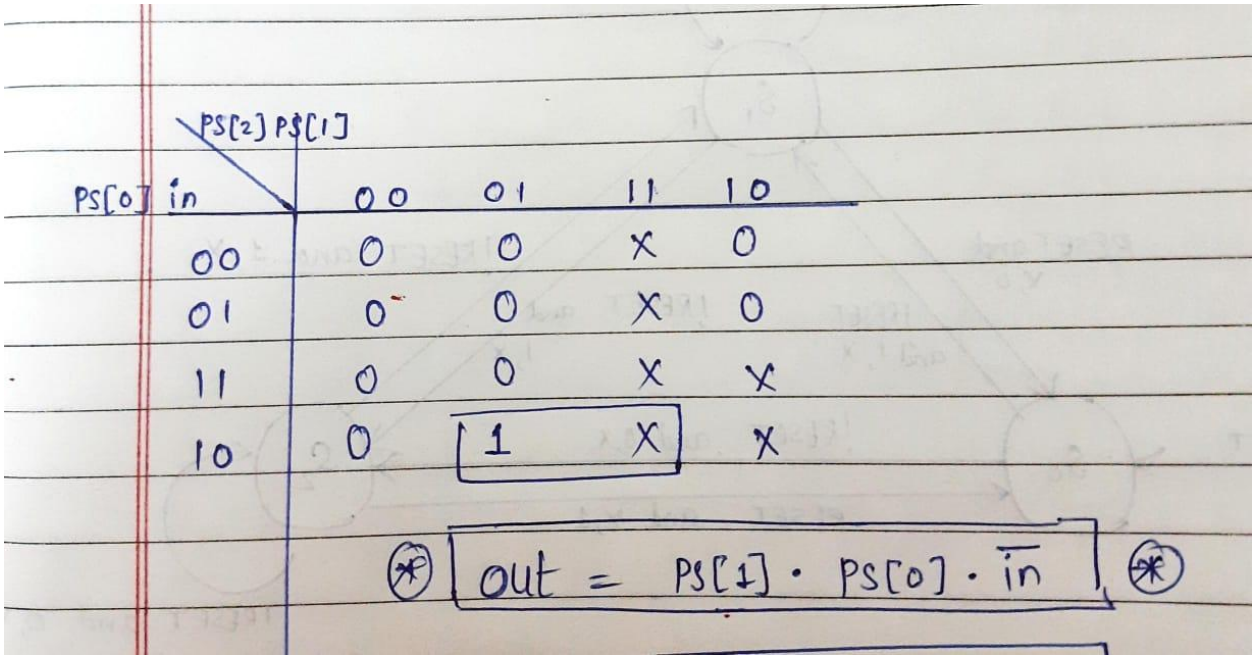


Present_state			in	next_state			out
PS[2]	PS[1]	PS[0]		NS[2]	NS[1]	NS[0]	
0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	0
0	0	1	0	0	1	0	0
0	0	1	1	0	0	1	0
0	1	0	0	0	0	0	0
0	1	0	1	0	1	1	0
0	1	1	0	1	0	0	1
0	1	1	1	0	0	1	0
1	0	0	0	0	0	0	0
1	0	0	1	0	1	1	0
1	0	1	0	x	x	x	x
1	0	1	1	x	x	x	x
1	1	0	0	x	x	x	x
1	1	0	1	x	x	x	x
1	1	1	0	x	x	x	x
1	1	1	1	x	x	x	x

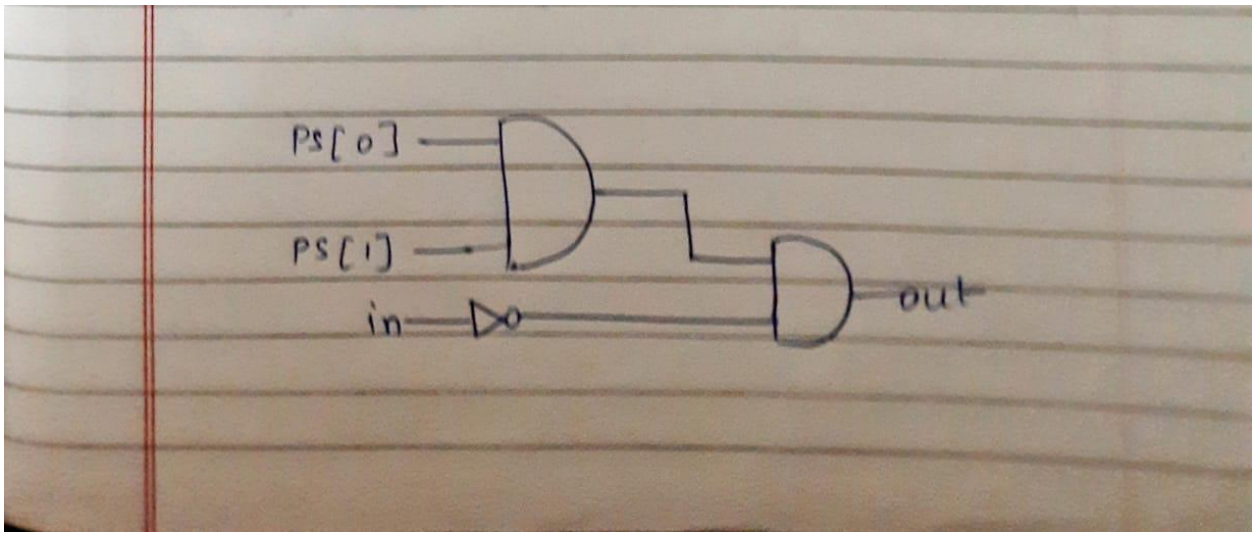
Where,"in" is the input bit and "out" is the output; PS[2] PS[1] PS[2] denotes the bits of the present state and NS[2] NS[1] NS[2] denotes the bits of the next state.
The values for states used correspond to the encodings defined earlier.

- **Logic and Circuit**

The following logic can be obtained from the k-map:



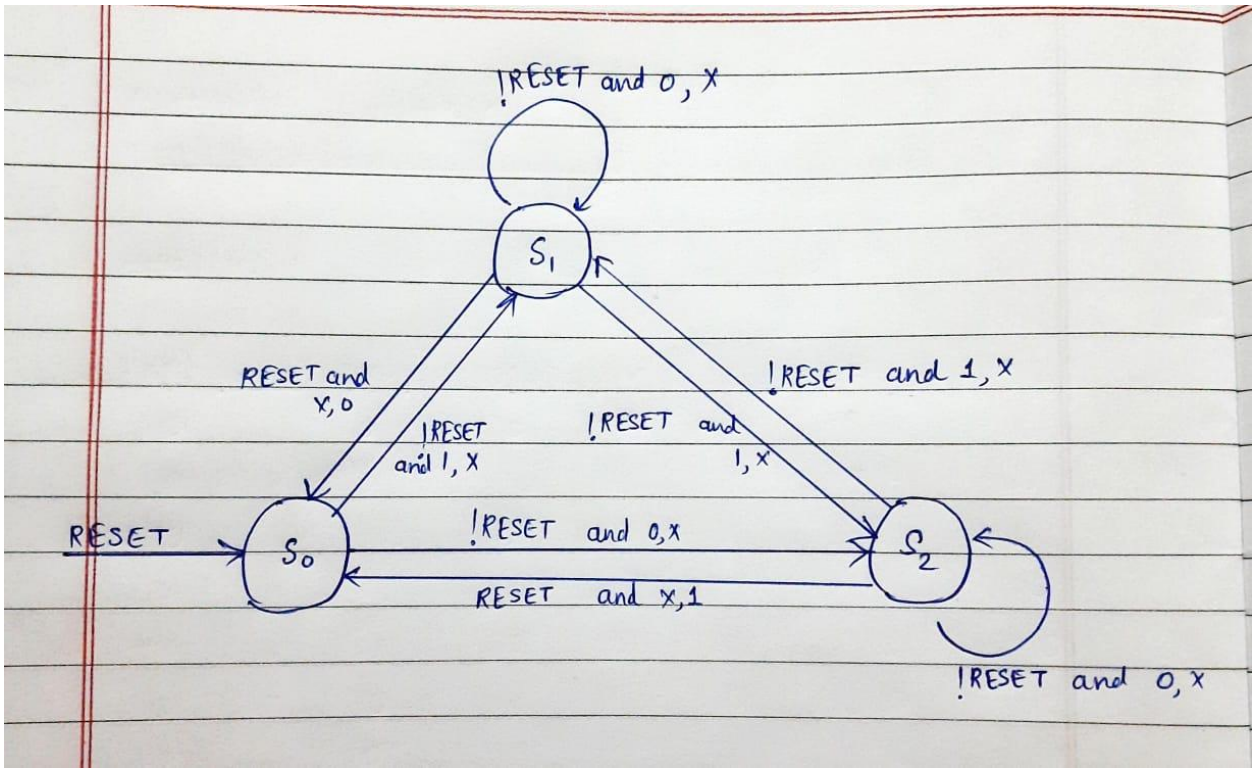
The logic in the above image can be implemented as following:



Where,"in" is the input bit and "out" is the output; PS[2] PS[1] PS[2] denotes the bits of the present state and NS[2] NS[1] NS[2] denotes the bits of the next state.
The values for states used correspond to the encodings defined earlier.

3-BIT ODD PARITY GENERATOR

- **State diagram**



We have 3 states for our FSM namely, S_0, S_1, and S_2
The state S_0 corresponds to the state when we try to reset the FSM.
S_1 corresponds to the state when the number of 1's so far in the string is odd.
S_2 corresponds to the state when the number of 1's so far in the string is even.

- **Approach**

  In the state S_0, we go to states S_1 and S_2 for input = 1 and input = 0 respectively (given that reset signal is low)

  Suppose we have a binary number at state S2, i.e. it is of odd parity;
  If we insert the next bit "0" to it, the binary number remains at state S2.
  And if we insert the next bit "1" to it, the binary number goes to state S1.
  If we have read the complete string, then reset gets high and then we move to S_0 instead.

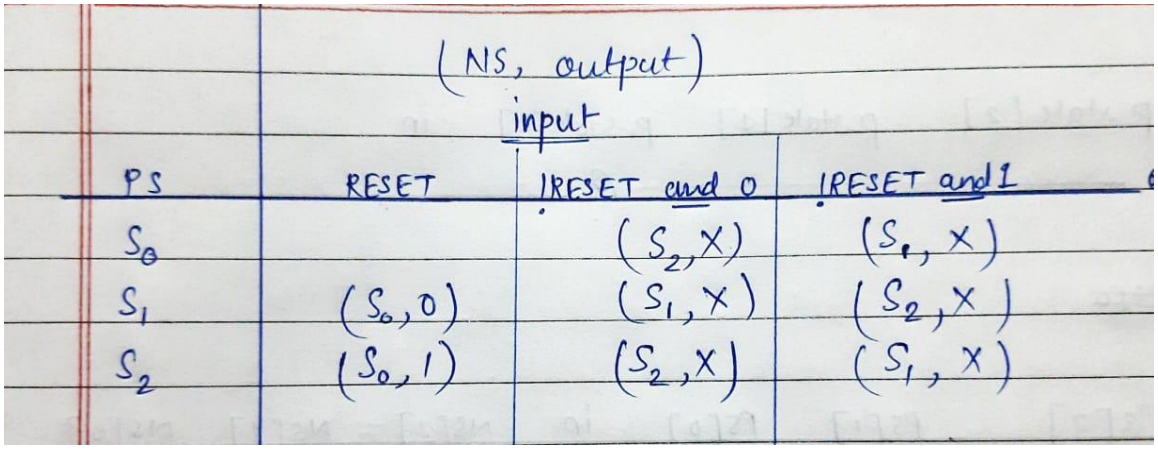
  Suppose we have a binary number at state S1, i.e. it is of even parity;

If we insert the next bit “0” to it, the binary number remains at state S1.
And if we insert the next bit “1” to it, the binary number goes to state S0.

For the odd parity generator, we want the number to be at S2 state when the 3-bit string has been read completely.
If we have read the complete string, then reset gets high and then we move to S_0 instead.

- **Implementation details/ Assumptions:**
  The input is in the form of consecutive 1-bit inputs. Each 3-bit string is represented by 3 consecutive inputs. For example, if the input is 000001010011, then the strings to be considered for parity bits are : 000, 001, 010, 011.
  The output is 1/0 when a complete 3-bit string has been read. At all other times, the output is x(don’t care).

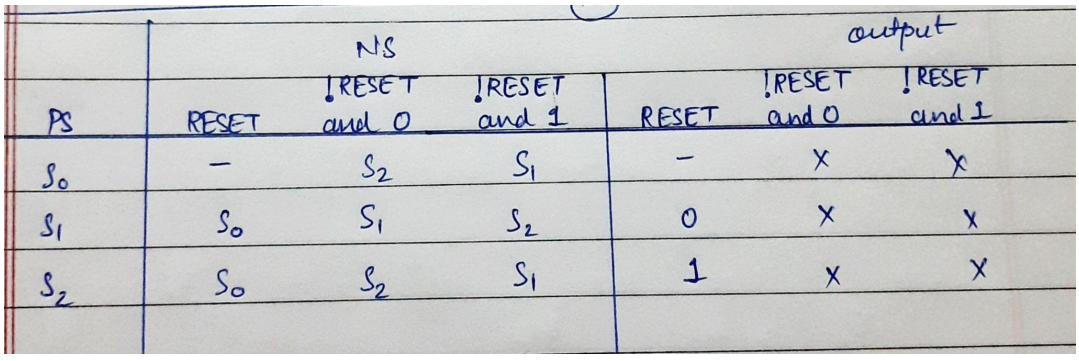
- **State Table**



	(NS, output)		
	input		
PS	RESET	!RESET and 0	!RESET and 1
S_0		(S_2, X)	(S_1, X)
S_1	$(S_0, 0)$	(S_1, X)	(S_2, X)
S_2	$(S_0, 1)$	(S_2, X)	(S_1, X)

Where, PS represents the present state and NS represents the next state.
In the above table, the “RESET” column doesn’t show the inputs given since the final state and output will not depend on the input in this case.

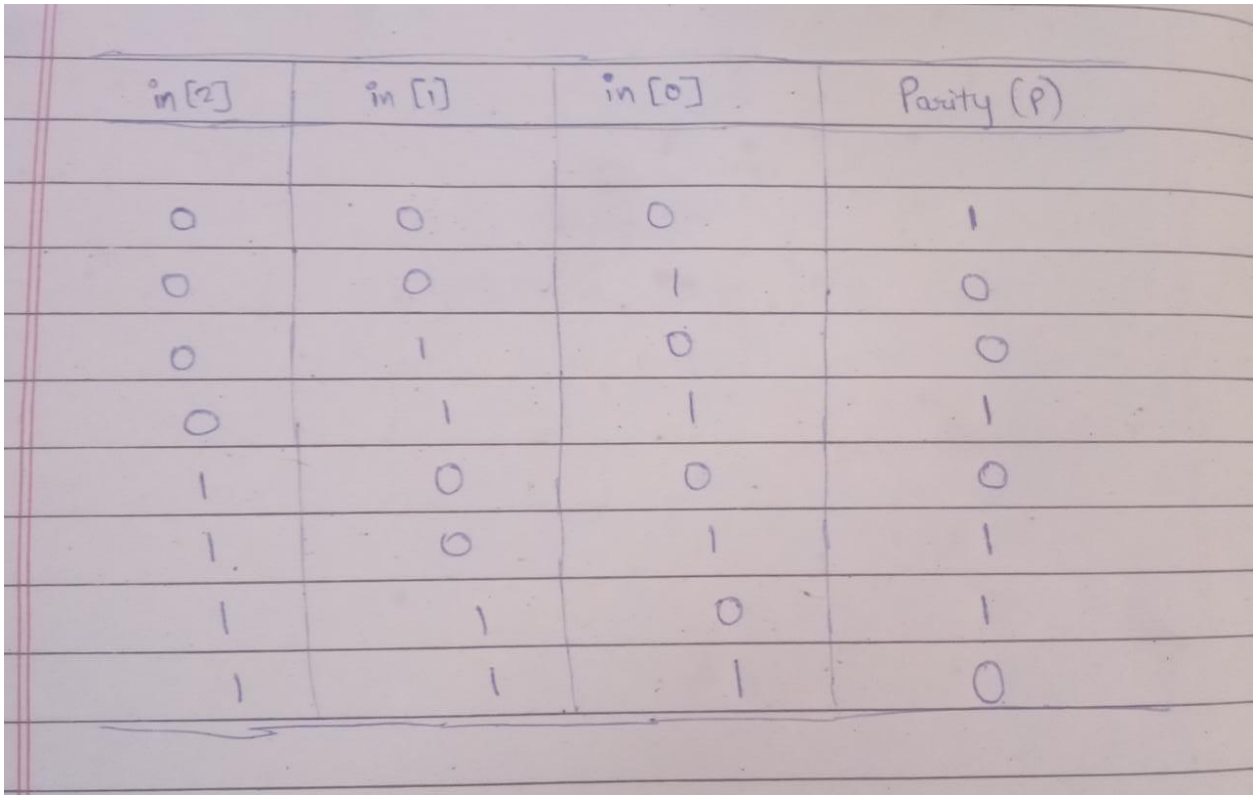
- **Transition and output table**



	NS		output		
PS	RESET	!RESET and 0	!RESET and 1	RESET	!RESET and 0	!RESET and 1
S_0	–	S_2	S_1	–	X	X
S_1	S_0	S_1	S_2	0	X	X
S_2	S_0	S_2	S_1	1	X	X

Where,PS represents the present state and NS represents the next state.

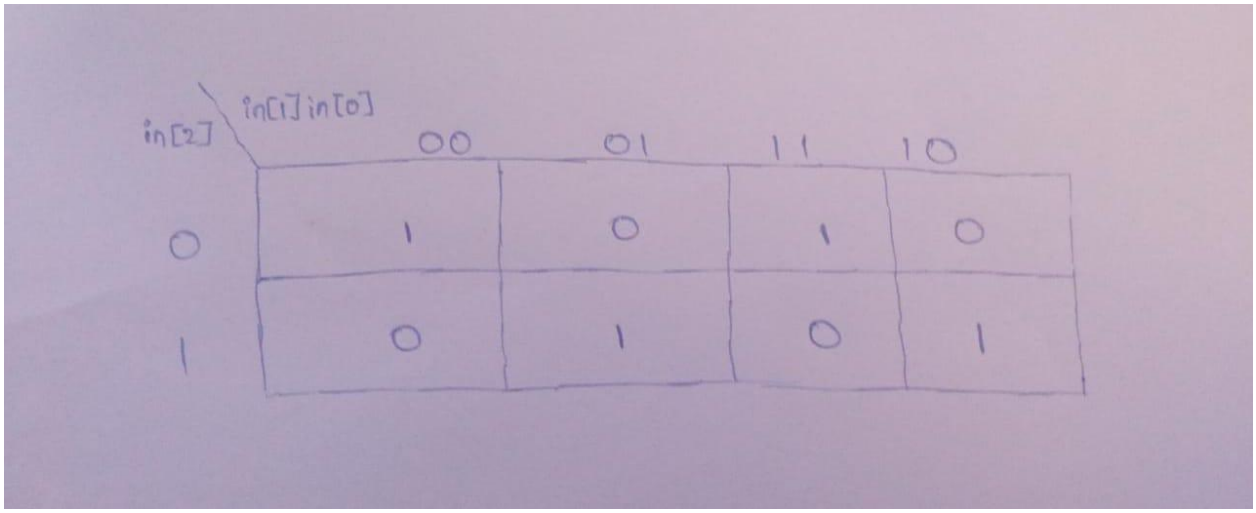
- **Excitation table**



in[2]	in[1]	in[0]	Parity (P)
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

In the above table, ‘in’ represents the 3-bit string which is to be read through 3 consecutive 1-bit inputs. The right-most column represents the output (the parity bit).

- **K-maps**

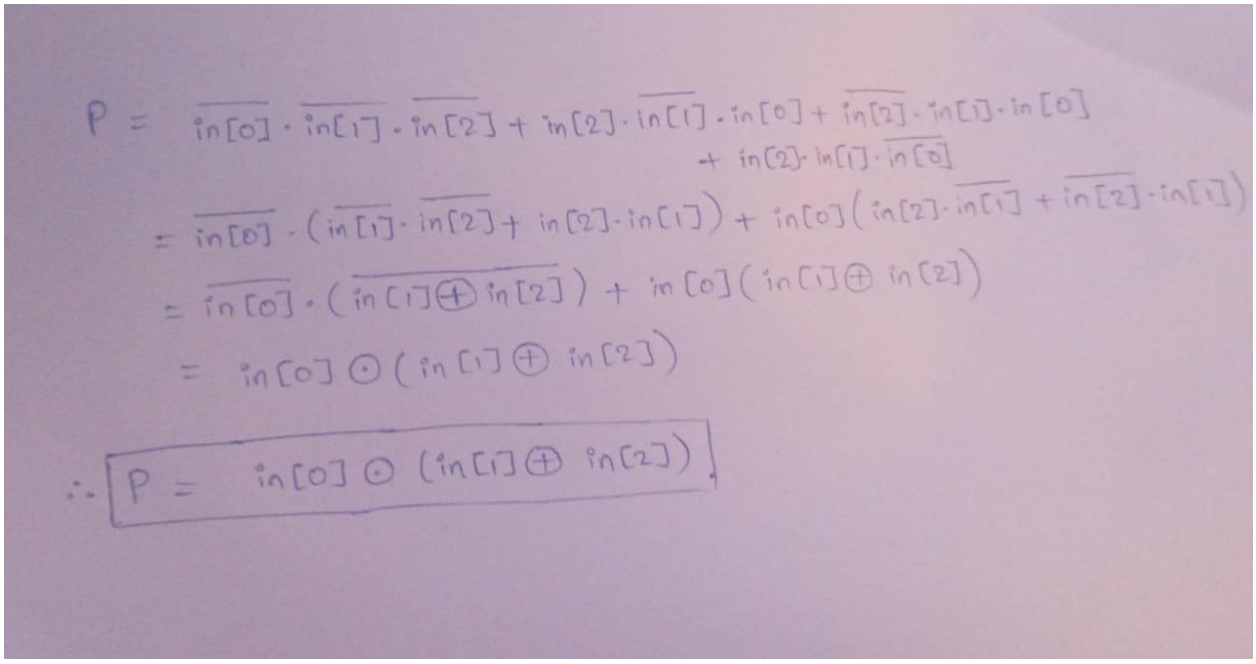


in[2] \ in[1] in[0]	00	01	11	10
0	1	0	1	0
1	0	1	0	1

Where, ‘in’ represents the 3-bit string which is to be read through 3 consecutive 1-bit inputs.

- **Logic**

The following logic can be obtained from the k-map drawn above:



- **Circuit**

The logic in the above image can be implemented as following: