Modeling Finite State Machines(FSMs)

What is a State Machine?

A state machine is a concept used in designing computer programs or digital logic. A state machine is any device that traverses through a predetermined sequence of states in an orderly fashion.

In Summary, a state machine can be described as:

- A set of input events
- A set of output events
- A set of states
- A function that maps states and input to output
- A function that maps states and inputs to states (which is called a state transition function)
- A description of the initial state

Modeling Finite State Machines(FSMs)

There are two types of state machines:

- Finite state machines
- Infinite state machines

Finite State Machine:

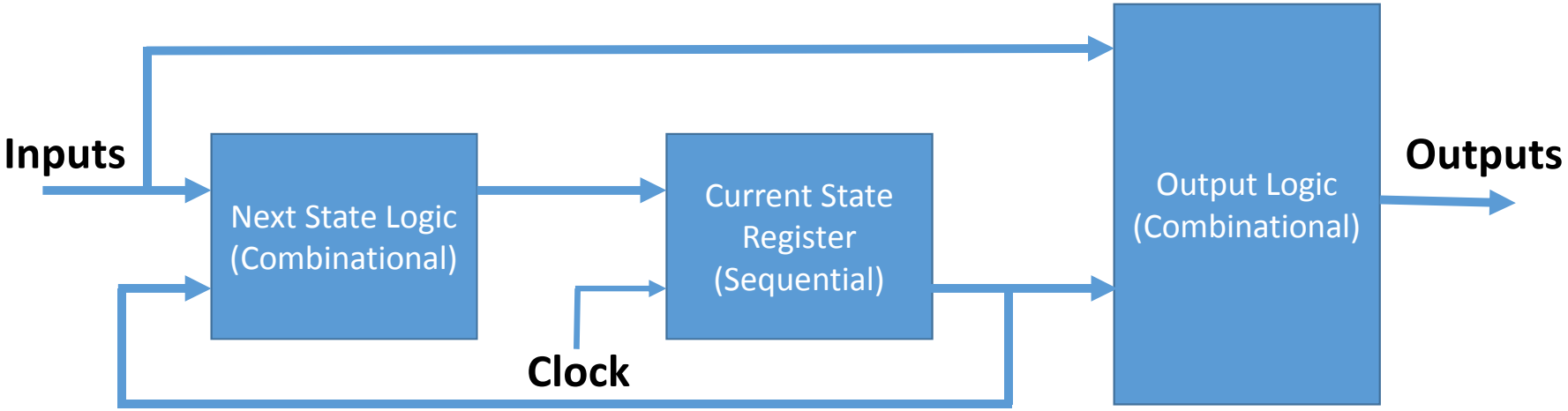
- Comprised of a finite number of states
- Comprised of a finite set of inputs and/or events that can trigger transitions between states
- Comprised of a finite set of outputs that can be describes as a function of state and input
- Comprised of an initial state

Infinite State Machine:

- An ISM can be conceived but is not practical and never used.

Structure of a Finite State Machine(FSM)

A general structure of a FSM is shown below:



Current State Register: Register of n-flip flops used to store the current state of the FSM. It is clocked from a free running clock source.

Next State Logic: Combinational logic used to generate next state in the sequence. Next state is a functional of FSM inputs and current state.

Output Logic: Combinational logic used to generate output signals. Outputs are a function of FSM current state and possibly inputs.

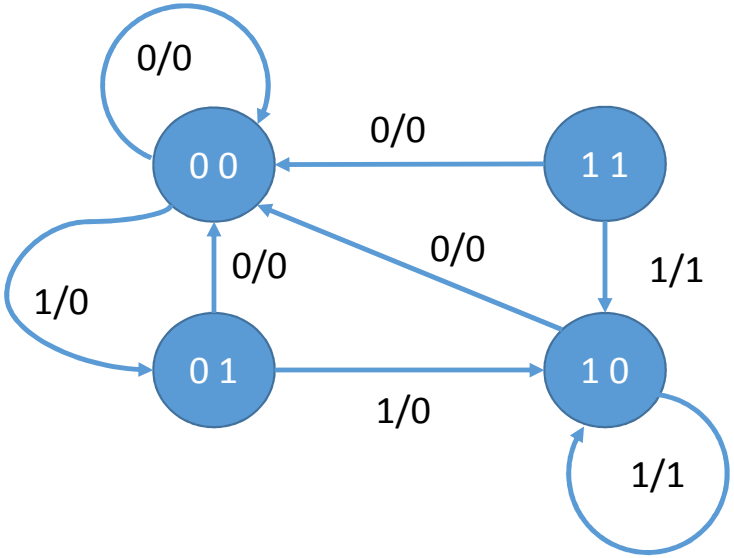
Modeling Finite State Machines(FSMs)

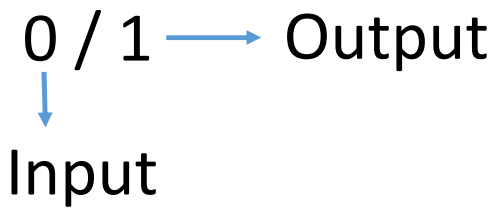
State Table: A state table is essentially a truth table that describes behavior of a FSM as a function of current state and inputs. For each combination of current state and inputs next state of the FSM is specified together with outputs. A state table is one of many ways to specify a state machine, other ways being a state diagram, and a characteristic equation.

Current State		Input	Next State		Output
Q0t	Q1t	X	Q0(t+1)	Q1(t+1)	Z
0	0	0	0	0	0
0	1	0	0	0	0
1	0	0	0	0	0
1	1	0	0	0	0
0	0	1	0	1	0
0	1	1	1	0	0
1	0	1	1	0	1
1	1	1	1	0	1

Modeling Finite State Machines(FSMs)

State Diagram: A state table is a graphical representation of FSM sequential operation. Figure below shows state diagram representation of the same state table described in previous slide. Circles represent states and lines with arrows represent transitions between states which occur every clock cycle. The input signal conditions that dictate state transitions are indicated next to the appropriate line and before any slash(/). A slash is used to separate input and output signals.



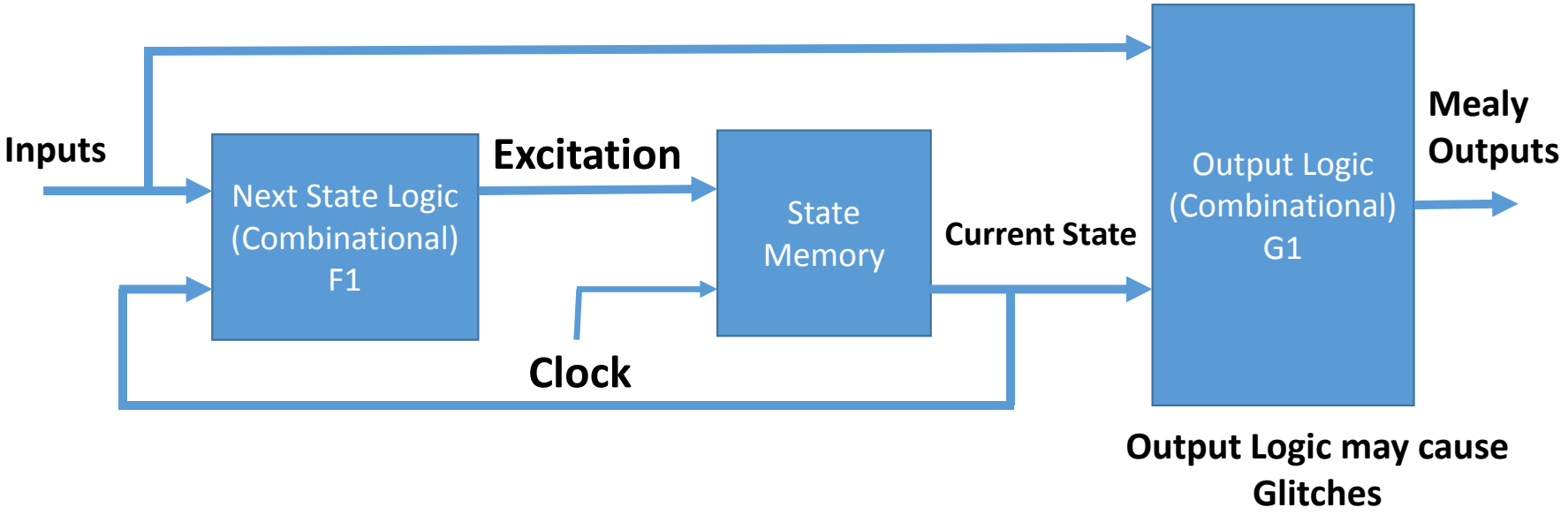


FSM Types

There are two types of FSMs:

- Mealy State Machines (outputs are function of current state and inputs)
- Moore State Machines (outputs are only function of current state)

Mealy State Machine

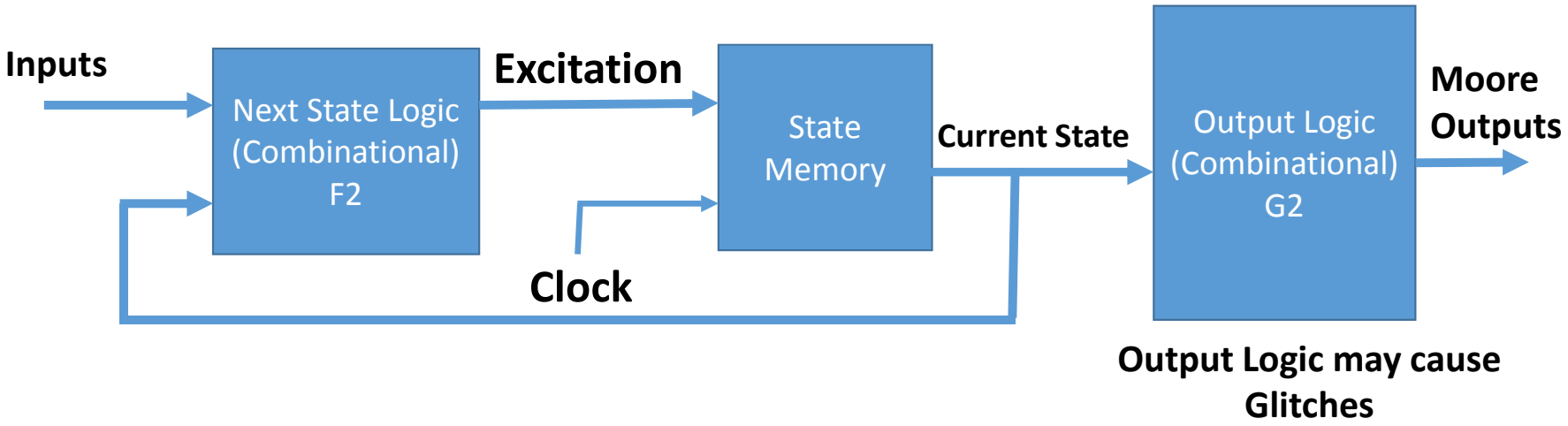


State Memory: Consists of n-flip flops that can store 2^n states.

Next state = F_1(current state, inputs)

Output = G_1(current state, inputs)

Moore State Machine

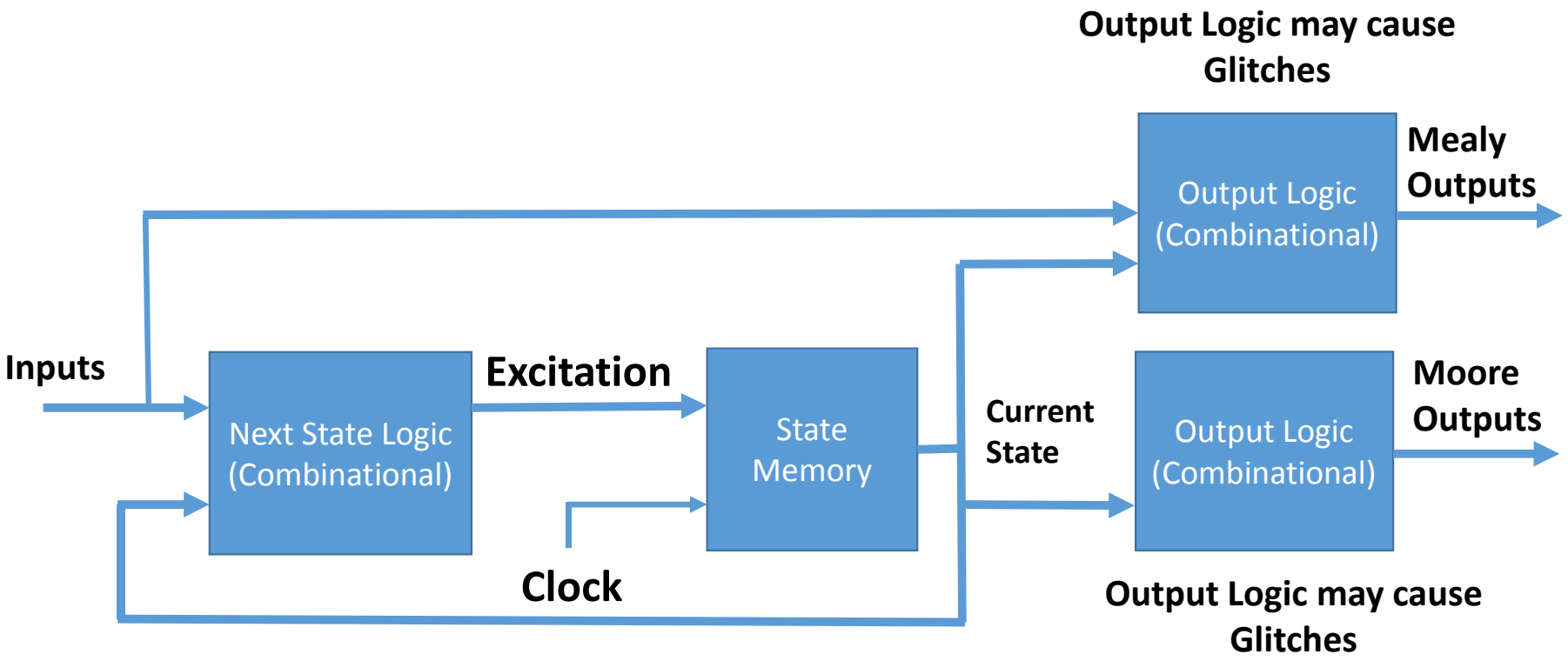


State Memory: Consists of n-flip flops that can store 2^n states.

Next state = F_2(current state, inputs)

Output = G_2(current state)

Mealy/Moore State Machine



Comparison of Mealy and Moore FSM

Mealy State Machine:

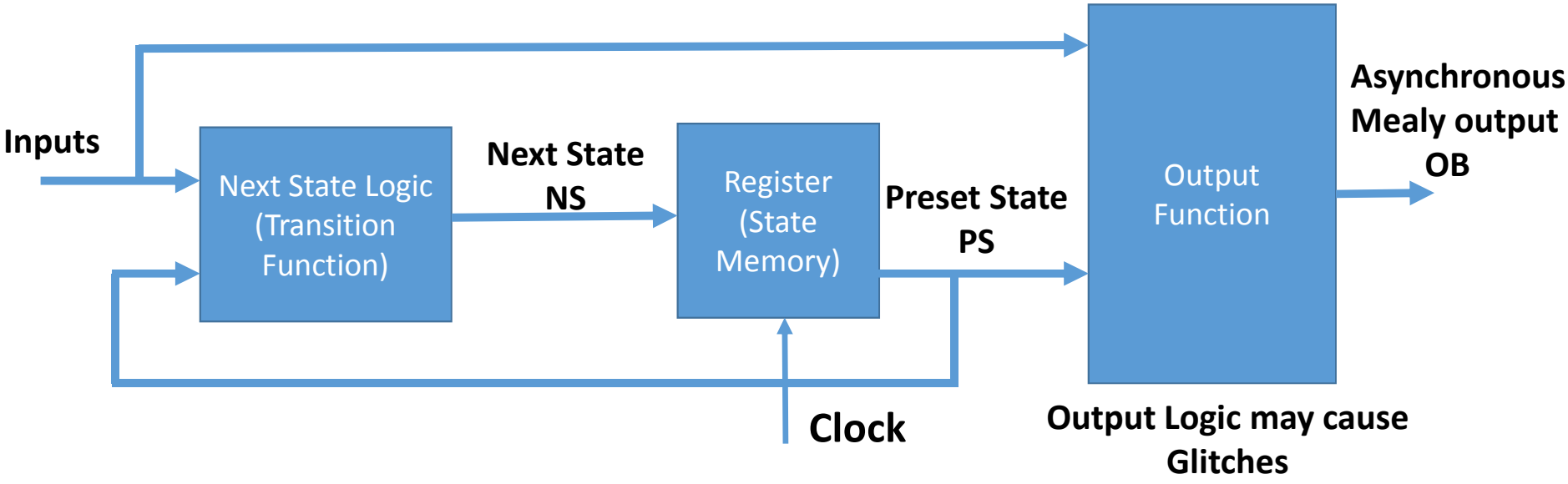
1. Output depends both upon present state and present input.
2. Generally, it has fewer states than Moore Machine(outputs are on transitions (n^2) rather than states (n).
3. Mealy machines react faster to inputs (React in same cycle - don't need to wait for clock). Inputs can effect outputs in current clock period.
4. Requires less hardware than Moore.
5. Outputs are unstable until current inputs achieve steady state.

Moore State Machine:

1. Output depends only upon the present state.
2. Generally, it has more states than Mealy Machine.
3. Moore machines are safer to use (Outputs change at clock edge - always one cycle later). Inputs can affect outputs in next clock period only.
4. More logic is needed to decode state into outputs because there may be more gate delays after clock edge.
5. Outputs are always stable since it depends only on current state which is always stable.

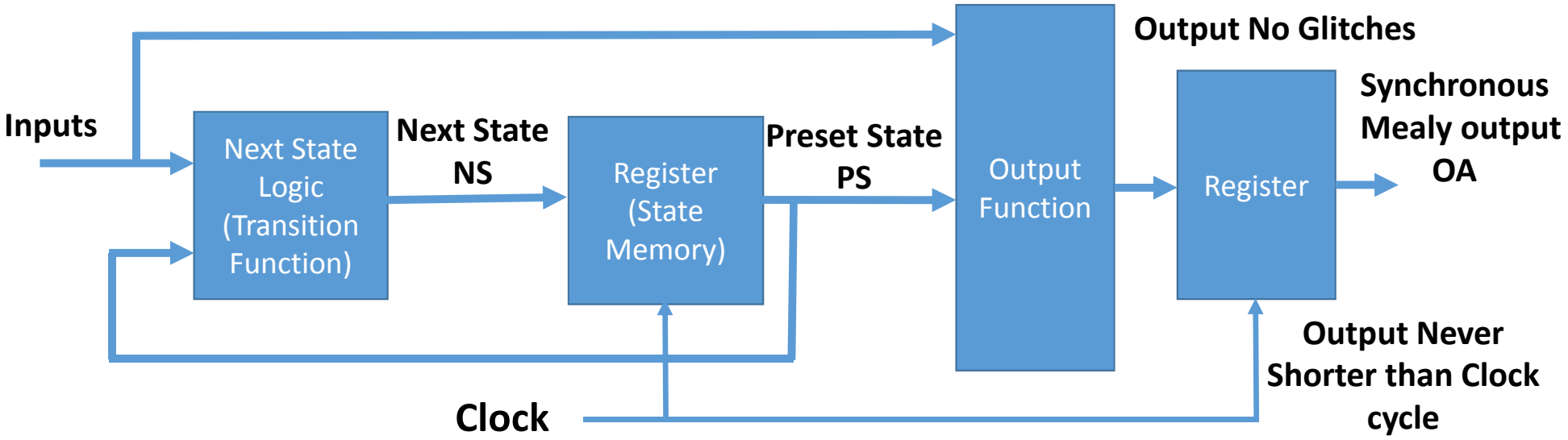
Asynchronous Mealy State Machine

The register outputs PS (Preset State) are fed back into the array and define the present state. The combinatorial logic implements the transition function, which produces the next state flip-flop inputs NS, and the output function, which produces the machine output OB. This is the asynchronous Mealy form.



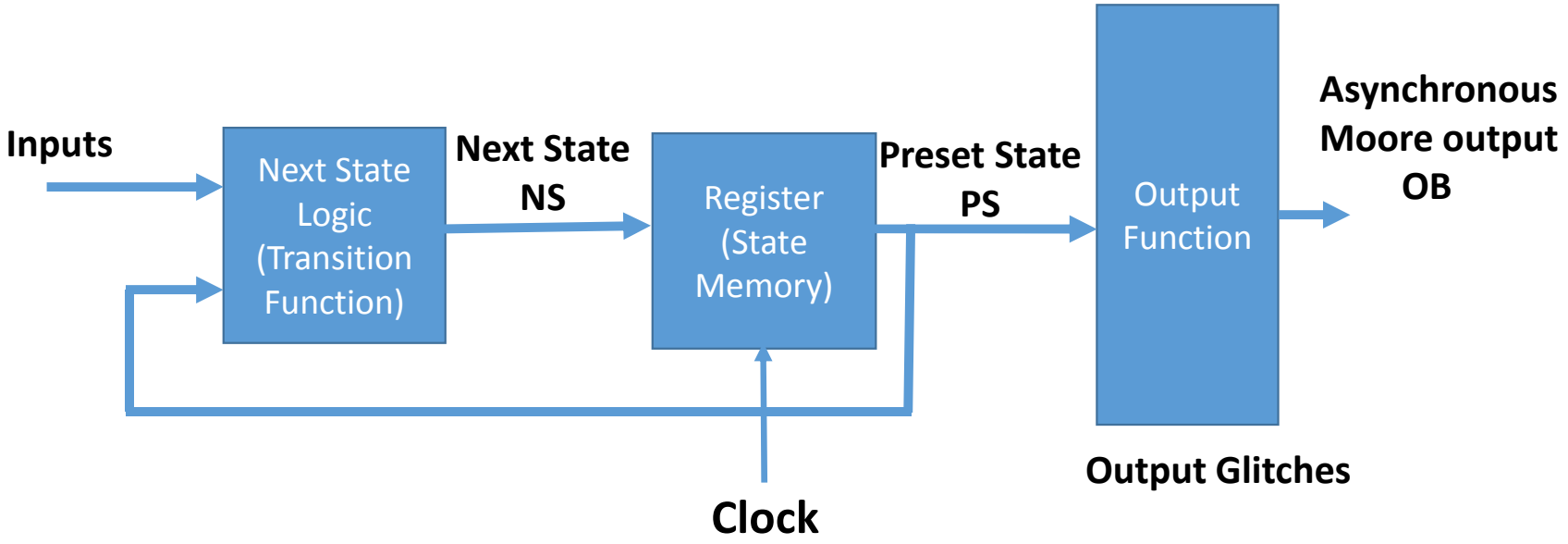
Synchronous Mealy State Machine

In Synchronous Mealy design the outputs are passed through an extra output register (OA) and thus, do not respond immediately to input changes. This is the synchronous Mealy form.



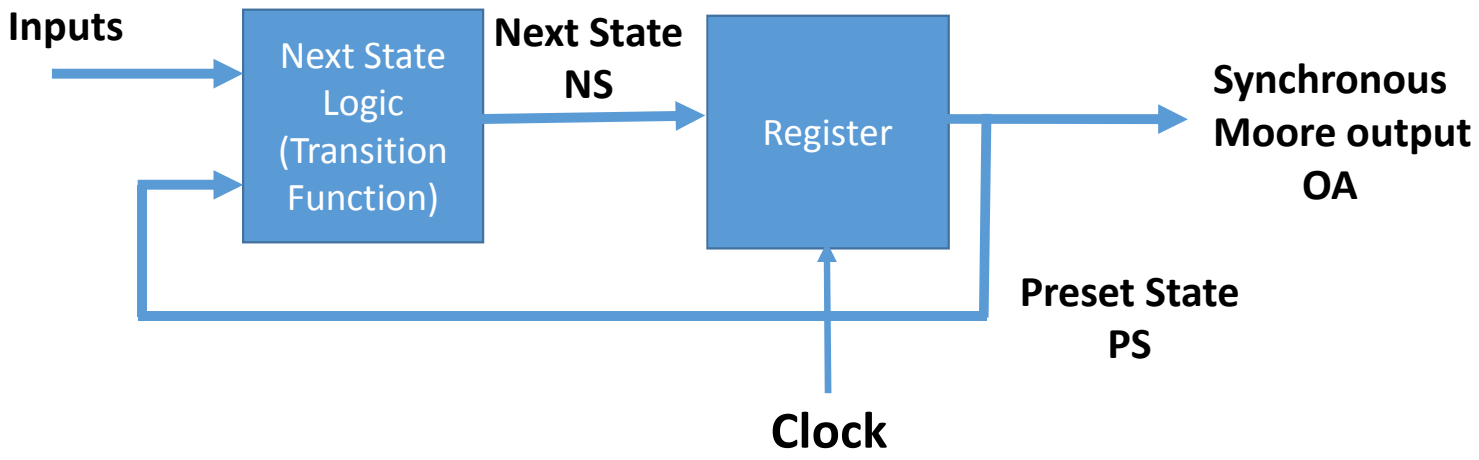
Asynchronous Moore State Machine

In Asynchronous Moore State machine design the outputs OB depend only on the present state PS. This is the asynchronous Moore form.



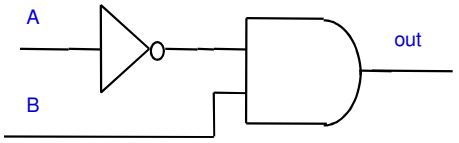
Synchronous Moore State Machine

The synchronous Moore form is shown in Figure below. In this case the combinatorial logic can be assumed to be the unity function. The outputs (OB) can be generated directly along with the present state (PS). Although these forms have been described separately, a single sequencer is able to realize a machine that combines them, provided that the required paths exist in the device.

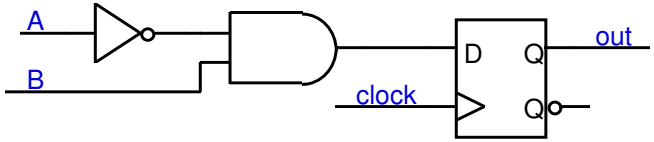


Mealy and Moore Example

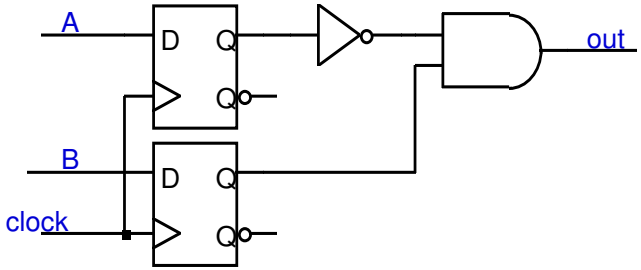
Mealy or Moore ?



Not a State Machine



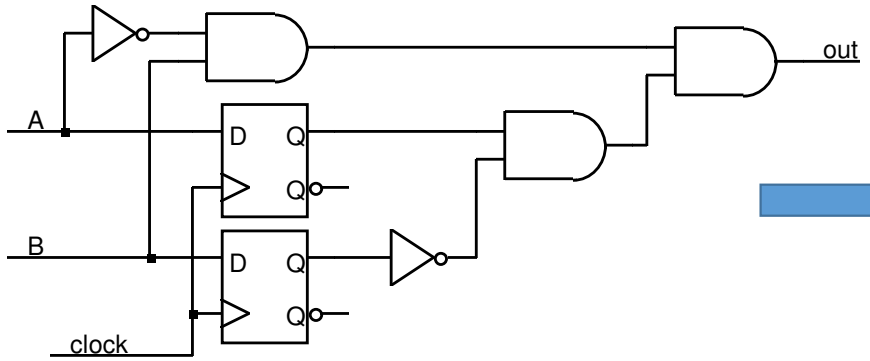
Moore: output = F(state)



Moore: output = F(state)

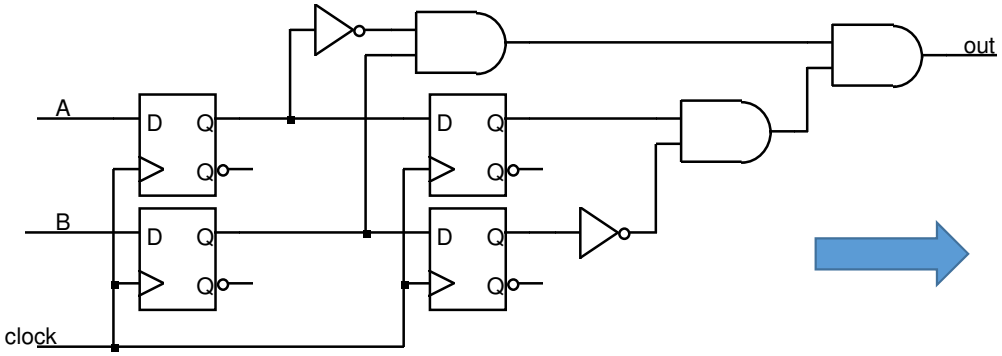
Mealy and Moore Example

Mealy or Moore ?



Mealy:

output = F(state, input)



Moore:

output = F(state)

Example1: A sequence detector (Mealy)

Lets design a sequence detector that produces an output of Z = 1 if an input sequence ending in 101 occurs, coincident with the last 1. The circuit does not reset when a 1 output occurs.

A typical input sequence and the corresponding output sequence are:

X = 0 0 1 1 0 1 1 0 0 1 0 1 0 1 0 0

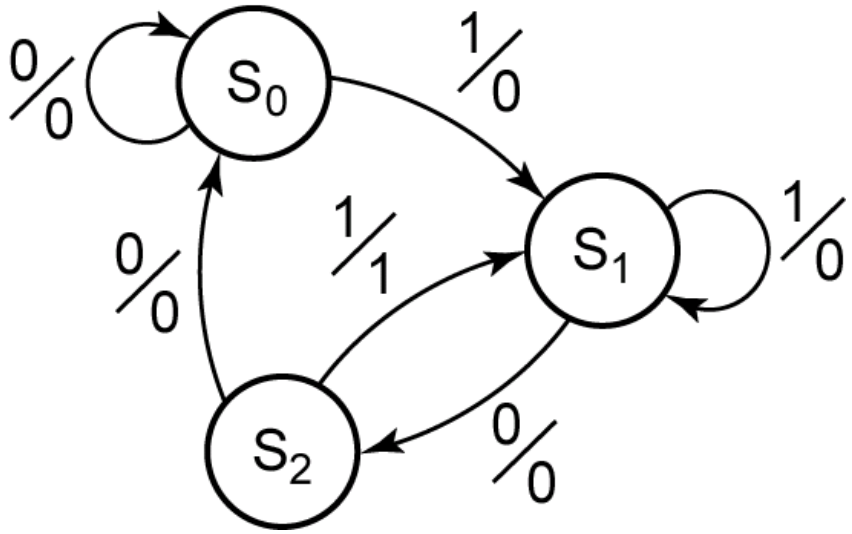
Z = 0 0 0 0 0 1 0 0 0 0 0 1 0 1 0 0

(time:0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)

Initially we do not know how many flip-flops will be required, so we will designate the circuit states as S_0, S_1, etc. We will start with a reset state designated S_0. If a 0 input is received, the circuit can stay in S_0 because the input sequence we are looking for does not start with a 0.

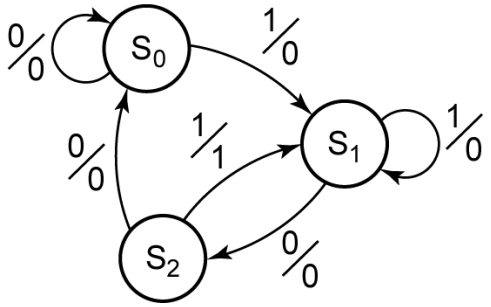
Example1: A sequence detector (Mealy)

Lets Draw the state diagram for the sequence detector. Since the sequence we are looking to detect is 101, lets define 3 states S0, S1 and S2.



Example1: A sequence detector (Mealy)

Lets Convert the state diagram to state table. Since there are 3 states, we only need 2 flip-flops for the circuit.



Present State	Next State $X=0$	Next State $X=1$	Present Output $X=0$	Present Output $X=1$
S_0	S_0	S_1	0	0
S_1	S_2	S_1	0	0
S_2	S_0	S_1	0	1

Example1: A sequence detector (Mealy)

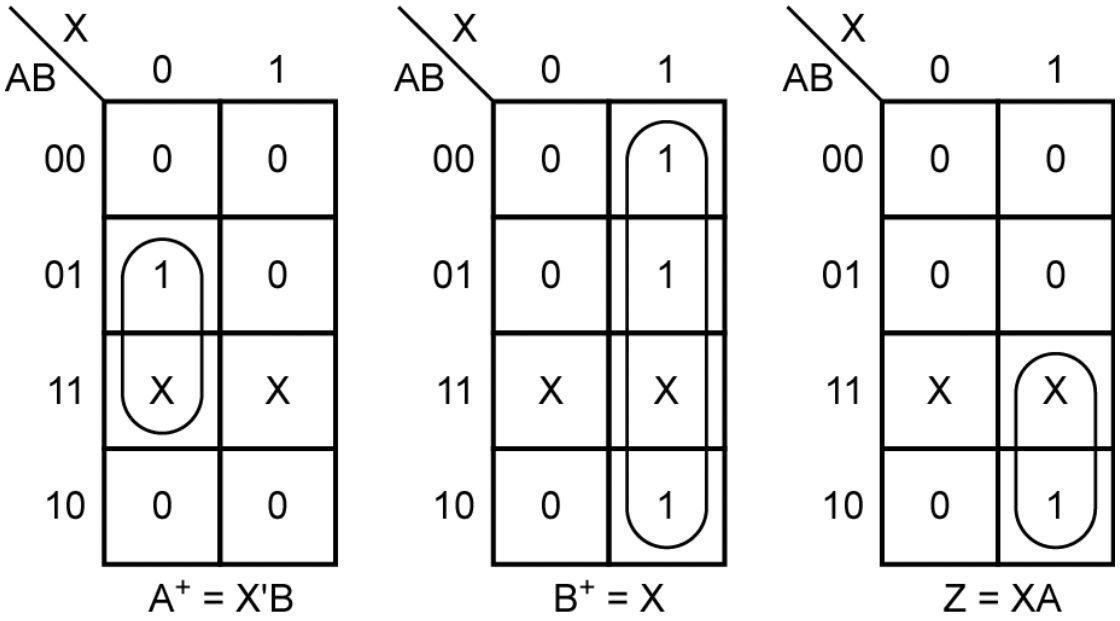
Lets convert our state table to a transition table:

Present State	Next State $X=0$	Next State $X=1$	Present Output $X=0$	Present Output $X=1$
S_0	S_0	S_1	0	0
S_1	S_2	S_1	0	0
S_2	S_0	S_1	0	1

PS	**NS** A^+B^+		**Output** Z	
AB	$X=0$	$X=1$	$X=0$	$X=1$
00	00	01	0	0
01	10	01	0	0
10	00	01	0	1

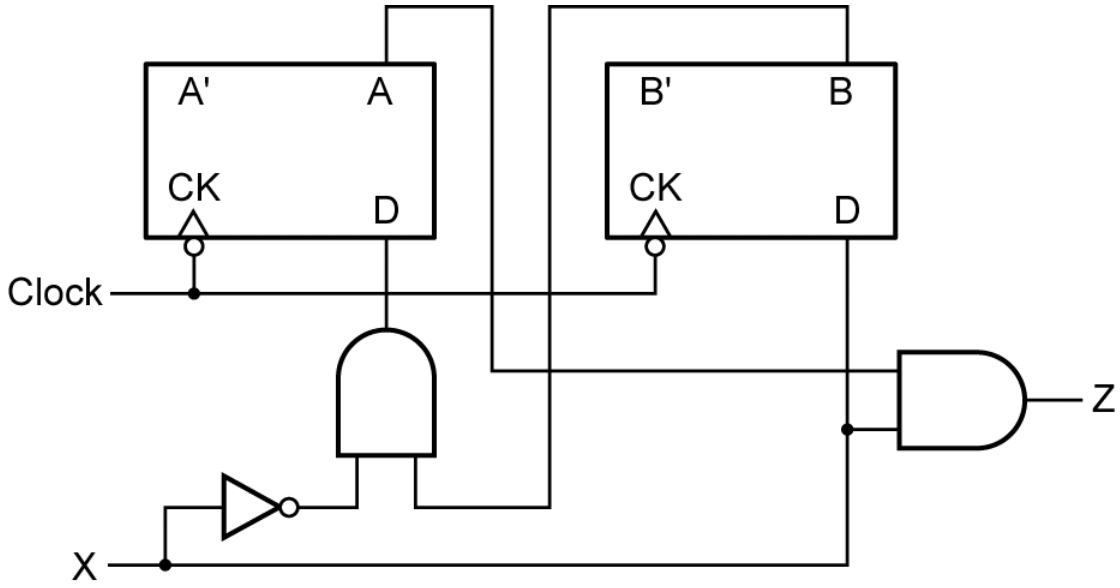
Example1: A sequence detector (Mealy)

Lets minimize the Next state of the Flip flop and the output using K-Map.



Example1: A sequence detector (Mealy)

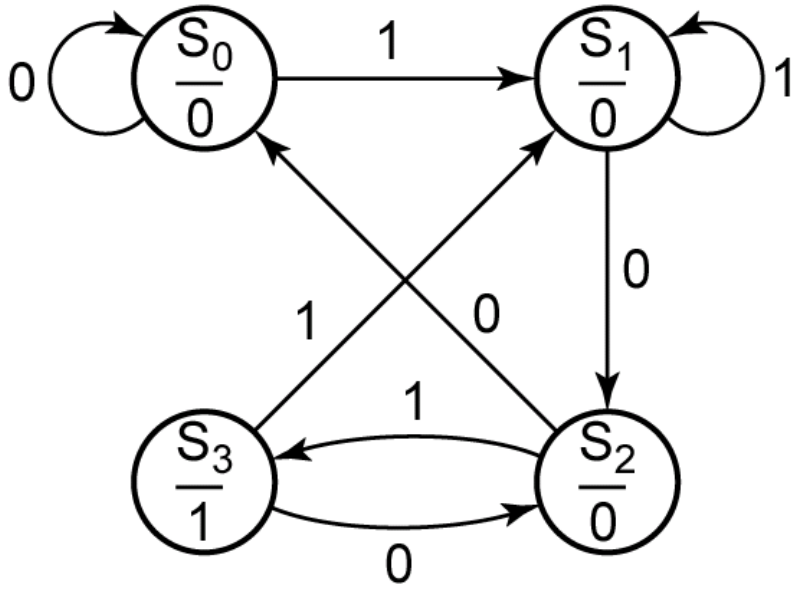
Using the derived equations, we can then draw the corresponding Mealy circuit diagram:



Output Z is a function of State and Input(X)

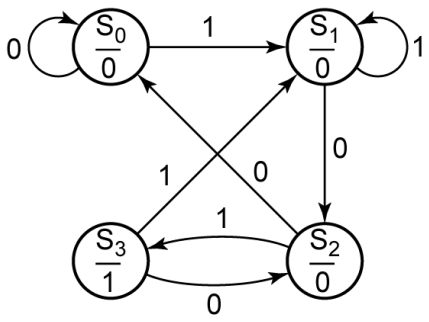
Example1: A sequence detector (Moore)

The procedure for finding the state graph for a Moore machine is similar to that used for a Mealy machine, except that the output is a function of the state. Let's take the previous example as a **Moore** machine: the circuit should produce an output of 1 only if an input sequence ending in 101 has occurred. State diagram for the Moore sequence detector is shown below:



Example1: A sequence detector (Moore)

The state and transition tables can be derived from the state graph and is shown below:



Present State	Next State $X=0$	Next State $X=1$	Present Output(Z)
S_0	S_0	S_1	0
S_1	S_2	S_1	0
S_2	S_0	S_3	0
S_3	S_2	S_1	1

AB	A^+B^+ $X=0$	A^+B^+ $X=1$	Z
00	00	01	0
01	11	01	0
11	00	10	0
10	11	01	1

State Reduction/Minimization

Goal?

- Given a state table reduce the number of states.
- Eliminate redundant states

Objectives?

- **Reduce the number of states in the state table to the minimum.**
  - Remove redundant states
  - use don't cares effectively
- **Reduction to the minimum number of states reduces**
  - The number of F/Fs needed
  - Reduces the number of next states that has to be generated → Reduced logic.

The three main methods of state reduction include:

- Row matching
- Implication table(chart)
- Successive partitioning.

State Reduction/Minimization

Row Matching: Row matching, which is the easiest of the three, works well for state transition tables which have an obvious next state and output equivalences for each of the present states. This method will generally not give the most simplified state machine available, but its ease of use and consistently fair results is a good reason to pursue the method.

It uses the state equivalence theorem: Si = Sj if and only if for every single input X, the outputs are the same and the next states are equivalent.

Here are the steps:

1. Start with the state transition table
2. Identify states with same output behavior
3. If such states transition to the same next state, they are equivalent
4. Combine into a single new renamed state
5. Repeat until no new states are combined

State Reduction/Minimization

Row Matching Example:

Input Sequence	Present State	Next State		Output	
		X = 0	X = 1	X = 0	X = 1
Reset	S0	S1	S2	0	0
0	S1	S3	S4	0	0
1	S2	S5	S6	0	0
00	S3	S7	S8	0	0
01	S4	S9	S10	0	0
10	S5	S11	S12	0	0
11	S6	S13	S14	0	0
000	S7	S0	S0	0	0
001	S8	S0	S0	0	0
010	S9	S0	S0	0	0
011	S10	S0	S0	1	0
100	S11	S0	S0	0	0
101	S12	S0	S0	1	0
110	S13	S0	S0	0	0
111	S14	S0	S0	0	0

State Reduction/Minimization

First, notice that S10 and S12 have the same next states as well as the same outputs. These two states can be combined and renamed into S10' (the tick mark ' signifies that this state has been merged with another and is considered a new, reduced state). Because S12 no longer exists, all instances of S12 will be replaced with S10'.

Input Sequence	Present State	Next State		Output	
		X = 0	X = 1	X = 0	X = 1
Reset	S0	S1	S2	0	0
0	S1	S3	S4	0	0
1	S2	S5	S6	0	0
00	S3	S7	S8	0	0
01	S4	S9	S10	0	0
10	S5	S11	S12	0	0
11	S6	S13	S14	0	0
000	S7	S0	S0	0	0
001	S8	S0	S0	0	0
010	S9	S0	S0	0	0
011 or 101	S10’	S0	S0	1	0
100	S11	S0	S0	0	0
110	S13	S0	S0	0	0
111	S14	S0	S0	0	0

State Reduction/Minimization

This process is continued until no new states can be combined. The resulting table can be seen below:

Input Sequence	Present State	Next State		Output	
		X = 0	X = 1	X = 0	X = 1
Reset	S0	S1	S2	0	0
0 1	S1 S2	S3' S4'	S4' S3'	0 0	0 0
00 or 11 01 or 10	S3' S4'	S7' S7'	S7' S7'	0 0	0 0
Not(011 or 101) 011 or 101	S7' S10'	S0 S0	S0 S0	0 1	0 0

In this particular example, the number of states was reduced from fifteen (15) to seven (7) states. This literally amounts to a reduction of eight (8) states and the elimination of one flip flop for the final design.

State Reduction/Minimization

Implication Chart Method: The implication chart method uses a graphical grid of sorts to systematically find equivalences among the states.

Here are the steps for the Implication Chart method:

1. Construct implication chart, one square for each combination of states taken two at a time.
2. Square labeled Si, Sj, if outputs differ then the square gets an 'X'. Otherwise, write down implied state pairs for all input combinations.
3. Advance through chart top-to-bottom and left-to-right. If square Si, Sj contains next state pair Sm, Sn and that pair labels a square already labeled 'X' then Si, Sj is labeled 'X‘.
4. Continue executing Step 3 until no new squares are marked with 'X‘.
5. For each remaining unmarked square Si, Sj, then Si and Sj are equivalent

State Reduction/Minimization

Consider the following transition table below:

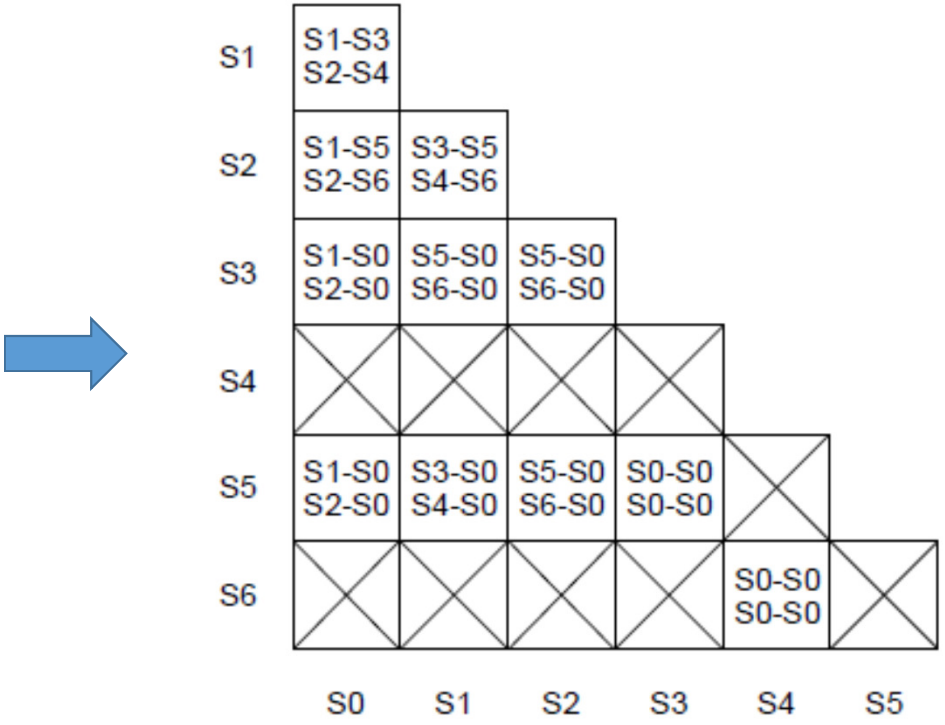
Input Sequence	Present State	Next State		Output	
		X = 0	X = 1	X = 0	X = 1
Reset	S0	S1	S2	0	0
0	S1	S3	S4	0	0
1	S2	S5	S6	0	0
00	S3	S0	S0	0	0
01	S4	S0	S0	1	0
10	S5	S0	S0	0	0
11	S6	S0	S0	1	0

Initial state transition table for implication chart method reduction.

State Reduction/Minimization

The first step is to construct an implication chart which can be seen in Table below: For the states to be equivalent, next state and output must be the same. Put 'X's in the table where the outputs differ. One column (or row) at a time, find implied pairs.

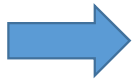
Constructing and completing the implication chart

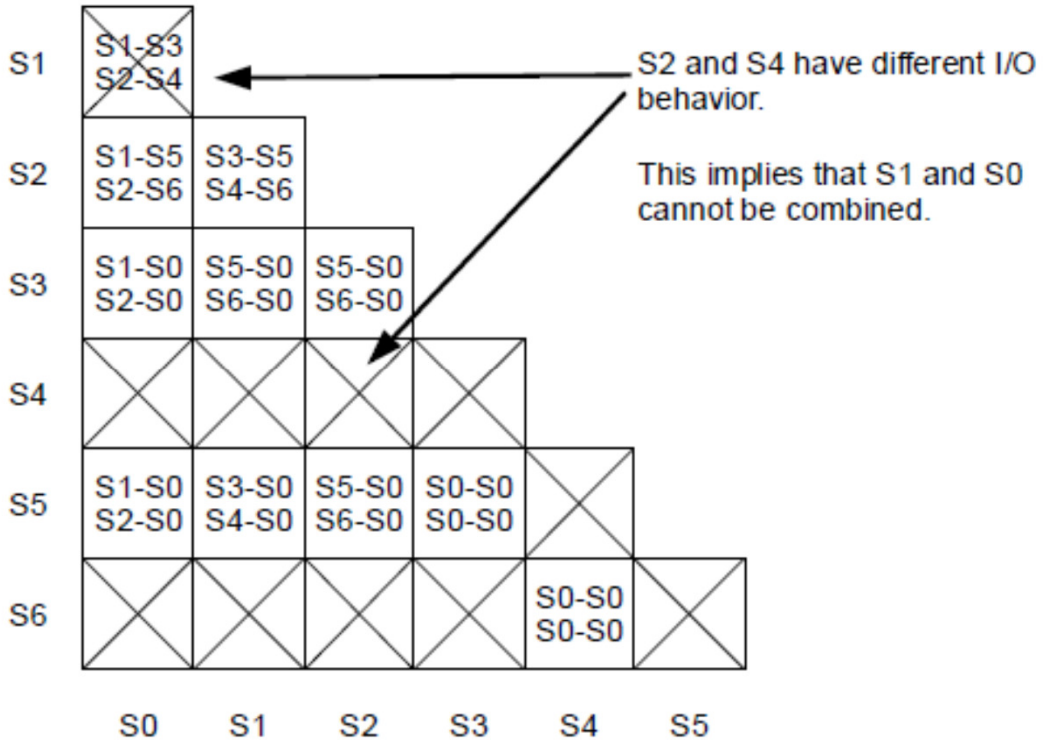


State Reduction/Minimization

In step 3, which is demonstrated in Table below, will further eliminate any states which cannot be implied.

A second pass which eliminates any additional states which cannot be implied.

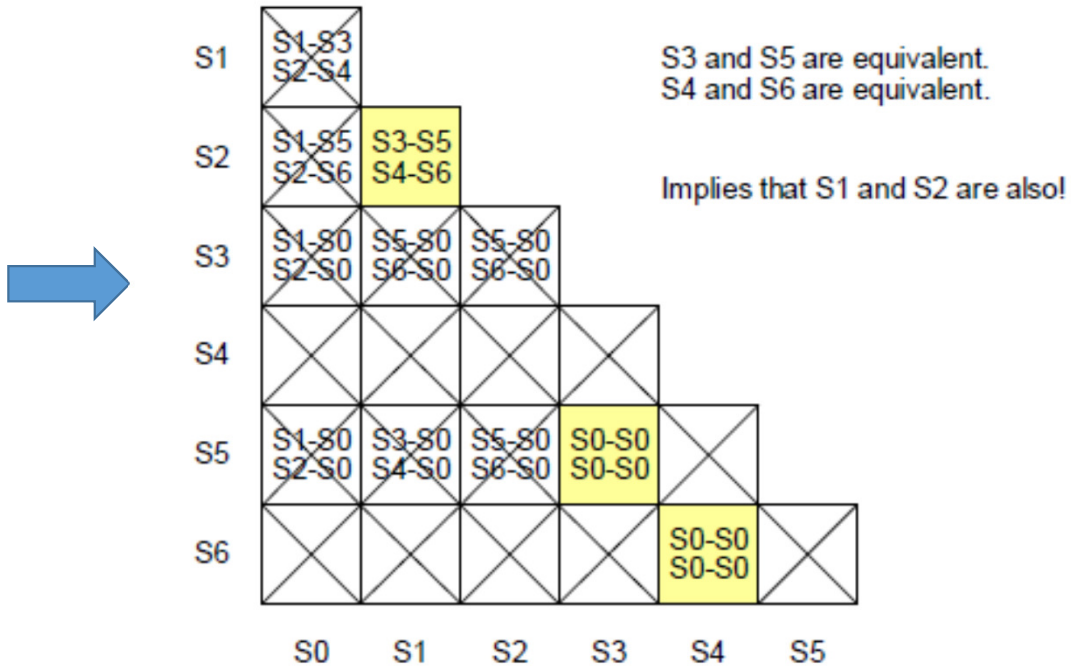




State Reduction/Minimization

Step 4 is to repeat the processes of step 3 until no more comparisons can be made. The final result is what is left over after this step is completed. The final implication chart is shown below:

Final implication table after all steps have been completed.



State Reduction/Minimization

The final state transition table is shown below. In this particular example, the implication chart method simplified the state transition table from seven (7) states down to five (5) states.

Input Sequence	Present State	Next State		Output	
		X = 0	X = 1	X = 0	X = 1
Reset	S0	S1’	S2’	0	0
0 or 1	S1’	S3’	S4’	0	0
00 or 10 01 or 11	S3’ S4’	S0 S0	S0 S0	0 1	0 0

Final state transition table.

State Assignment and Encoding

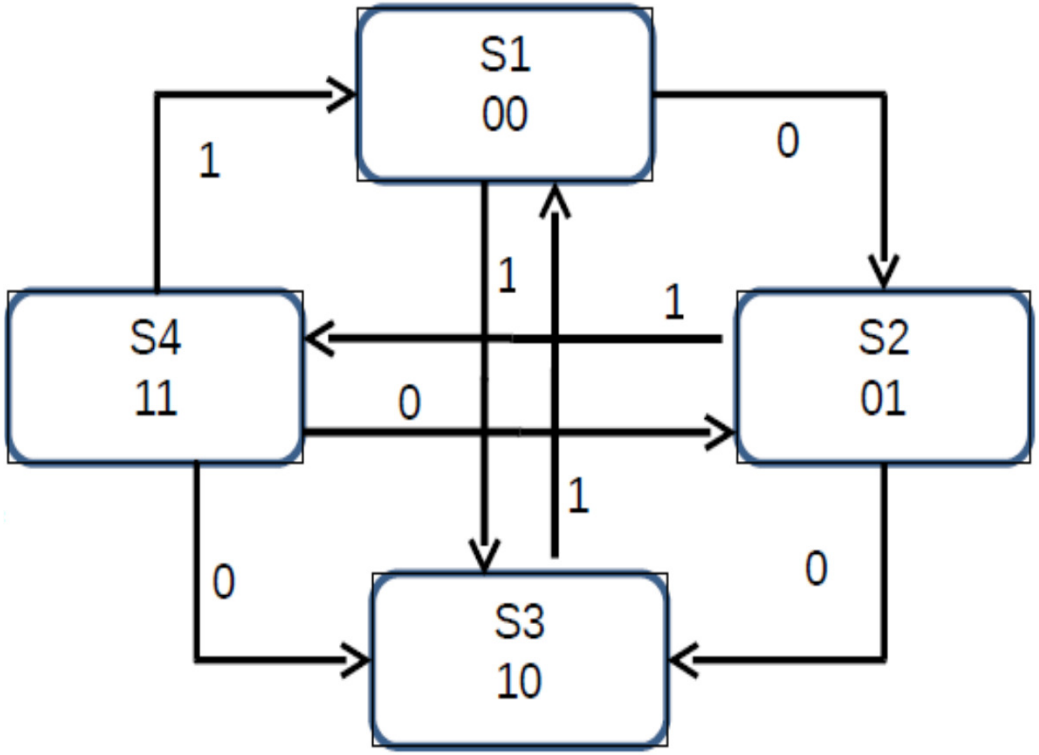
The way in which binary numbers are assigned to states is called state encoding. The different state encoding formats commonly used are:

- Binary(Sequential)
- Gray
- One-hot

Binary: Each state is assigned increasing binary numbers. However, this method is very inefficient. There are a minimum number of bits used to encode the machine, but the encoding equations derived for each bit are often large and complex.

State Assignment and Encoding

The state representation of binary code is shown below.



A state representation of binary code

The equations derived for table is:

Q0+ = (Q0' * Q1) + (X * Q0') + (Q0 * Q1' * X')

Q1+ = (X' * Q1') + (X' * Q0) + (X * Q0' * Q1)

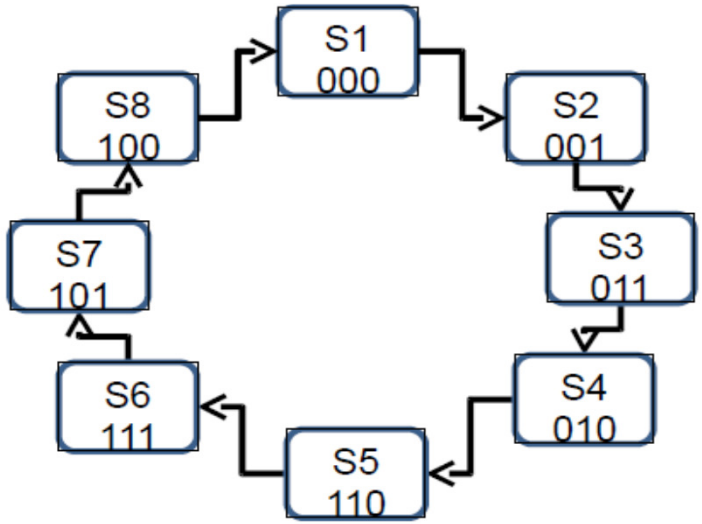
Because of this reason, different methods have been created that shorten the glue logic needed between the different gates.

State Assignment and Encoding

Gray: In gray code, each successive state differs from the previous state by only one bit. With only one bit changing from each state to the next, power consumption is reduced from binary code where multiple states can change at the same time. In addition, This allows for a minimum number of bits used and an also a small equation for each bit. However, the encoding only works if the machine transfers from one state to the next in order. If the states do not travel in order, then the gray code does not work very well.

The equations for each flip-flop are as shown:

A+ = BC' + AC
B+ = A'C + BC'
C+ = A'B' + AB



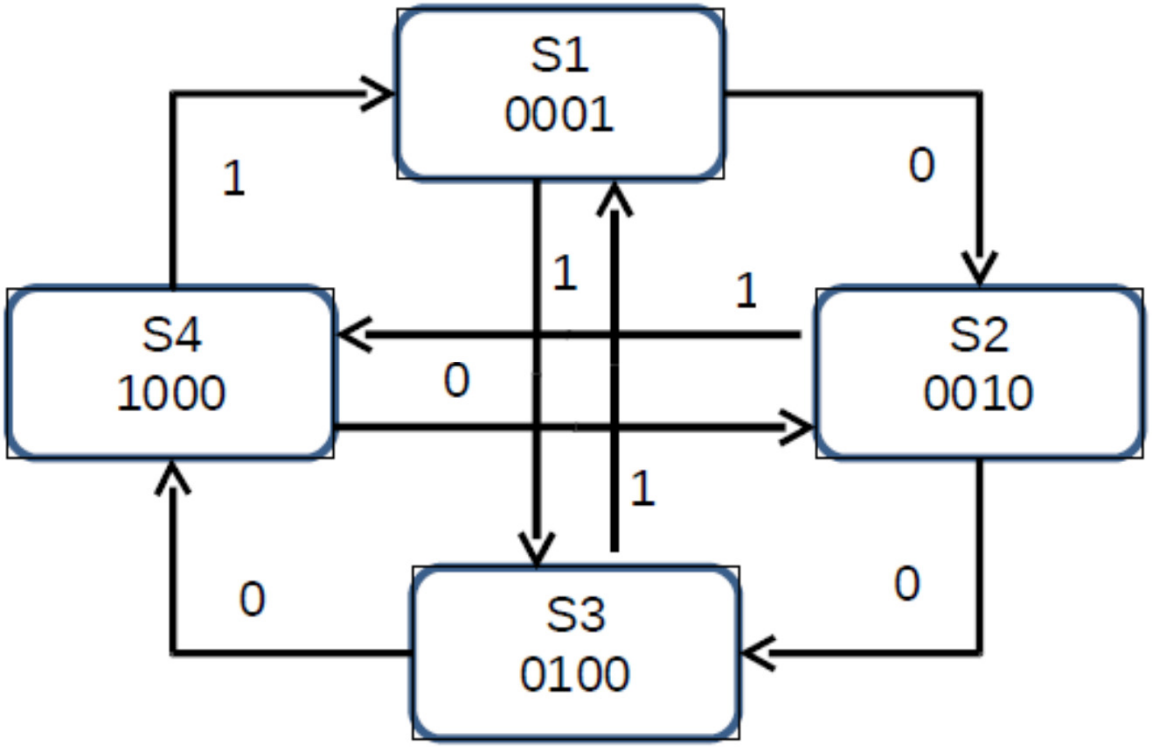
A map of an state gray code

State Assignment and Encoding

One Hot Encoding: One Hot Encoding is an encoding sequence that uses as many registers as there are states. With One Hot Encoding, only one of the bits are 1 or “Hot” at any given state. All the other bits are 0. This makes sure only two bits change when moving from one state to the next. With a maximum of two bits changing at any given time less power is consumed. In addition, One Hot Encoding reduces the logic needed to implement each bit. However, it also uses many more logic gates to implement the state design.

One Hot is often used where there are many states that need to be implemented in the design. Using binary encoding, the complexity of the design grows with each bit added. With One Hot, the number of flip-flops grows with each state added, but the complexity of each equation does not. Because of this, it is more difficult to accidentally mess up the logic needed to implement the bits. K-Maps are not needed for this type of encoding either. Instead, a truth table is created and used to find the equation for each bit.

State Assignment and Encoding



One Hot Encoding Map

State Assignment and Encoding

Input	Current State				Next State			
X	Q0	Q1	Q2	Q3	N0	N1	N2	N3
0	0	0	0	1	0	0	1	0
0	0	0	1	0	0	1	0	0
0	0	1	0	0	1	0	0	0
0	1	0	0	0	0	0	1	0
1	0	0	0	1	0	1	0	0
1	0	0	1	0	1	0	0	0
1	0	1	0	0	0	0	0	1
1	1	0	0	0	0	0	0	1

Truth Table One Hot Encoding Map

State Assignment and Encoding

Using the standard method, the following equations can be easily derived by looking at the 1 on each next state.

N0 = (X'*Q0'*Q1*Q2'* Q3') + (X*Q0'*Q1'*Q2*Q3')

N1 = (X'*Q0'*Q1'*Q2*Q3') + (X*Q0'*Q1'*Q2'*Q3)

N2 = (X'*Q0'*Q1'*Q2'*Q3) + (X'*Q0*Q1'*Q2'*Q3')

N3 = (X*Q0'*Q1*Q2'*Q3') + (X*Q0*Q1'*Q2'*Q3')

Since only one of these states are Hot at any given time, this means if Q0 is 1, then Q1, Q2 and Q3 are all 0. Likewise with all bits, if one is a 1, then all others are 0. Because of this, the equations can be reduced to:

N0 = (X' * Q1) + (X * Q2)

N1 = (X' * Q2) + (X * Q3)

N2 = (X' * Q3) + (X' * Q0)

N3 = (X * Q1) + (X * Q0)

One hot encoding uses more bits to encode the states, but reduces the logic needed to glue the bits together. It is a very common coding technique used.