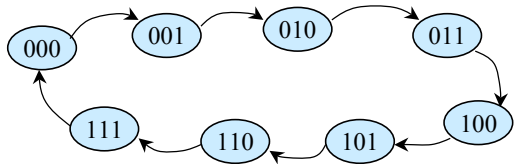


## Finite State Machines

- Flip-flops store more than just register values for data operations. They can also record **state** information.
- Counters are an example of this. Each clock cycle causes the flip-flops to record a different value → i.e. proceeding from state to state.



## Components Of An FSM

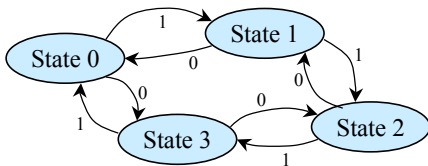
- To process a sequence of inputs instead of a single snapshot, a **finite state machine (FSM)** is needed.
  - **state**: current stage in the sequence.
  - **finite**: number of states is limited and known.
  - **machine**: it's a machine.
- Finite state machines are composed of three parts:
  - a set of states
  - a set of transitions between states
  - a set of actions, associated with each transition
    - in general FSMs, actions can be associated with entering, exiting or remaining in a certain state.
- Also known as a **finite state automaton**.

## State Diagrams

- Transitions between states can be represented in table format:

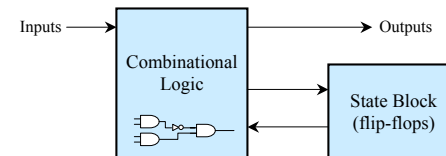
	State 0	State 1	State 2	State 3
Input 0	State 3	State 0	State 1	State 2
Input 1	State 1	State 2	State 3	State 0

- State diagrams are the more common representation:



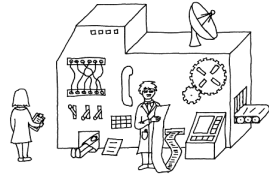
## Designing An FSM

- Actual circuit will be composed of two blocks:
  - **combinational logic block**: takes in the inputs and the current state, and decides what the outputs and the next state will be.
  - **state block**: the flip-flops that record the current state.
- Everything is assumed to be synchronous, and the flip-flops are generally assumed to be D-type.



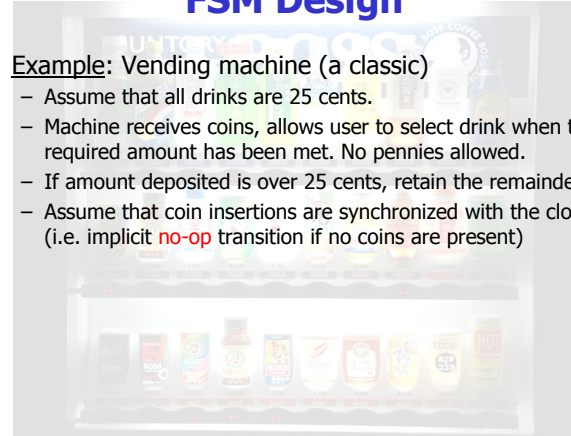
## FSM Design Steps

- Finite state machines in five easy steps:
  1. Make state diagram or state table for system.
  2. Select number of flip-flops to store states.
  3. Assign flip-flop values for each state (**state assignment**).
  4. Develop a truth table for the inputs and current state.
    - outputs would be both system outputs and next state.
  5. Implement the combinational logic using gate design techniques.



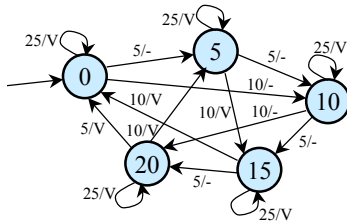
## FSM Design

- Example: Vending machine (a classic)
  - Assume that all drinks are 25 cents.
  - Machine receives coins, allows user to select drink when the required amount has been met. No pennies allowed.
  - If amount deposited is over 25 cents, retain the remainder.
  - Assume that coin insertions are synchronized with the clock (i.e. implicit **no-op** transition if no coins are present)



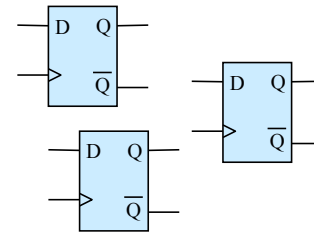
## FSM Design

- Step 1: Design appropriate state machine
  - **states**: amount of money deposited (0, 5, 10, 15, 20).
  - **transitions**: coin being inserted (5, 10, 25).
  - **output**: "V" (vend drink) or "-" (no vend).



## FSM Design

- Step #2: Allocate flip-flops
  - Since there are 5 states, use 3 flip-flops
  - General rule is  $\lceil \log_2 n \rceil$  flip-flops for  $n$  states.



## FSM Design

- **Step 3: State assignment**

- Call flip-flops  $F_0$ ,  $F_1$  &  $F_2$ .

Current State	$F_2$	$F_1$	$F_0$
0 cents	0	0	0
5 cents	0	0	1
10 cents	0	1	1
15 cents	0	1	0
20 cents	1	1	0

- **Note:** state assignment is completely arbitrary, so try to minimize flip-flop transitions between states.

## FSM Design

- **Step 4: Truth table**

- Inputs to truth table:
  - coins (nickel, dime, quarter)
  - flip-flops ( $F_0$ ,  $F_1$ ,  $F_2$ )
- Outputs from truth table:
  - vend signal
  - flip-flops ( $F_0$ ,  $F_1$ ,  $F_2$ )
- $2^6 = 64$  rows in table!
- Since many inputs and/or states will never occur, table can be reduced somewhat, or these don't care conditions can be handled intelligently
  - logical reduction rules ☺

N	D	Q	$F_2$	$F_1$	$F_0$	V	$F_2'$	$F_1'$	$F_0'$
0	0	0	0	0	0	X	X	X	X
0	0	0	0	0	1	X	X	X	X
0	0	0	0	1	0	X	X	X	X
0	0	0	0	1	1	X	X	X	X
0	0	0	1	0	0	X	X	X	X
0	0	0	1	0	1	X	X	X	X
0	0	0	1	1	0	X	X	X	X
0	0	0	1	1	1	X	X	X	X
0	0	1	0	0	0	1	0	0	0
0	0	1	0	0	1	1	0	0	1
0	0	1	0	1	0	1	1	0	1
0	0	1	0	1	1	1	0	1	1
0	0	1	1	0	0	X	X	X	X
...	...	...	...	...	...	...	...	...	...
1	1	1	1	1	0	X	X	X	X
1	1	1	1	1	1	X	X	X	X

## FSM Design

- **Step 4, continued**

- all conditions not shown here are assumed to be "don't care" conditions.
- easier to fill out Karnaugh map if you only have to look for certain combinations of inputs.

N	D	Q	$F_2$	$F_1$	$F_0$	V	$F_2'$	$F_1'$	$F_0'$
0	0	1	0	0	0	1	0	0	0
0	0	1	0	0	1	1	0	0	1
0	0	1	0	1	0	1	0	1	0
0	0	1	0	1	1	1	0	1	1
0	0	1	1	1	0	1	1	1	0
0	1	0	0	0	0	0	0	1	1
0	1	0	0	0	1	0	0	1	0
0	1	0	0	1	0	1	0	0	0
0	1	0	0	1	1	0	1	1	0
0	1	0	1	1	0	1	0	0	1
1	0	0	0	0	0	0	0	0	1
1	0	0	0	0	1	0	0	1	1
1	0	0	0	1	0	0	1	1	0
1	0	0	0	1	1	0	0	1	0
1	0	0	1	1	0	1	0	0	0

## FSM Design

- **Step 5: Logic gate design (for Vend)**

NDQ	$F_2F_1F_0$							
	000	001	011	010	110	111	101	100
000	X	X	X	X	X	X	X	X
001	1	1	1	1	1	X	X	X
011	X	X	X	X	X	X	X	X
010	0	0	0	1	1	X	X	X
110	X	X	X	X	X	X	X	X
111	X	X	X	X	X	X	X	X
101	X	X	X	X	X	X	X	X
100	0	0	0	0	1	X	X	X

## FSM Design

- Step 5: Logic gate design (for  $F_2'$ )

NDQ \ $F_2F_1F_0$	000	001	011	010	110	111	101	100
000	X	X	X	X	X	X	X	X
001	0	0	0	0	1	X	X	X
011	X	X	X	X	X	X	X	X
010	0	0	1	0	0	X	X	X
110	X	X	X	X	X	X	X	X
111	X	X	X	X	X	X	X	X
101	X	X	X	X	X	X	X	X
100	0	0	0	1	0	X	X	X

## FSM Design

- Step 5: Logic gate design (for  $F_1'$ )

NDQ \ $F_2F_1F_0$	000	001	011	010	110	111	101	100
000	X	X	X	X	X	X	X	X
001	0	0	1	1	1	X	X	X
011	X	X	X	X	X	X	X	X
010	1	1	1	0	0	X	X	X
110	X	X	X	X	X	X	X	X
111	X	X	X	X	X	X	X	X
101	X	X	X	X	X	X	X	X
100	0	1	1	1	0	X	X	X

## FSM Design

- Step 5: Logic gate design (for  $F_0'$ )

Why is this row a potential problem?

NDQ \ $F_2F_1F_0$	000	001	011	010	110	111	101	100
000	X	X	X	X	X	X	X	X
001	0	1	1	0	0	X	X	X
011	X	X	X	X	X	X	X	X
010	1	0	0	0	1	X	X	X
110	X	X	X	X	X	X	X	X
111	X	X	X	X	X	X	X	X
101	X	X	X	X	X	X	X	X
100	1	1	0	0	0	X	X	X

## Transition Problems

- When two flip-flops have to change their values at the same time, it introduces a possibility of a race condition.
  - when the FSM's state will change from 110 to 000, it will have to pass through either 010 or 100 first, for an instant.
  - if the output is supposed to remain at 1, passing through state 010 will cause the output to pulse at 0 briefly, which could produce unwanted behaviour elsewhere.
- To remedy this, try to do the following:
  - ensure that state transitions only involve changing a single bit
  - whenever a state transition changes multiple bits, arrange it so that intermediate states are not part of the FSM already
  - when all else fails, create multiple destination states.