Chapter 7.
finite state machines (FSMs)

In chapter 6, we looked at counters, whose values are useful for representing states. Normally the number of states is finite. And a circuit or a system is modeled as a machine that makes transitions among states. The state is the main theme of this chapter. Typically counters are moving among states without external inputs but FSMs usually have external inputs. So FSM is a kind of a superset.

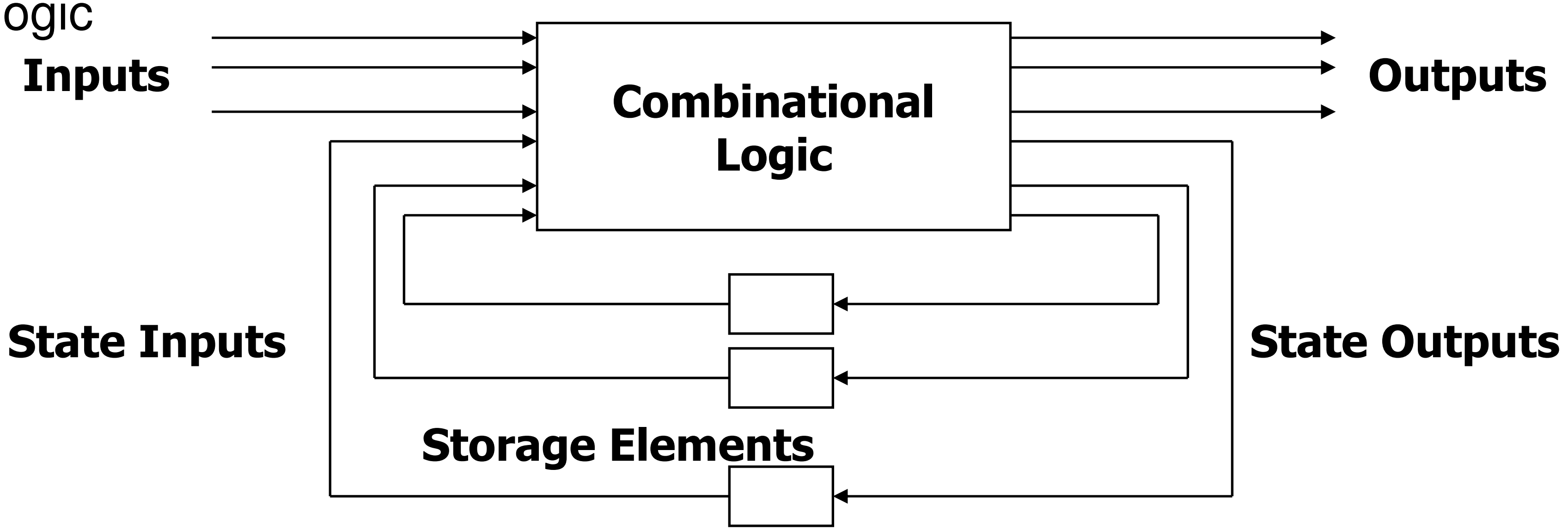
Finite State Machines

- Sequential circuits
  - primitive sequential elements
  - combinational logic
- Models for representing sequential circuits
  - finite-state machines (Moore and Mealy)
- Basic sequential circuits revisited
  - shift registers
  - counters
- Design procedure
  - state diagrams
  - state transition table
  - next state functions
- Hardware description languages

These are the topics that will be discussed in this chapter. Of course the next state depends on the current state and the input values. The FSMs fall into two categories: Moore and Mealy machines. Also we will look at how inputs are handled in sequential systems. Still registers and counters are key parts of the sequential circuits.

Abstraction of state elements

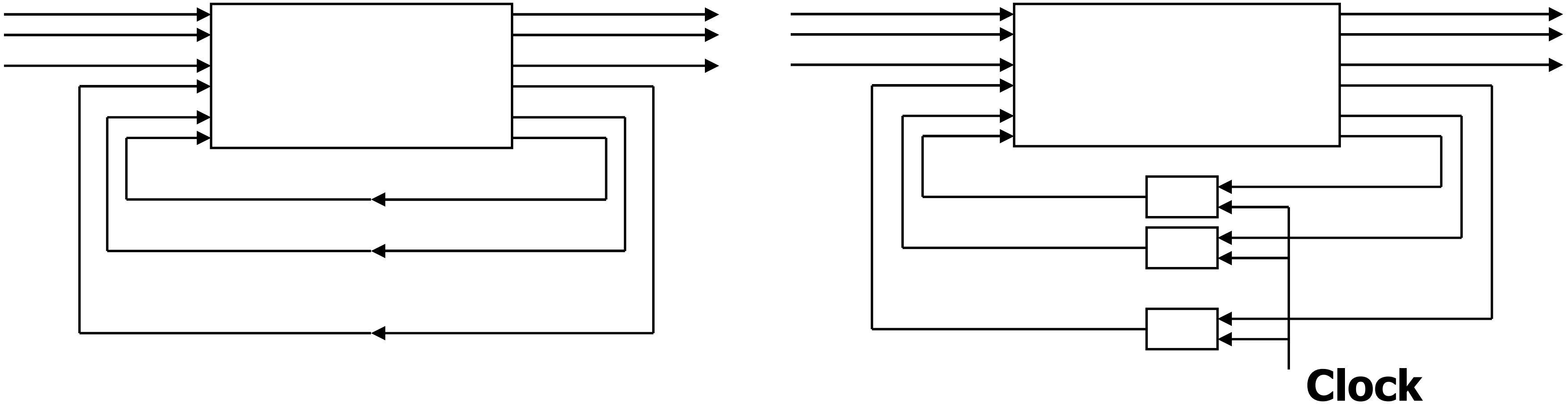
- Divide circuit into combinational logic and state
- Localize the feedback loops and make it easy to break cycles
- Implementation of storage elements leads to various forms of sequential logic



Now we are gonna break down a sequential logic system into two parts: combinational logic part and memory part (states). Multiple storage elements will be used to abstract the system states. The state, together with inputs, will determine the system operation: e.g. what is the next state, output?

Forms of sequential logic

- Asynchronous sequential logic – state changes occur whenever state inputs change (elements may be simple wires or delay elements)
- Synchronous sequential logic – state changes occur in lock step across all storage elements (using a periodic waveform - the clock)

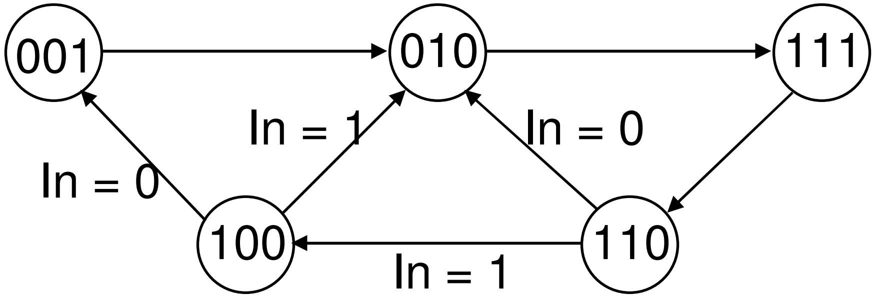


Asynchronous sequential logic circuits are operating without a clock, as shown on the left. The majority of sequential logic is synchronous logic circuits operating with clock signals, as illustrated on the right.. With the clock signal, it is more convenient to control state transitions.

Finite state machine representations

- States: determined by possible values in sequential storage elements
- Transitions: change of state
- Clock: controls when state can change by controlling storage elements

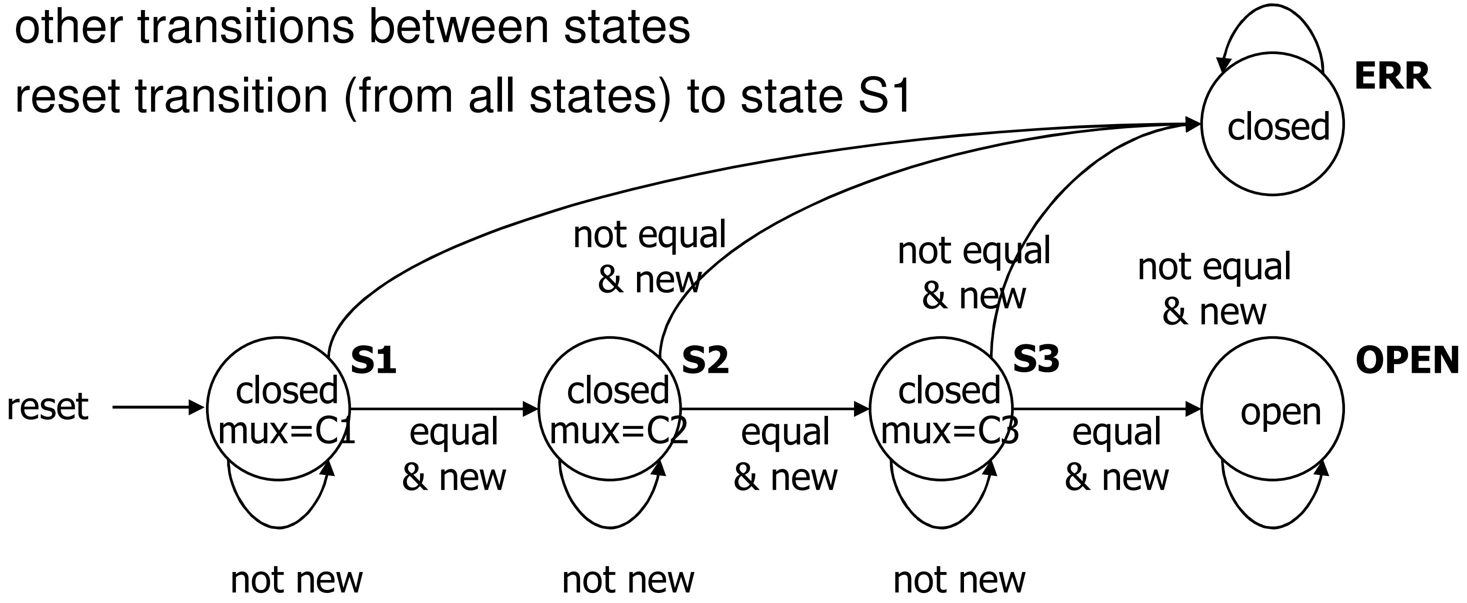
- Sequential logic
  - sequences through a series of states
  - based on sequence of values on input signals
  - clock period defines elements of sequence



Now we use 5 states to describe the system behavior. The number in the circle represents the state. The state transition is determined by the current state and the input. Sometimes, the state transition takes place without an input, e.g. just by a clock tick.

Example finite state machine diagram

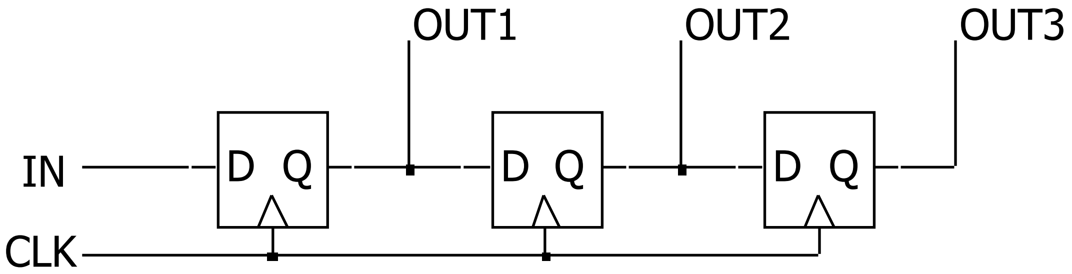
- Combination lock from introduction to course
  - 5 states
  - 5 self-transitions
  - 6 other transitions between states
  - 1 reset transition (from all states) to state S1

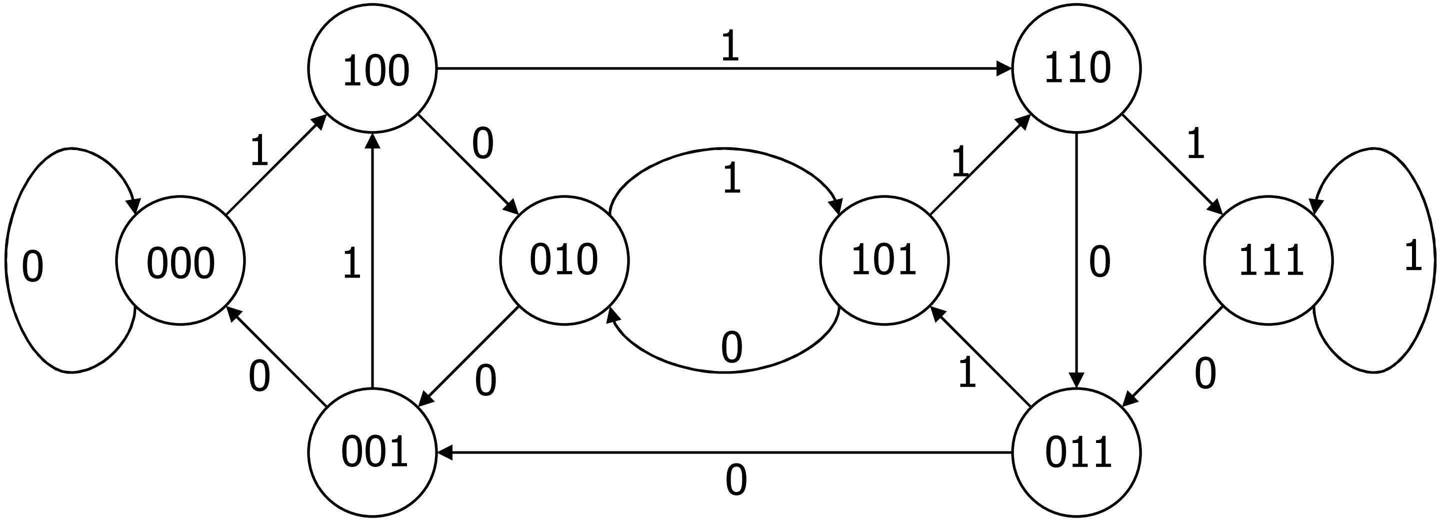


Let’s revisit the door combination lock system briefly. In the example in chapter 1, there were 5 states. There are two kinds of transitions: self-transition, and ordinary transition.

Can any sequential system be represented with a state diagram?

- Shift register
  - input value shown on transition arcs
  - output values shown within state node

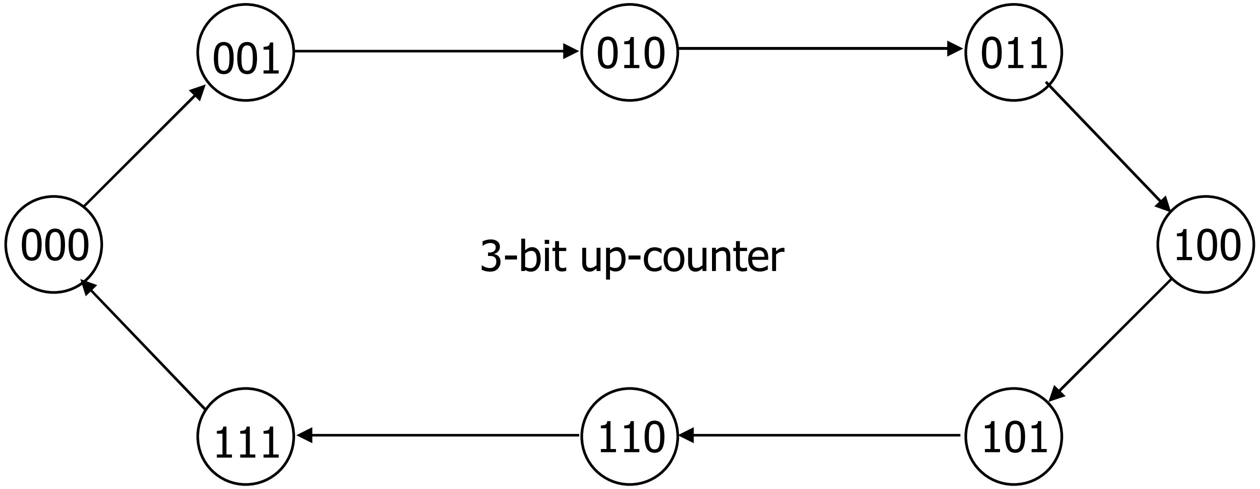




This state diagram shows all the possible states and transitions of a 3 bit shift register. In this case, the new state values are equal to output values. For example, when the system moves from 100 to 010, the output is 010. First of all, 3 bit will represent 8 states. And in each state, two input values are expected.

Counters are simple finite state machines

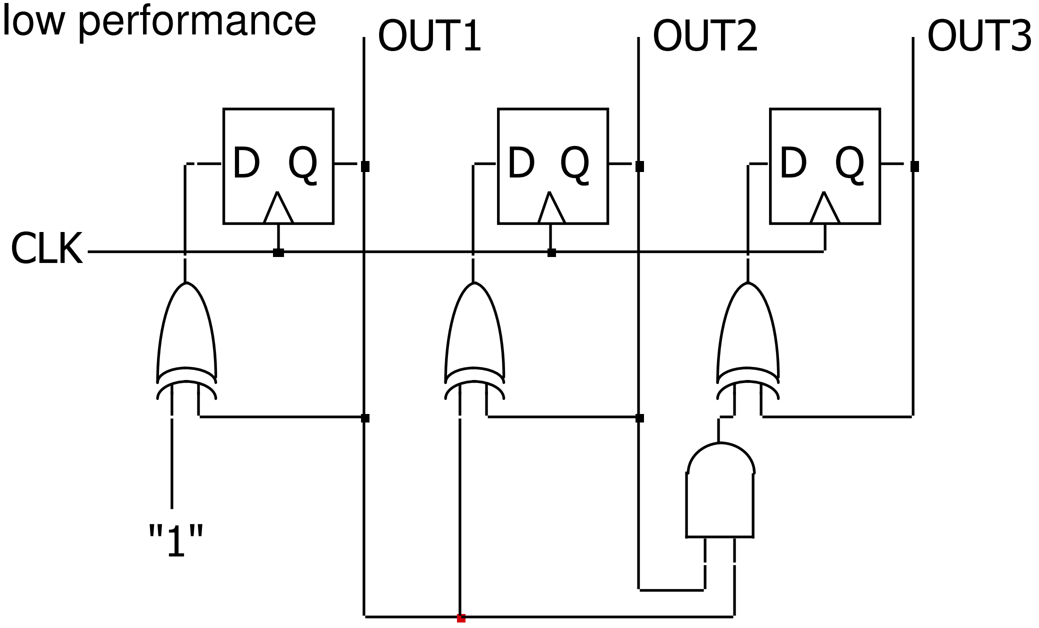
- Counters
  - proceed through well-defined sequence of states in response to enable
- Many types of counters: binary, BCD, Gray-code
  - 3-bit up-counter: 000, 001, 010, 011, 100, 101, 110, 111, 000, ...
  - 3-bit down-counter: 111, 110, 101, 100, 011, 010, 001, 000, 111, ...



In the case of a 3bit up counter, every clock tick will make a transition without any inputs. In this case the numbers follow binary coding; hence it is called a binary counter.

How do we turn a state diagram into logic?

- Counter
  - 3 flip-flops to hold state
  - logic to compute next state
  - clock signal controls when flip-flop memory can change
    - wait long enough for combinational logic to compute new value
    - don't wait too long as that is low performance



This one is a 3bit binary (up) counter. Recall that when all the lower bits are true, the higher bit should be toggled at the next clock tick.

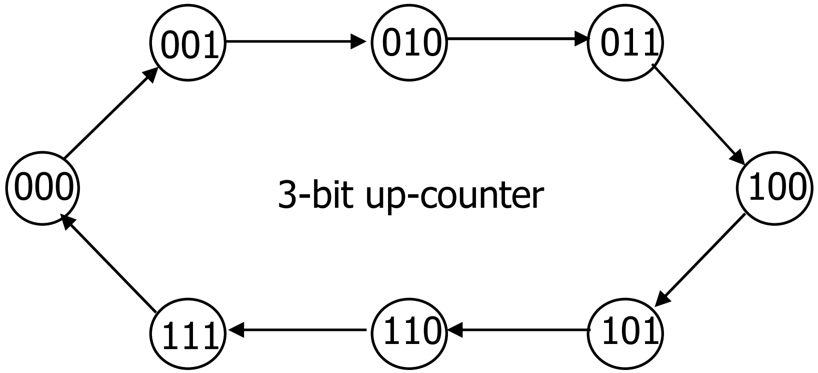
FSM design procedure

- Start with counters
  - simple because output is just state
  - simple because no choice of next state based on input
- State diagram to state transition table
  - tabular form of state diagram
  - like a truth-table
- State encoding
  - decide on representation of states
  - for counters it is simple: just its value
- Implementation
  - flip-flop for each state bit
  - combinational logic based on encoding

In the case of sequential logic, the design steps are somewhat different from those of combinational logic. First, we have to deal with states. If no external inputs, a counter is a good, simple choice. Then, how states are changed is described by a state transition table, which is similar to a truth-table. To embody the state transition table, we have to choose how to encode states as well as inputs and outputs.

FSM design procedure: state diagram to encode state transition table

- Tabular form of state diagram
- Like a truth-table (specify output for all input combinations)
- Encoding of states: easy for counters – just use value



present state		next state	
0	000	001	1
1	001	010	2
2	010	011	3
3	011	100	4
4	100	101	5
5	101	110	6
6	110	111	7
7	111	000	0

For the 3-bit up counter, here is the state transition table. It's like there are three inputs and three outputs. In this case the literals for states are the inputs for the state transition. If there are other outside inputs, those should be also written in the table.

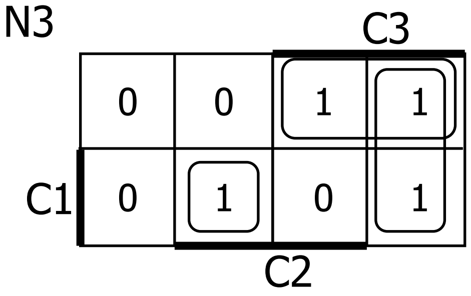
Implementation

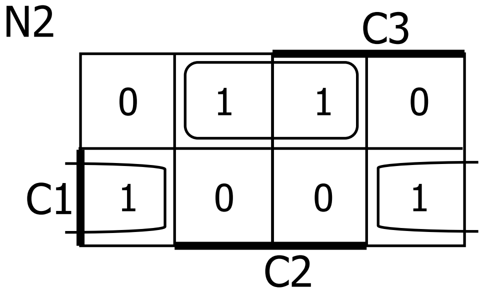
- D flip-flop for each state bit
- Combinational logic based on encoding

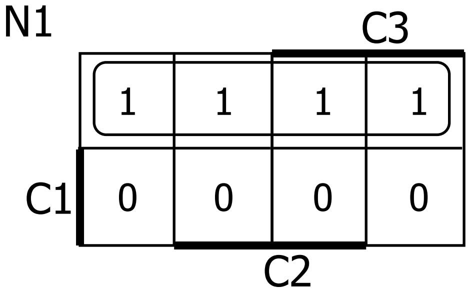
Verilog notation to show function represents an input to D-FF

C3	C2	C1	N3	N2	N1
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

N1 <= C1′
N2 <= C1C2′ + C1′C2
<= C1 <u>xor</u> C2
N3 <= C1C2C3′ + C1′C3 + C2′C3
<= (C1C2)C3′ + (C1′ + C2′)C3
<= (C1C2)C3′ + (C1C2)′C3
<= (C1C2) <u>xor</u> C3





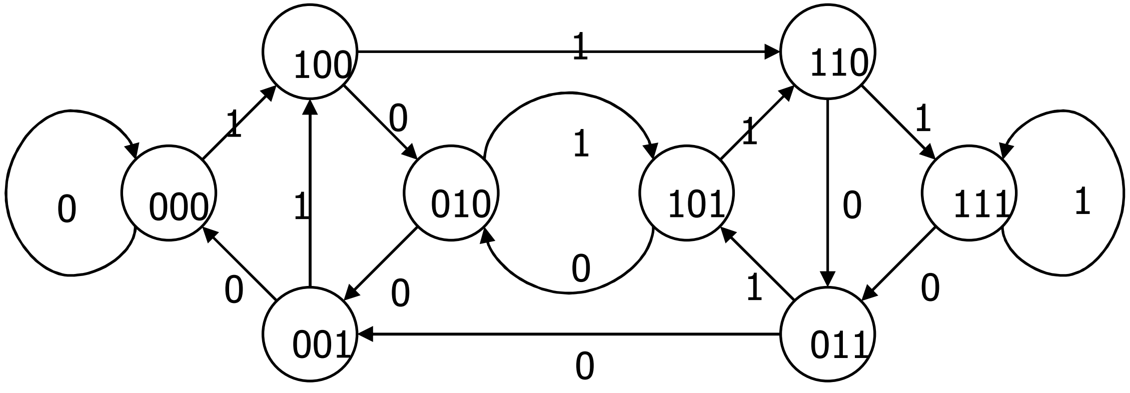


These are K-maps for three outputs. <= is a non-blocking assignment operator. That is, all values (N1, N2, N3) are changed at the same time, not sequentially.

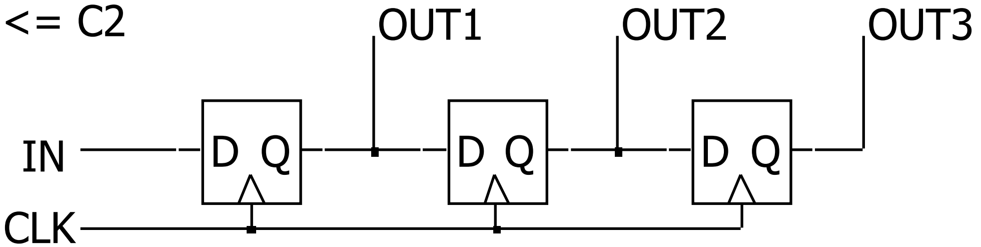
Back to the shift register

- Input determines next state

In	C1	C2	C3	N1	N2	N3
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	0	1	0
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	0	1	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	1	0	1
1	0	1	1	1	0	1
1	1	0	0	1	1	0
1	1	0	1	1	1	0
1	1	1	0	1	1	1
1	1	1	1	1	1	1



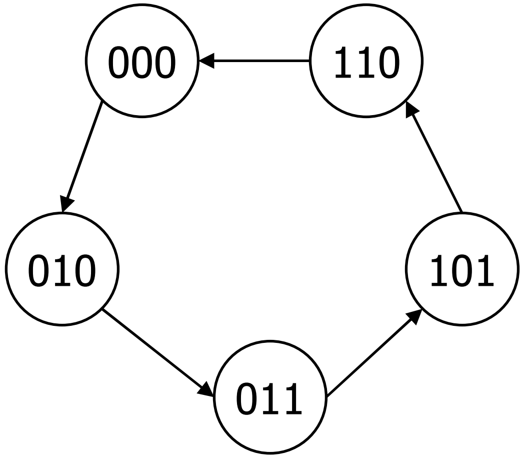
N1 <= In
N2 <= C1
N3 <= C2



Here is the state transition table for a 3bit shift register. In this case, there is one external input in addition to the current states.

More complex counter example

- Complex counter
  - repeats 5 states in sequence
  - not a binary number representation
- Step 1: derive the state transition diagram
  - count sequence: 000, 010, 011, 101, 110
- Step 2: derive the state transition table from the state transition diagram



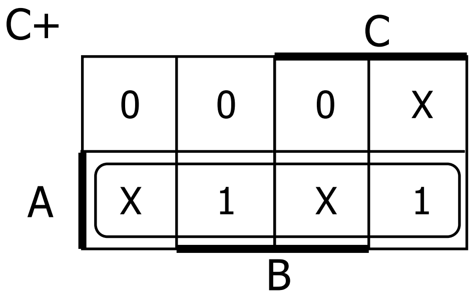
Present State			Next State		
C	B	A	C+	B+	A+
0	0	0	0	1	0
0	0	1	–	–	–
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	–	–	–
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	–	–	–

note the don't care conditions that arise from the unused state codes

In this case, only five states are used out of 8 possible binary values; so three don’t care cases appear. Note that the next state literals are denoted with + symbol.

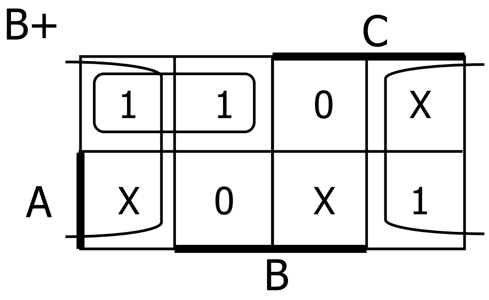
More complex counter example (cont'd)

- Step 3: K-maps for next state functions



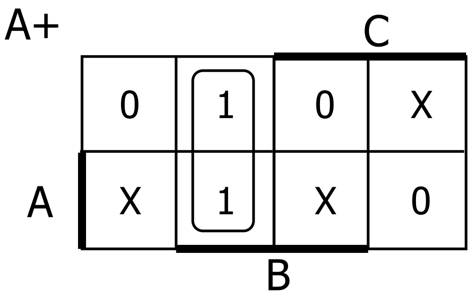
C+

	0	0	0	X
A	X	1	X	1



B+

	1	1	0	X
A	X	0	X	1



A+

	0	1	0	X
A	X	1	X	0

C+ <= A

B+ <= B′ + A′C′

A+ <= BC′

We see K-maps for three counter variables here.

Self-starting counters (cont'd)

- Re-deriving state transition table from don't care assignment

C+

			C	
	0	0	0	0
A	1	1	1	1
		B		

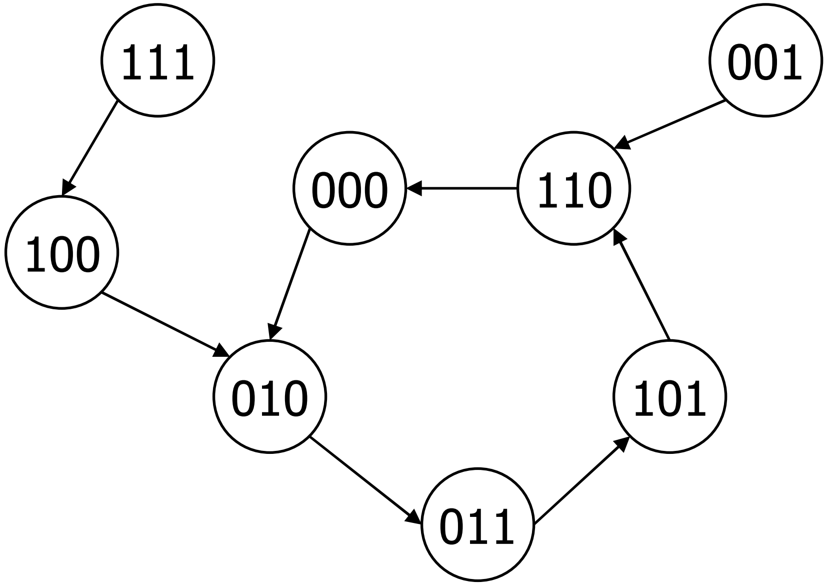
B+

			C	
	1	1	0	1
A	1	0	0	1
		B		

A+

			C	
	0	1	0	0
A	0	1	0	0
		B		

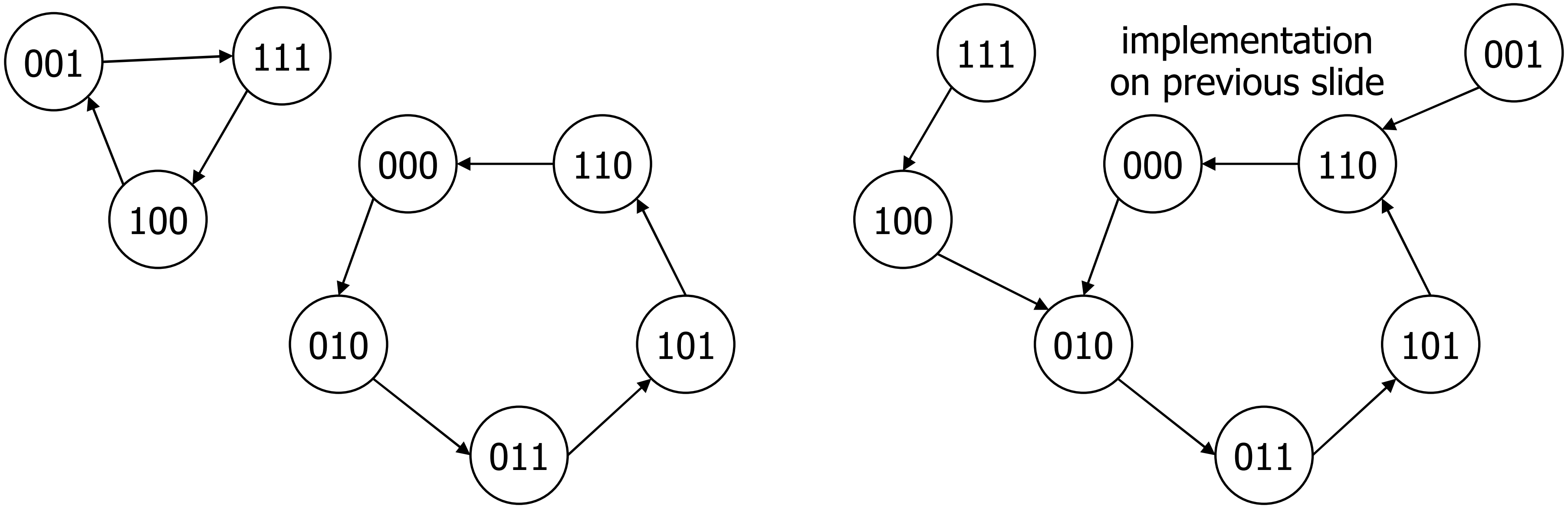
Present State			Next State		
C	B	A	C+	B+	A+
0	0	0	0	1	0
0	0	1	1	1	0
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	0	1	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	1	0	0



Counters have two categories: self-starting and non self-starting. In self-starting, even though the system starts in one of all possible states, which may not be legal, the system will eventually go to one of valid states. And then, the system will remain in the set of legitimate states. Note that there are no don't care terms.

Self-starting counters

- Start-up states
  - at power-up, counter may be in an unused or invalid state
  - designer must guarantee that it (eventually) enters a valid state
- Self-starting solution
  - design counter so that invalid states eventually transition to a valid state
  - may limit exploitation of don't cares



The left two counters are non-self-starting since if the system is in the state not shown in the state diagram, it will not work. Meanwhile the right one is self-starting.

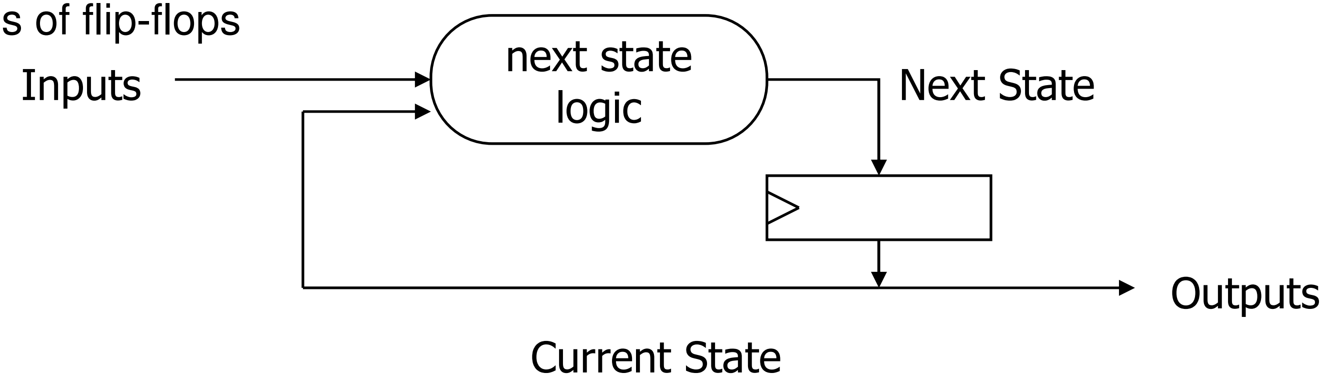
Activity

- 2-bit up-down counter (2 inputs)
  - direction: D = 0 for up, D = 1 for down
  - count: C = 0 for hold, C = 1 for count

Activity (cont’d)

Counter/shift-register model

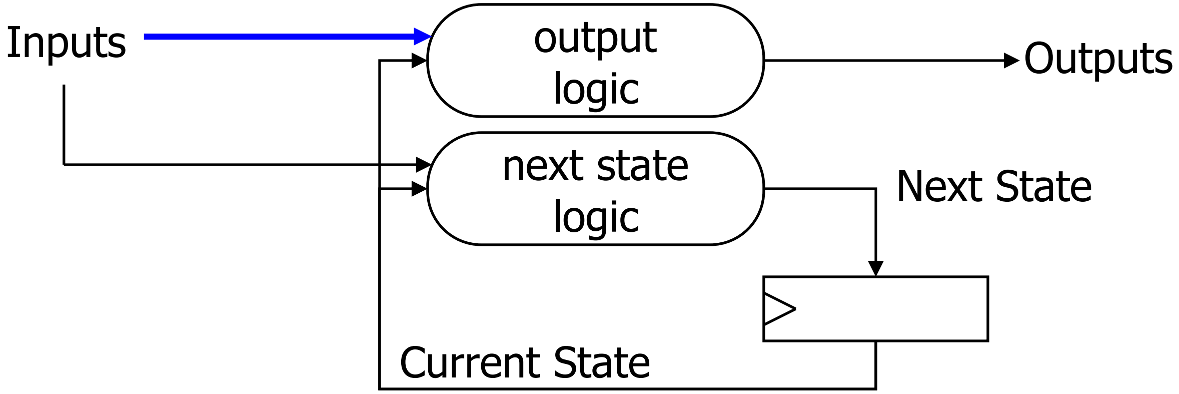
- Values stored in registers represent the state of the circuit
- Combinational logic computes:
  - next state
    - function of current state and inputs
  - outputs
    - values of flip-flops



Here is the big picture of counter- or shift register-based sequential logic systems. The current state and the input will decide the next state by forming a combinational logic in the oval. In the case of counters or registers, the values in the storage elements form the output. What if the state is not exactly the output?

General state machine model

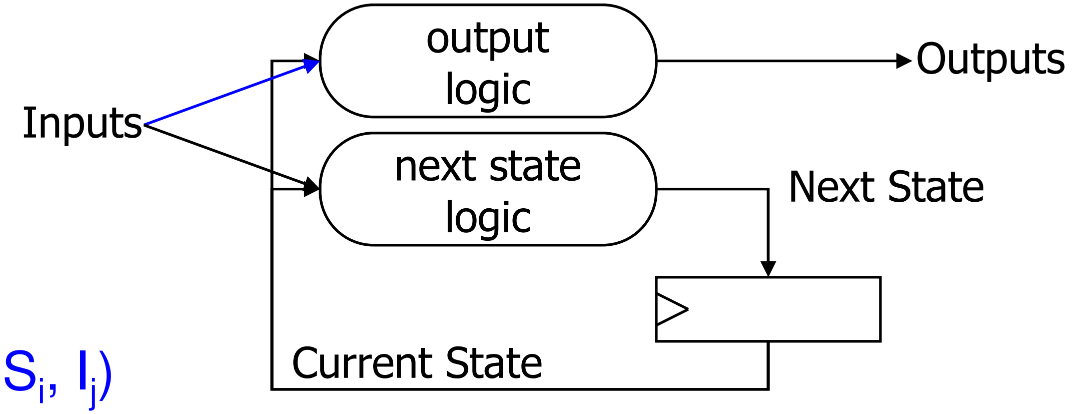
- Values stored in registers represent the state of the circuit
- Combinational logic computes:
  - next state
    - function of current state and inputs
  - outputs
    - function of current state and inputs (Mealy machine)
    - function of current state only (Moore machine)

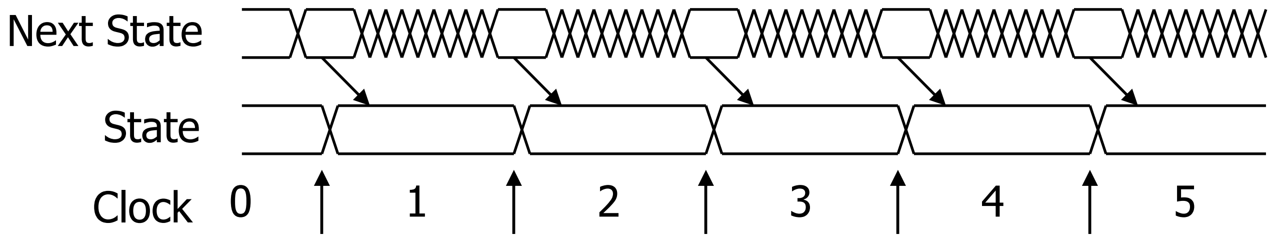


If output is different from the state, there should be one more combinational logic, the upper oval. There is another important classification: depending on the combinational logic for outputs. If outputs are functions of only current state, that model is called a Moore machine. On the other hand, if outputs are also dependent on external inputs, this is called a Mealy machine (drawn by a blue arrow).

State machine model (cont'd)

- States: $S_1, S_2, \ldots, S_k$
- Inputs: $I_1, I_2, \ldots, I_m$
- Outputs: $O_1, O_2, \ldots, O_n$
- Transition function: $F_s(S_i, I_j)$
- Output function: $F_o(S_i)$ or $F_o(S_i, I_j)$





Again, the state transition time is the reference time, which is typically positive- (or negative-) edge of the clock signal depending on FF types. The clock period should be long enough to allow full propagation of input and the current state signals through combinational logic parts.

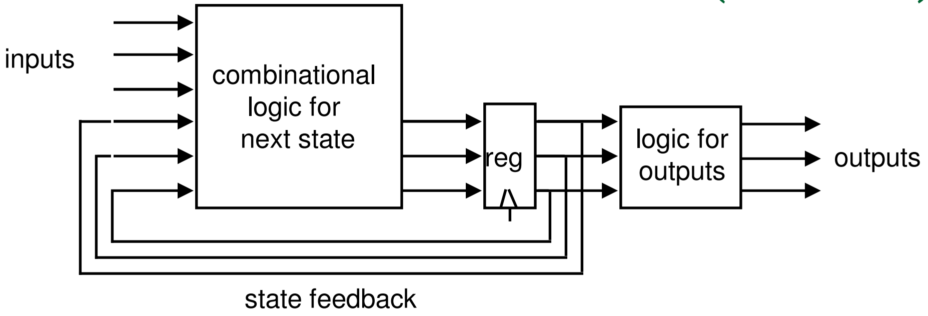
Comparison of Mealy and Moore machines

- Mealy machines tend to have less states
  - different outputs on arcs (n^2) rather than states (n)
- Moore machines are safer to use
  - outputs change at clock edge (always one cycle later)
  - in Mealy machines, input change can cause output change as soon as logic is done – a big problem when two machines are interconnected – asynchronous feedback may occur if one isn’t careful
- Mealy machines react faster to inputs
  - react in same cycle – don't need to wait for clock
  - in Moore machines, more logic may be necessary to decode state into outputs – more gate delays after clock edge

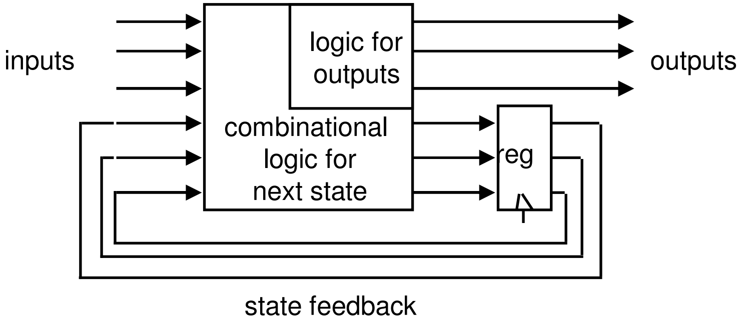
As outputs of a Mealy machine are functions of the external inputs and the present state, the number of states may be less. Information for the next state transition is split between inputs from outside and the state. In Moore machines, outputs are dependent only on the present state, the output will change synchronously if combinational logic has no problem. In Mealy machines, external inputs can change the output anytime with combinational logic delay somewhat independently of the clock. If two machines perform the same function, Mealy machines react faster since inputs are already changing the combinational logic for output.

Comparison of Mealy and Moore machines (cont'd)

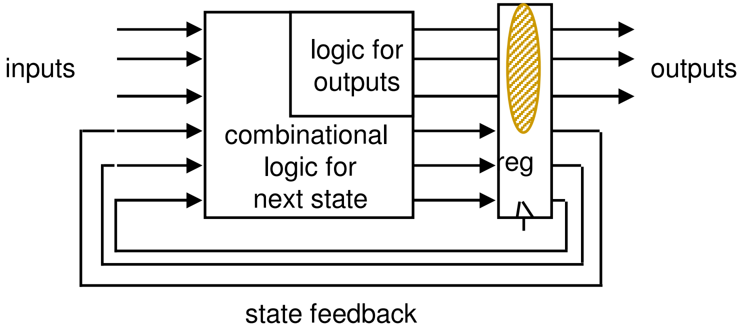
- Moore



- Mealy



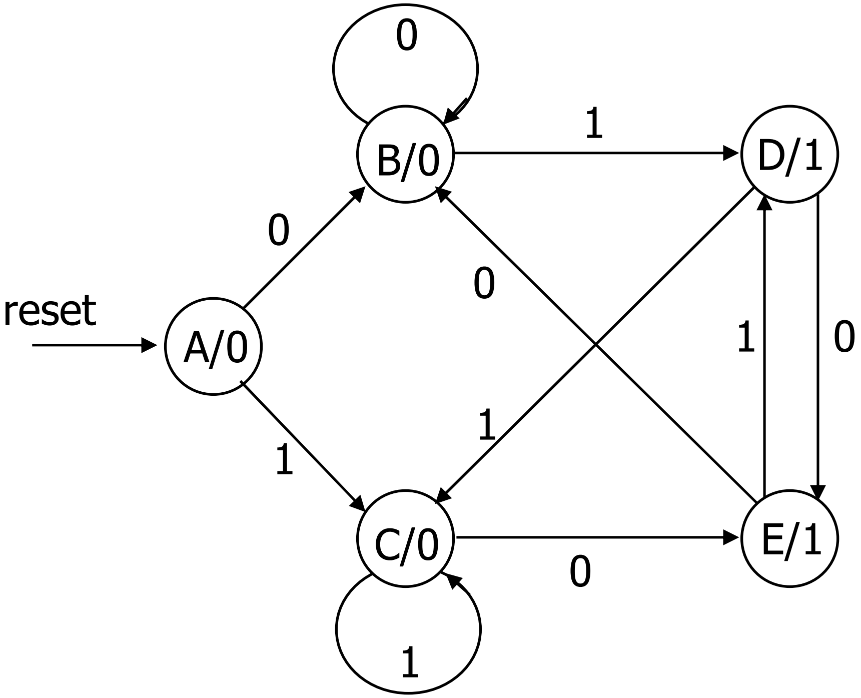
- Synchronous Mealy



This slide illustrates the three types of sequential systems. Synchronous Mealy machines solve the potential glitches and asynchronous change of outputs of Mealy machines by inserting clock-triggered memory elements.

Specifying outputs for a Moore machine

- Output is only function of state
  - specify in state bubble in state diagram
  - example: sequence detector for 01 or 10

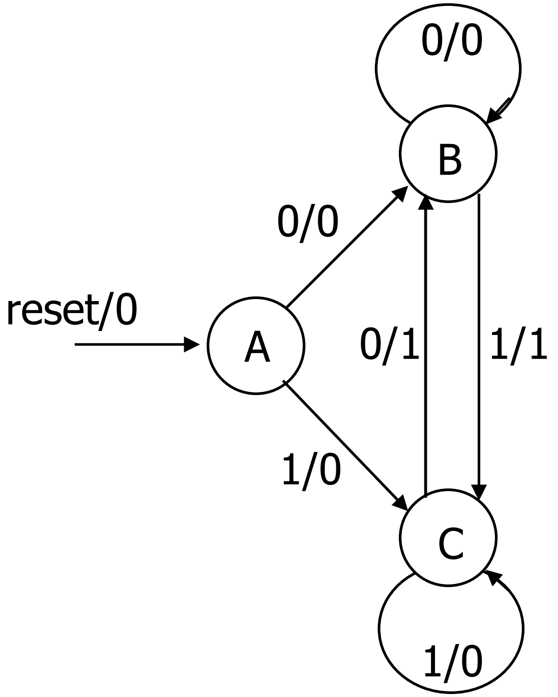


reset	input	current state	next state	output
1	–	–	A	
0	0	A	B	0
0	1	A	C	0
0	0	B	B	0
0	1	B	D	0
0	0	C	E	0
0	1	C	C	0
0	0	D	E	1
0	1	D	C	1
0	0	E	B	1
0	1	E	D	1

Let’s see how a Moore machine can be described. Here, X/Y is the tuple of state X and the output Y. The label in each incoming arc is the input. The output is associated with the current state. Actually, the output signal will be asserted until the system goes to the next state. This Moore machine detects whether the recent input string is 01 or 10.

Specifying outputs for a Mealy machine

- Output is function of state and inputs
  - specify output on transition arc between states
  - example: sequence detector for 01 or 10

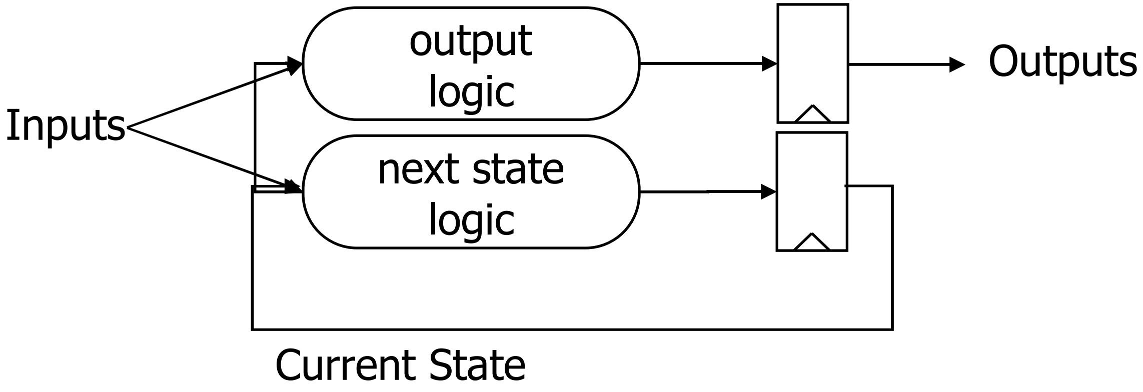


reset	input	current state	next state	output
1	–	–	A	0
0	0	A	B	0
0	1	A	C	0
0	0	B	B	0
0	1	B	C	1
0	0	C	B	1
0	1	C	C	0

In a Mealy machine, both the input and present state determine the next state. X/Y notation in each arrow means input X will generate output Y. Compare the number of states; the Mealy model has only 3 states. The problem of the Mealy machine is that we cannot be sure of the exact timing of output change, not to mention glitch.

Registered Mealy machine (really Moore)

- Synchronous (or registered) Mealy machine
  - registered state AND outputs
  - avoids 'glitchy' outputs
  - easy to implement in PLDs
- Moore machine with no output decoding
  - outputs computed on transition to next state rather than after entering
  - view outputs as expanded state vector

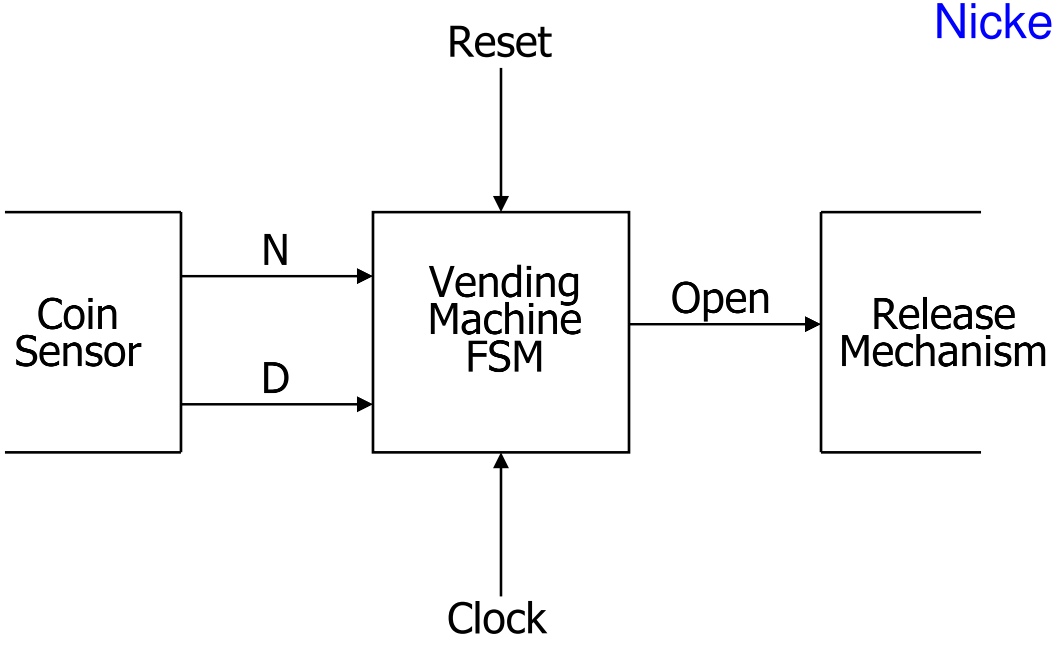


That is why we need a synchronous Mealy machine. Now we can ensure the output will be changed at exact timing. The output will be stored just like state. But it incurs more delay. There are two versions of synchronous Mealy machines. We will look at both versions with the example of a vending machine from now on.

Example: vending machine

- Release item after 15 cents are deposited
- Single coin slot for dimes, nickels
- No change

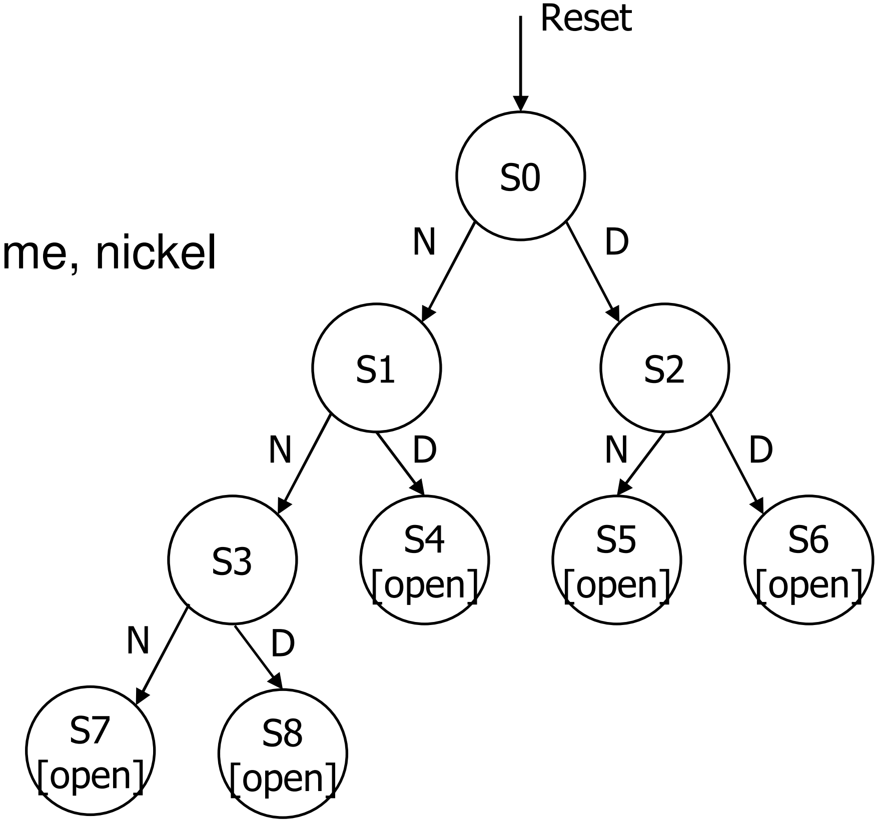
Dime: 10 cent coin
Nickel: 5 cent coin



Now we will see three or four implementations of the same vending machine that sells an item which costs 15 cents. We don’t need to figure out the exact mechanism of identifying dimes and nickels. Just assume that the corresponding wire will be asserted: N for nickel and D for dime. Also, for simplicity, we do not care about change.

Example: vending machine (cont'd)

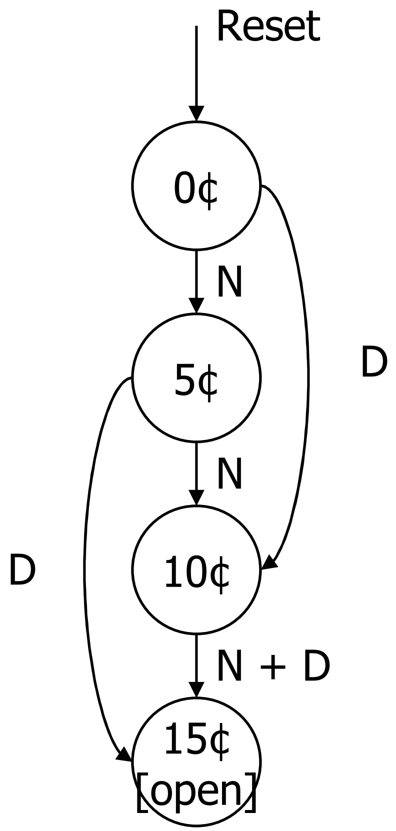
- Suitable abstract representation
  - tabulate typical input sequences:
    - 3 nickels or two dimes
    - nickel, dime or 2nickel, dime, or dime, nickel
  - draw state diagram:
    - inputs: N, D, reset
    - output: open chute
  - assumptions:
    - assume N and D asserted for one cycle
    - each state has a self loop for N = D = 0 (no coin)



First, what would be a state? Probably, the sum of inserted coins is good for a state. Then we should draw a state diagram by listing all the possible scenarios of releasing an item. Once the total amount becomes equal to or greater than 15 cents, the item will be released. Here, we draw all the scenarios of depositing coins, but they should be reflected into the state transition table. In a Moore machine, after the system reaches one of S4..S8, the item will then be released. What about the Mealy machine?

Example: vending machine (cont'd)

- Minimize number of states - reuse states whenever possible



present state	inputs D	N	next state	output open
0¢	0	0	0¢	0
	0	1	5¢	0
	1	0	10¢	0
	1	1	–	–
5¢	0	0	5¢	0
	0	1	10¢	0
	1	0	15¢	0
	1	1	–	–
10¢	0	0	10¢	0
	0	1	15¢	0
	1	0	15¢	0
	1	1	–	–
15¢	–	–	15¢	1

symbolic state table

Here is the simple state transition table for the vending machine. This is kind of a Moore machine since the output becomes 1 after the system moves to state 15¢. First of all, a dime and a nickel cannot be inserted at the same time, which implies don't care terms.

Example: vending machine (cont’d)

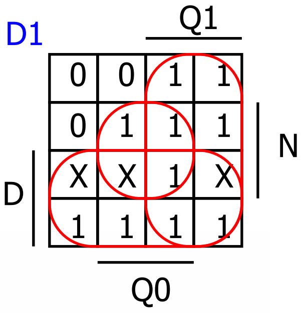
- Uniquely encode states

present state		inputs		next state		output
Q1	Q0	D	N	D1	D0	open
0	0	0	0	0	0	0
		0	1	0	1	0
		1	0	1	0	0
		1	1	–	–	–
0	1	0	0	0	1	0
		0	1	1	0	0
		1	0	1	1	0
		1	1	–	–	–
1	0	0	0	1	0	0
		0	1	1	1	0
		1	0	1	1	0
		1	1	–	–	–
1	1	–	–	1	1	1

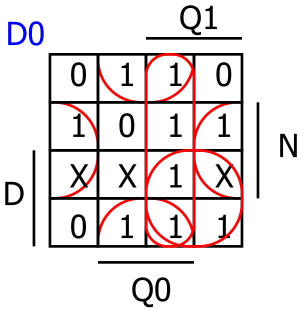
So there are 4 states of the system (0,5,10,15¢), which requires minimum two FFs. The number of bits to represent states can be determined in many ways; we will look at two cases here. Two external inputs and one external output are already explained. This is a simple Moore machine since the output is dependent only on state.

Example: Moore implementation

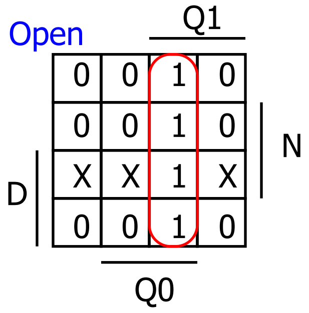
- Mapping to logic



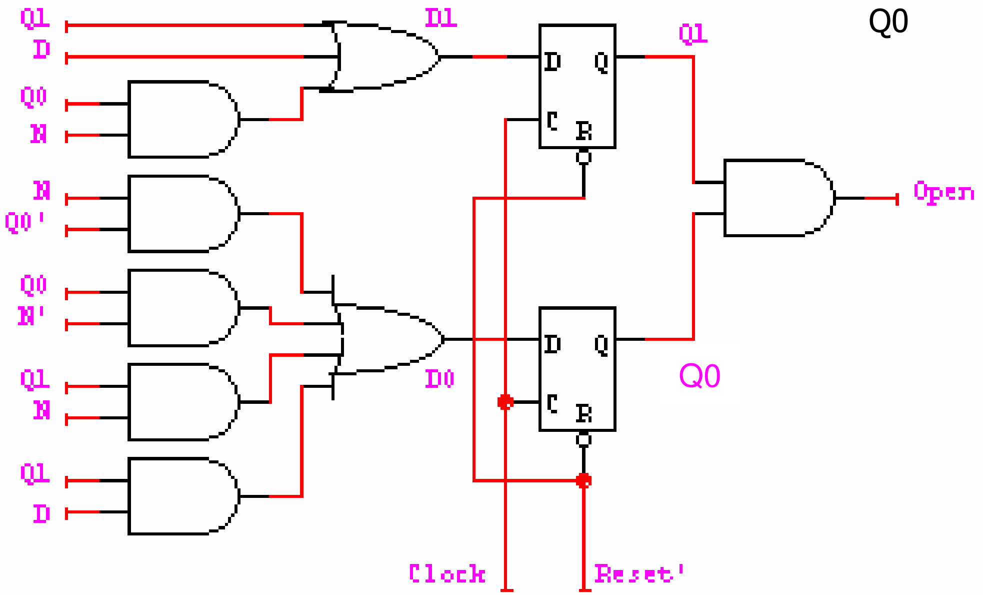
0	0	1	1
0	1	1	1
X	X	1	X
1	1	1	1



0	1	1	0
1	0	1	1
X	X	1	X
0	1	1	1



0	0	1	0
0	0	1	0
X	X	1	X
0	0	1	0



D1 = Q1 + D + Q0 N

D0 = Q0′ N + Q0 N′ + Q1 N + Q1 D

OPEN = Q1 Q0

There are total 4 input variables for each output. OPEN seems to be the simplest logic. In this case, the external output is a function of only state variables. (typo: Q2 -> Q0). For simplicity, we skip the feedback parts of Q1 and Q0 wires.

Example: vending machine (cont'd)

- One-hot encoding

present state				inputs		next state				output
Q3	Q2	Q1	Q0	D	N	D3	D2	D1	D0	open
0	0	0	1	0	0	0	0	0	1	0
				0	1	0	0	1	0	0
				1	0	0	1	0	0	0
				1	1	-	-	-	-	-
0	0	1	0	0	0	0	0	1	0	0
				0	1	0	1	0	0	0
				1	0	1	0	0	0	0
				1	1	-	-	-	-	-
0	1	0	0	0	0	0	1	0	0	0
				0	1	1	0	0	0	0
				1	0	1	0	0	0	0
				1	1	-	-	-	-	-
1	0	0	0	-	-	1	0	0	0	1

$D0 = Q0\ D'\ N'$

$D1 = Q0\ N + Q1\ D'\ N'$

$D2 = Q0\ D + Q1\ N + Q2\ D'\ N'$

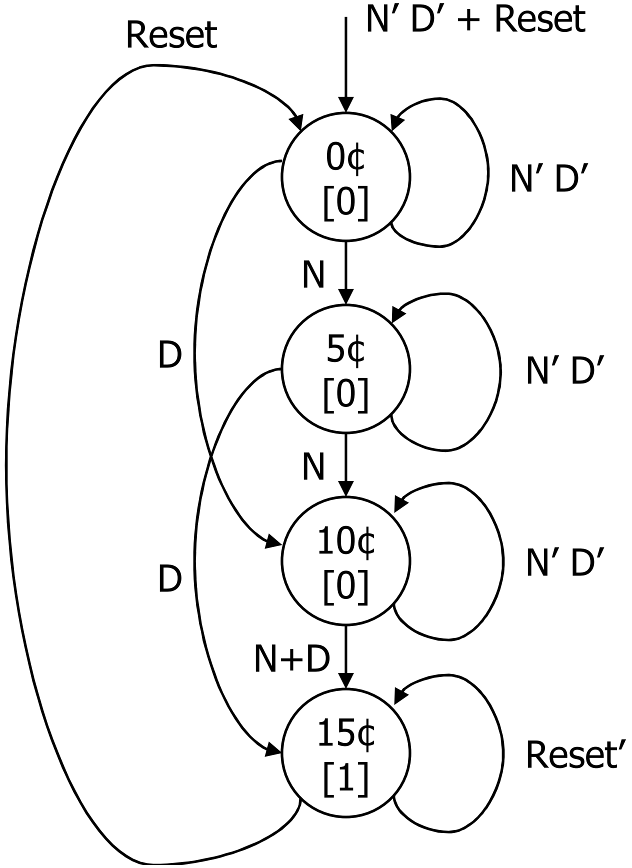
$D3 = Q1\ D + Q2\ D + Q2\ N + Q3$

$OPEN = Q3$

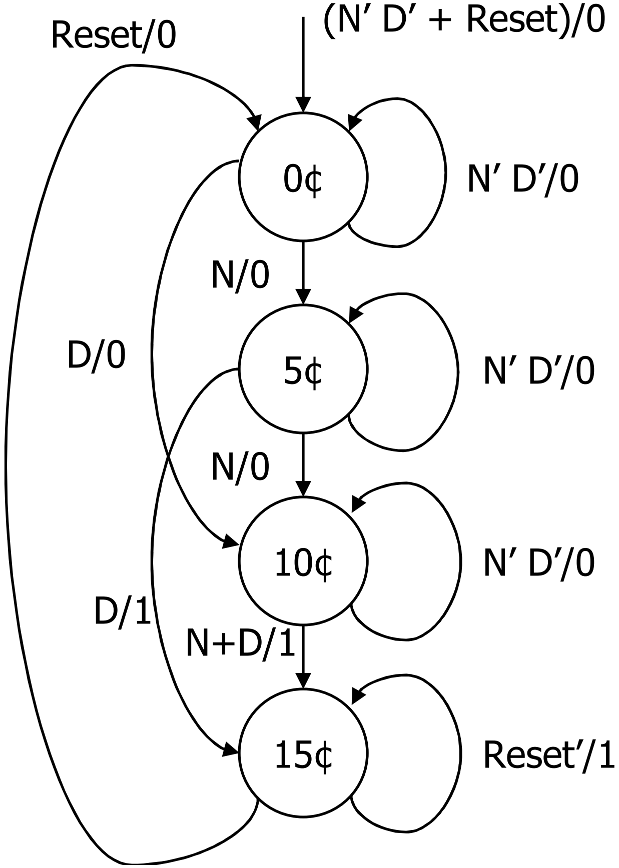
One-hot encoding means one bit is used for each state; that is, only one state variable is set, or "hot," for each state. So the number of bits to represent states is the same as the number of states. The benefit is that the next state generation function may be simple since the number of product terms for each output is typically small. In this case, we use 4 bits or FFs.

Equivalent Mealy and Moore state diagrams

- Moore machine
  - outputs associated with state

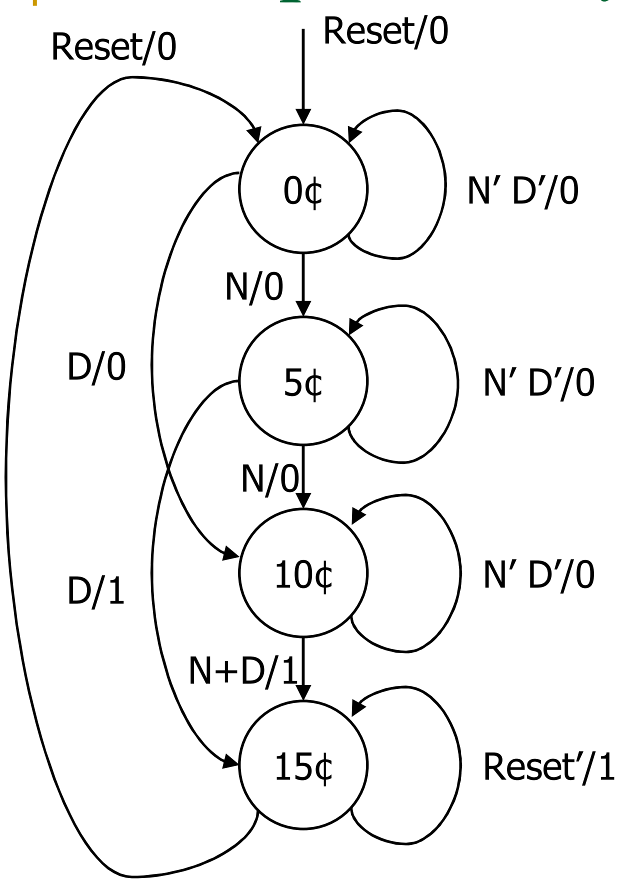


- Mealy machine
  - outputs associated with transitions

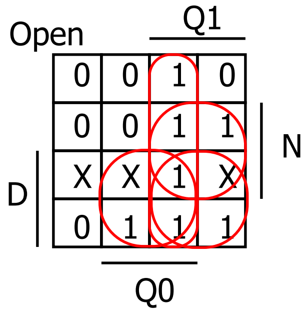


This slide shows the complete state transition diagram of the vending machine in two versions. In the Moore model, the number in [ ] is the output. Whereas, in the Mealy model, the output is associated with each arc.

Example: Mealy implementation



present state		inputs		next state		output
Q1	Q0	D	N	D1	D0	open
0	0	0	0	0	0	0
		0	1	0	1	0
		1	0	1	0	0
		1	1	–	–	–
0	1	0	0	0	1	0
		0	1	1	0	0
		1	0	1	1	1
		1	1	–	–	–
1	0	0	0	1	0	0
		0	1	1	1	1
		1	0	1	1	1
		1	1	–	–	–
1	1	–	–	1	1	1



0	0	1	0
0	0	1	1
X	X	1	X
0	1	1	1

D0 = Q0’N + Q0N’ + Q1N + Q1D

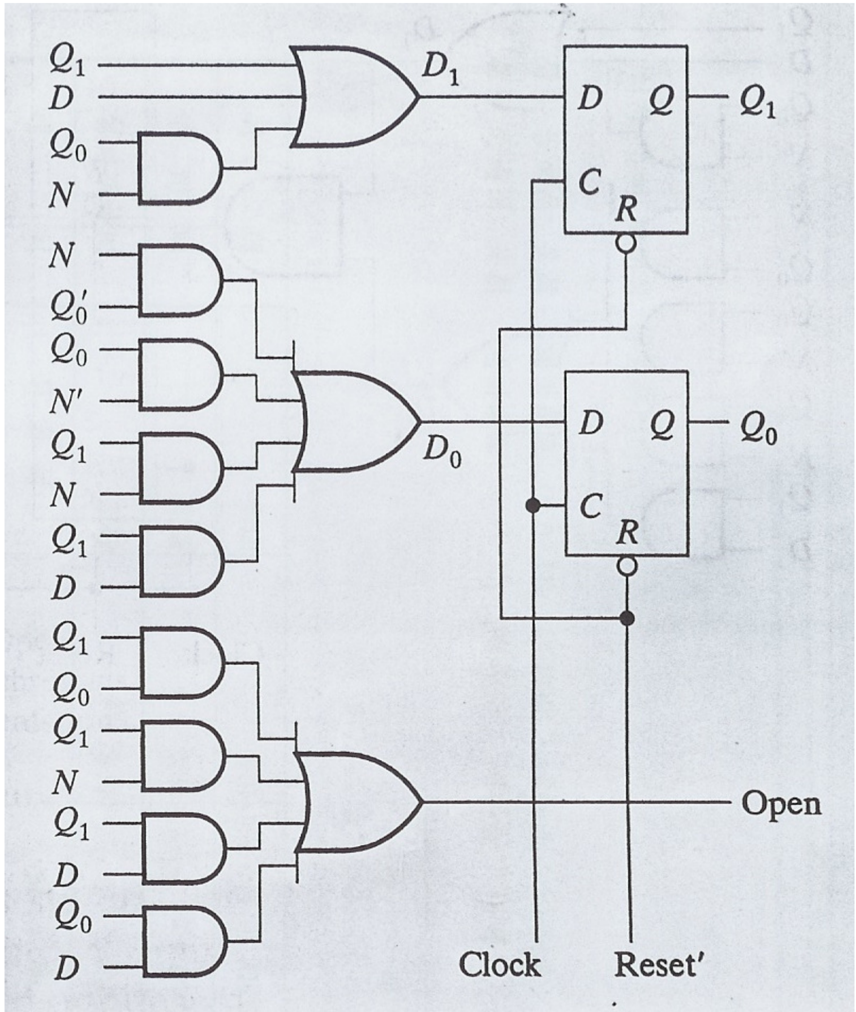
D1 = Q1 + D + Q0N

OPEN = Q1Q0 + Q1N + Q1D + Q0D

In the case of a Mealy machine, the output, OPEN, is a function of state and the inputs.

Example: Mealy implementation

D0 = Q0’N + Q0N’ + Q1N + Q1D
D1 = Q1 + D + Q0N
OPEN = Q1Q0 + Q1N + Q1D + Q0D

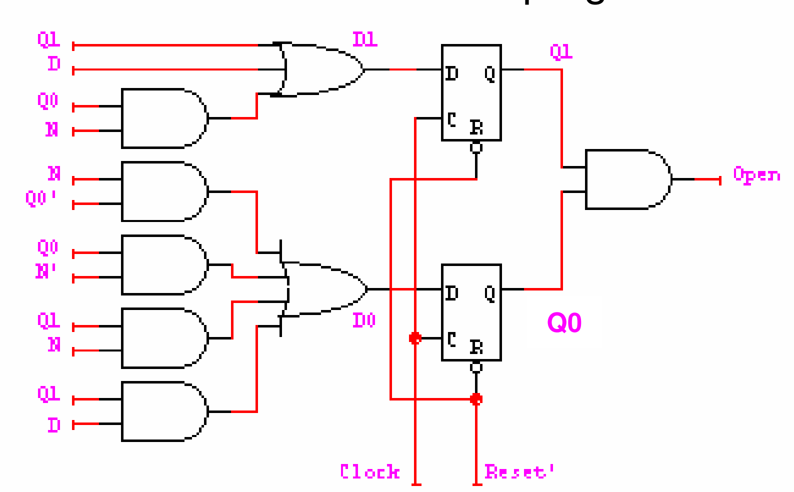


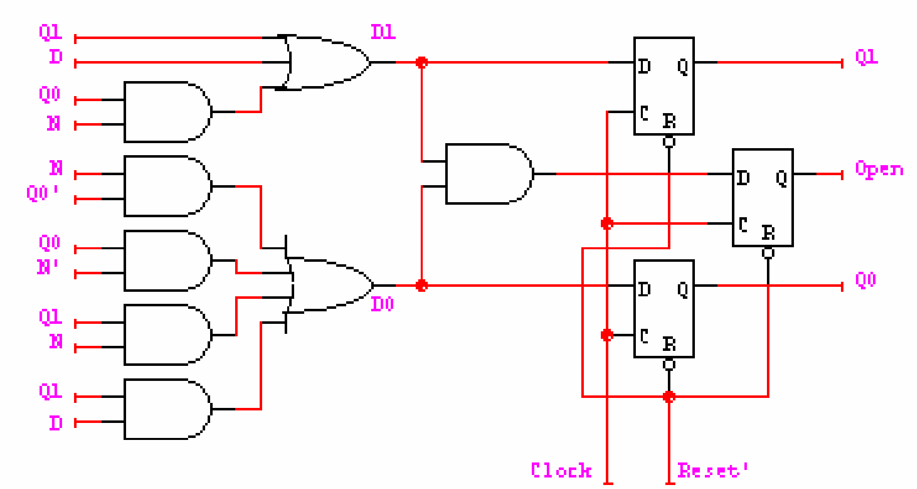
Here is the overall implementation of the vending machine based on the Mealy model.

Vending machine: Moore to synch. Mealy

- OPEN = Q1Q0 creates a combinational delay after Q1 and Q0 change in Moore implementation
- This can be corrected by retiming, i.e., move flip-flops and logic through each other to improve delay
- OPEN.d = Q1.d and Q2.d

  = (Q1 + D + Q0N)(Q0'N + Q0N' + Q1N + Q1D)
- Implementation now looks like a synchronous Mealy machine
  - it is common for programmable devices to have FF at end of logic

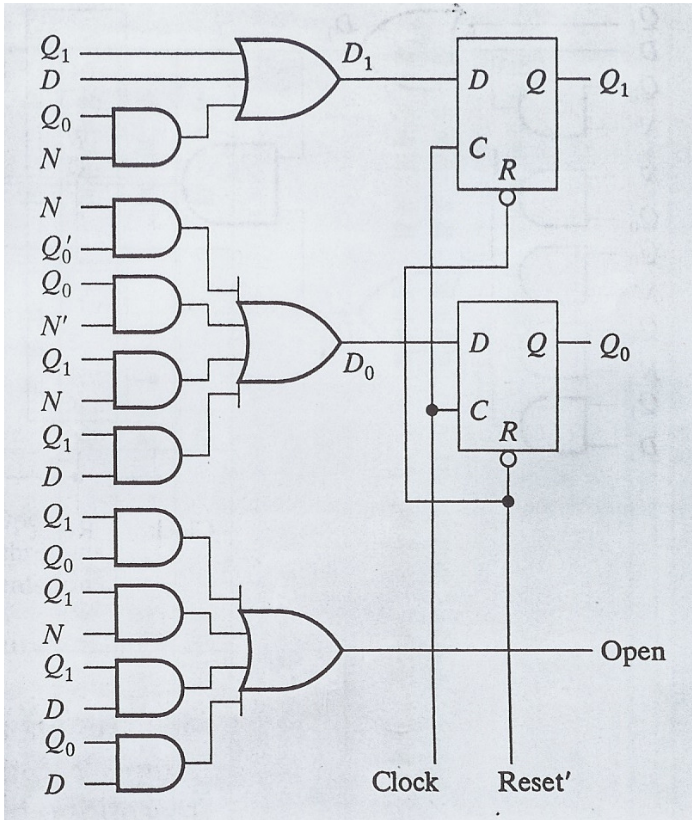


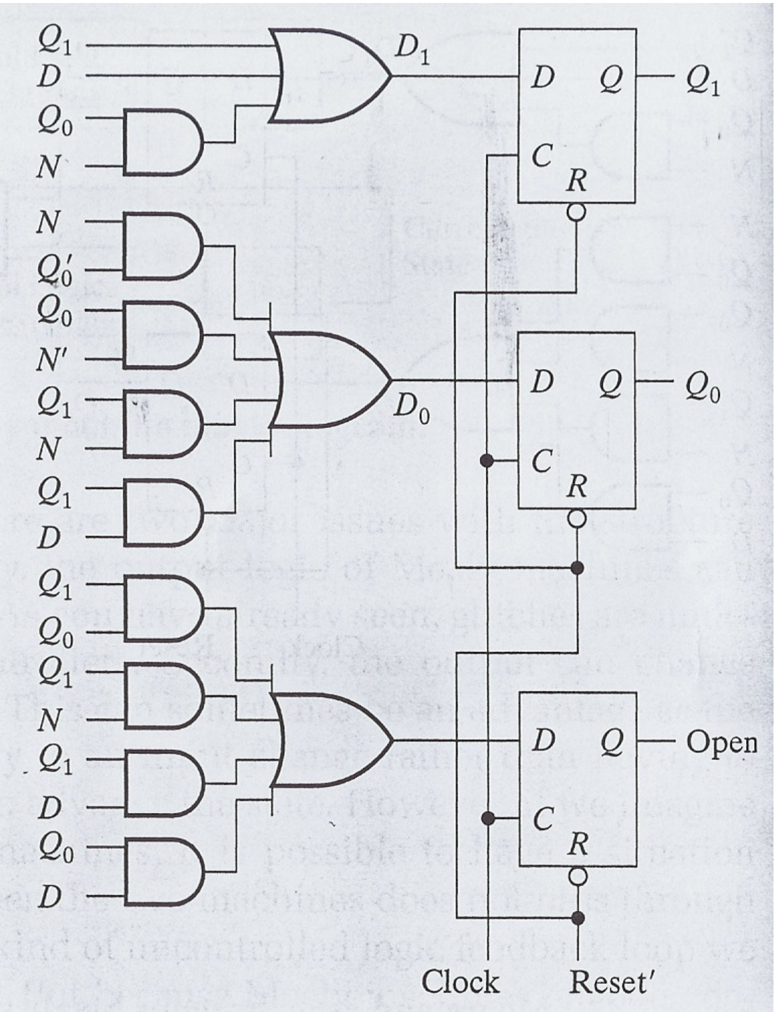


If we want to realize a synch Mealy model, the 1st option is to derive its implementation from the Moore model. Reduce the delay by shifting the logic for OPEN inside; now, the D input of the FF for OPEN will be D1D0. So when the clock cycle is finished for the next state 15¢, the stored TRUE value for OPEN will be propagated at the next state.

Vending machine: Mealy to synch. Mealy

- OPEN.d = Q1Q0 + Q1N + Q1D + Q0D

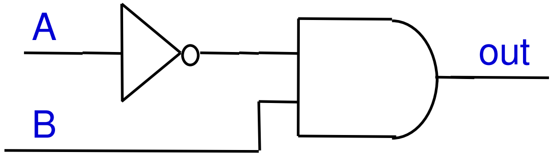


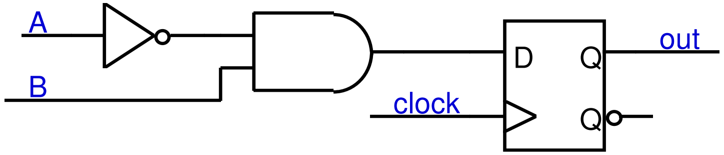


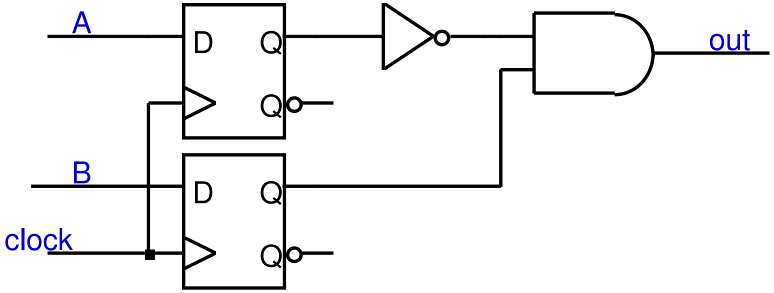
The other option is to derive the synch Mealy version from the original Mealy model-based implementation. We just add one FF at the OPEN output.

Mealy and Moore examples

- Recognize A,B = 0,1
  - Mealy or Moore?

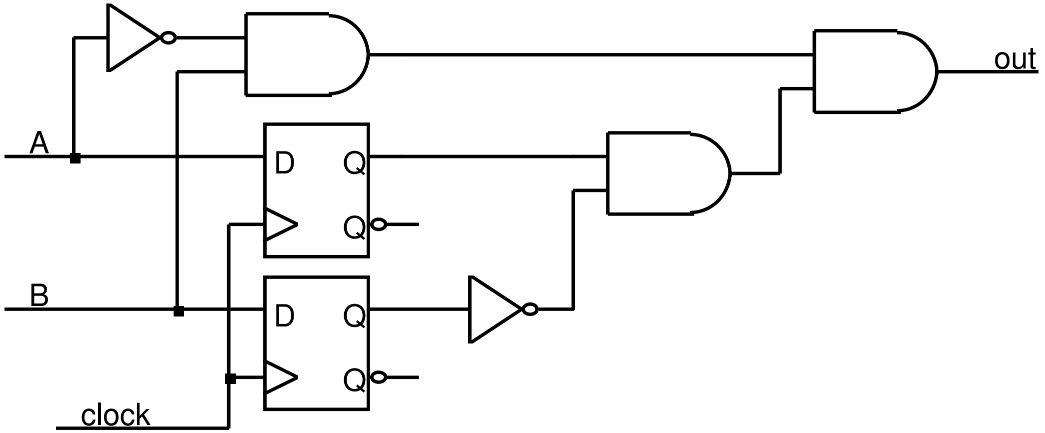


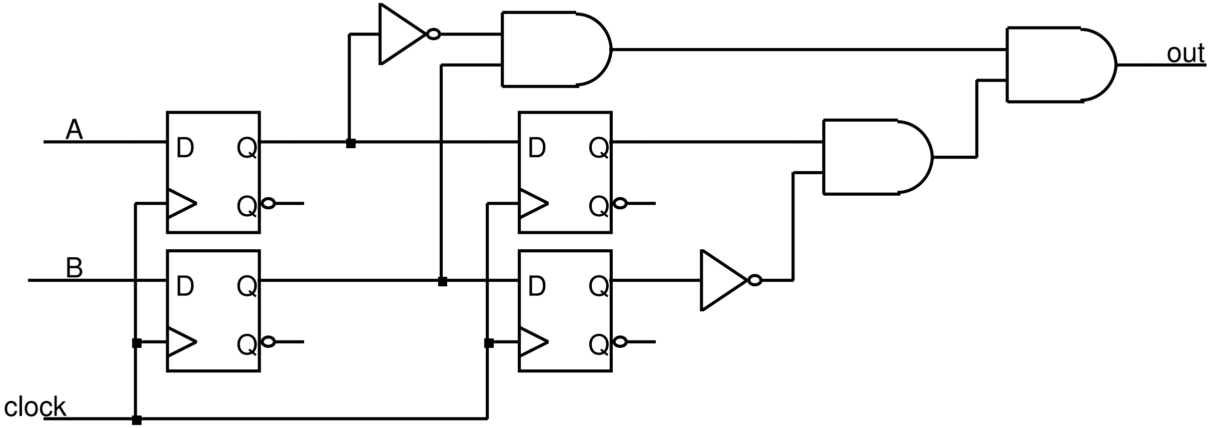




Mealy and Moore examples (cont'd)

- Recognize A,B = 1,0 then 0,1
    - Mealy or Moore?





How about the lower design? Is it a Mealy machine? It tries to remove the unpredictable update of the output. In general, Mealy machines are not so predictable, so their synchronous versions or Moore machines are preferred.

Hardware Description Languages and Sequential Logic

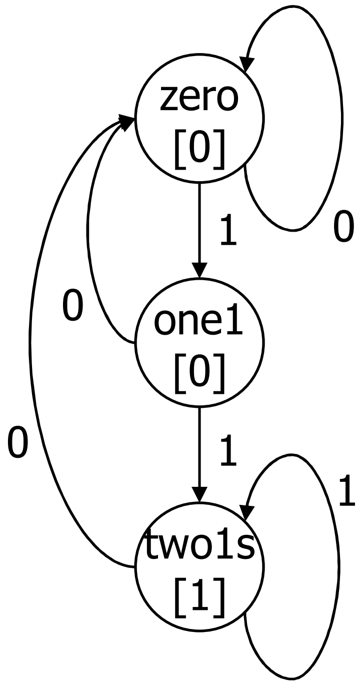
- Flip-flops
  - representation of clocks - timing of state changes
  - asynchronous vs. synchronous
- FSMs
  - structural view (FFs separate from combinational logic)
  - behavioral view (synthesis of sequencers – not in this course)
- Data-paths = data computation (e.g., ALUs, comparators) + registers
  - use of arithmetic/logical operators
  - control of storage elements

Let's see how we can write Verilog programs for FSMs.

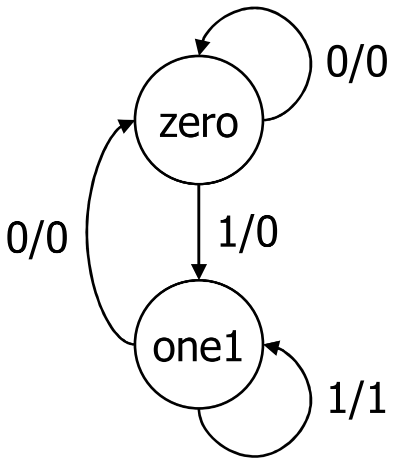
Example: reduce-1-string-by-1

- Remove one 1 from every string of 1s on the input

Moore



Mealy



We will look at a simple example; input is a binary string. When there is a string of consecutive 1s, we will replace only the first 1 by 0. Suppose a string starts like 011011100, the output would be 001001100. Recall that the number of states of a Mealy machine can be smaller than that of a Moore machine for the same function.

Verilog FSM - Reduce 1s example

- Moore machine

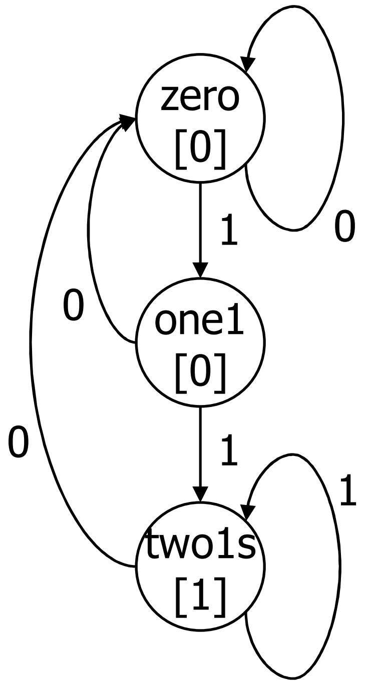
```
module reduce (clk, reset, in, out);
  input clk, reset, in;
  output out;

  parameter zero  = 2'b00;
  parameter one1  = 2'b01;
  parameter two1s = 2'b10;

  reg out;
  reg [2:1] state;       // state variables
  reg [2:1] next_state;

  always @(posedge clk)
    if (reset) state = zero;
    else       state = next_state;
```

state assignment (easy to change, if in one place)



Let's start with a Moore machine. First we declare some constants to differentiate three states. The index of the register can be [1:0]; we need two bits for three states.

Moore Verilog FSM (cont'd)

```
always @(in or state)

  case (state)
    zero:  // last input was a zero
       begin
            if (in) next_state = one1;
            else    next_state = zero;
       end
    one1:  // we've seen one 1
       begin
            if (in) next_state = two1s;
            else    next_state = zero;
       end
    two1s: // we've seen at least 2 ones
       begin
            if (in) next_state = two1s;
            else    next_state = zero;
       end
  endcase
```

crucial to include
all signals that are
input to state determination

```
always @(state)
  case (state)
       zero: out = 0;
       one1: out = 0;
       two1s: out = 1;
  endcase

endmodule
```

note that output
depends only on state

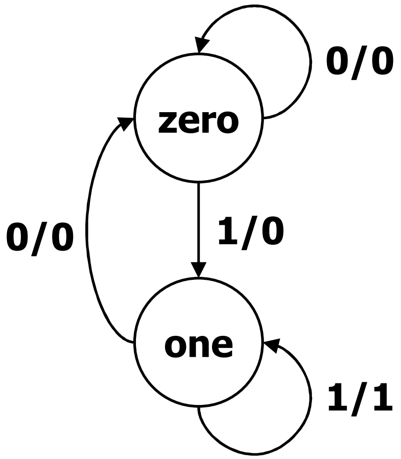
The upper always block is nothing but a description of state diagram. Depending on the current state and the input (In), the next state will be determined. In the lower block, the output is associated with only state, which proves this is a Moore model.

Mealy Verilog FSM

```
module reduce (clk, reset, in, out);
  input clk, reset, in;
  output out;
  reg out;
  reg state; // state variables
  reg next_state;

  always @(posedge clk)
    if (reset) state = zero;
    else       state = next_state;

  always @(in or state)
    case (state)
      zero:  // last input was a zero
         begin
             out = 0;
             if (in) next_state = one;
             else    next_state = zero;
         end
      one:   // we've seen one 1
         if (in) begin
             next_state = one;
             out = 1;
         end else begin
             next_state = zero;
             out = 0;
         end
    endcase
endmodule
```



This is a Mealy model; so, the output is associated with the input (and the state as well). In each state, input decides the output.

Synchronous Mealy Machine

```
module reduce (clk, reset, in, out);
  input clk, reset, in;
  output out;
  reg out;
  reg state; // state variables

  always @(posedge clk)
    if (reset) state = zero;
    else
     case (state)
         zero:    // last input was a zero
             begin
                 out = 0;
                 if (in) state = one;
                 else    state = zero;
             end
         one:     // we've seen one 1
             if (in) begin
                 state = one; out = 1;
             end else begin
                 state = zero; out = 0;
             end
    endcase
endmodule
```

This is a synchronous Mealy model; again, the output is associated with the input (and the state as well). In this case, we don’t need the variable next_state since the output is changing synchronously with the state.

Finite state machines summary

- Models for representing sequential circuits
  - abstraction of sequential elements
  - finite state machines and their state diagrams
  - inputs/outputs
  - Mealy, Moore, and synchronous Mealy machines
- Finite state machine design procedure
  - deriving state diagram
  - deriving state transition table
  - determining next state and output functions
  - implementing combinational logic
- Hardware description languages

We start with simple FSMs like counters and shift registers, where states are outputs directly. We should differentiate Moore and Mealy models. With either model, we should be able to design a FSM.