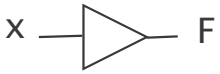
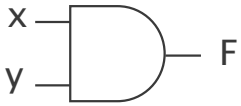
# Basic logic gates



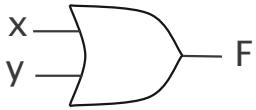

| x | F |
|---|---|
| 0 | 0 |
| 1 | 1 |

F = x
Driver



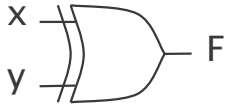

| x | y | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

F = x y
AND



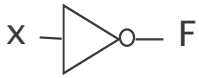

| x | y | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

F = x + y
OR



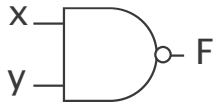

| x | y | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

F = x ⊕ y
XOR



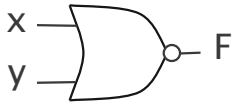

| x | F |
|---|---|
| 0 | 1 |
| 1 | 0 |

F = x'
Inverter



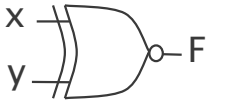

| x | y | F |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

F = (x y)'
NAND




| x | y | F |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

F = (x+y)'
NOR




| x | y | F |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

F = x ⊙ y
XNOR

# Combinational logic design

A) Problem description

y is 1 if a is to 1, or b and c are 1.
z is 1 if b or c is to 1, but not both, or if all are 1.

B) Truth table

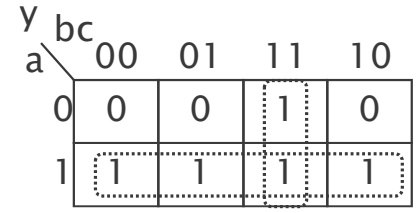
| Inputs | | | Outputs | |
|---|---|---|---|---|
| a | b | c | y | z |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

C) Output equations

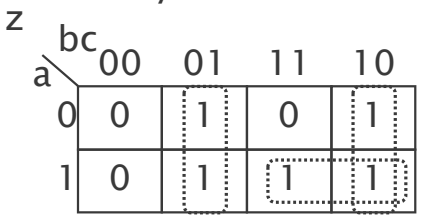
y = a'bc + ab'c' + ab'c + abc' + abc

z = a'b'c + a'bc' + ab'c + abc' + abc

D) Minimized output equations

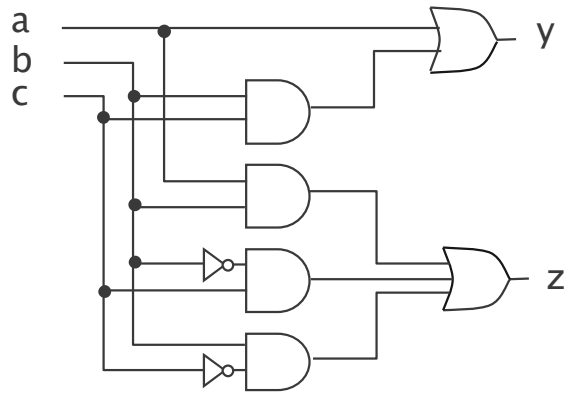



y = a + bc




z = ab + b'c + bc'

E) Logic Gates



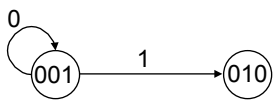
# Finite State Machines

- Finite State Machines (FSMs)
  - general models for representing sequential circuits
  - two principal types based on output behavior (Moore and Mealy)
- Basic sequential circuits revisited and cast as FSMs
  - shift registers
  - counters
- Design procedure for FSMs
  - state diagrams
  - state transition table
  - next state functions
  - potential optimizations
- Hardware description languages



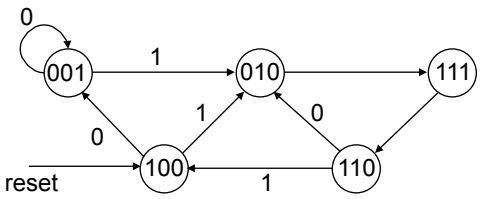
# Finite state machine

- A set of States – the FSM is in one state at any time
- Inputs – inputs used by the FSM
- Next state function – Determines how the FSM moves from one state to another based on the state and the inputs
- Output function – Compute the output based on current state (and possibly the inputs)
- The FSM transitions from one state to another as determined by the next state function function



# Example finite state machine diagram

- 5 states
- 8 other transitions between states
  - 6 conditioned by input
  - 1 self-transition (on 0 from 001 to 001)
  - 2 independent of input (to/from 111)
- 1 reset transition (from all states) to state 100
  - represents 5 transitions (from each state to 100), one a self-arc
  - simplifies condition on other transitions –all would include **AND reset'** )
  - short-hand – rather than drawing a transition arc from each state



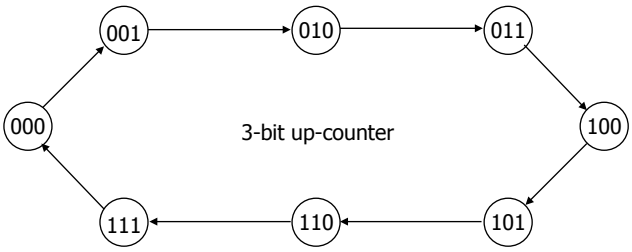



# State diagrams

- Like a program
- Start in some state
- For each state:
  - For all possible input combinations
    - Determine what the next state should be
    - Determine what the output should be
- States are used to remember what happened in the past
  - Typically, being in a state means something, e.g.
    - We've seen an even number of 1's
    - Button A has been pressed
    - We are waiting until the input goes back low
    - We've counted up to 5

# Counters are simple finite state machines

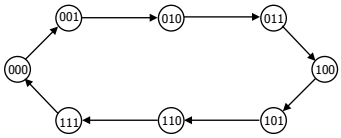
- Counters
  - proceed through well-defined sequence of states
- Many types of counters: binary, BCD, Gray-code, etc.…
  - 3-bit up-counter: 000, 001, 010, 011, 100, 101, 110, 111, 000, ...
  - 3-bit down-counter:  111, 110, 101, 100, 011, 010, 001, 000, 111, ...

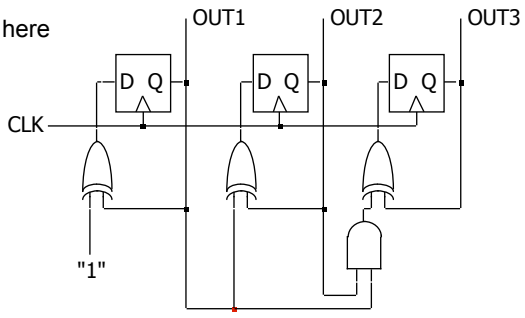





# How do we turn a state diagram into logic?

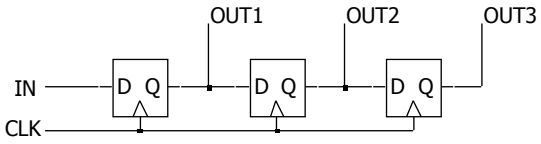
- Counter
  - 3 flip-flops to hold state
    - clock signal controls when flip-flop memory changes
      - move to next state with clock ticks
      - wait long enough for combinational logic to compute new value
  - Logic to compute next state – just an increment function
  - Logic to compute output
    - Just the flip-flop outputs here

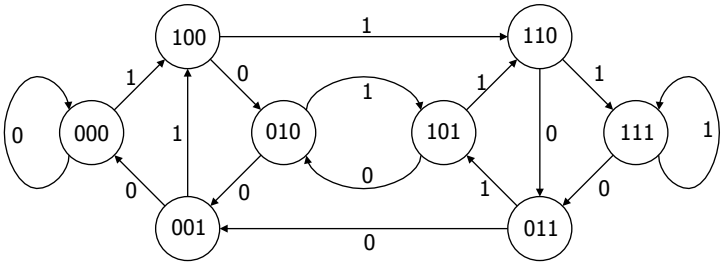


# Any sequential system be represented with a state diagram

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

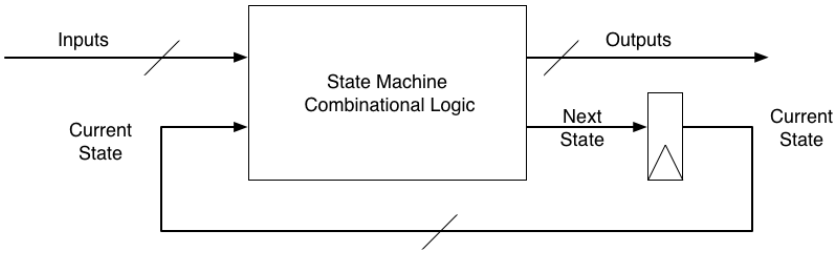





# General Finite State Machine Implementation

- The state register holds the current state of the machine
  - Similar to a program counter
  - Different value for each state
- The state machine logic computes:
  - The next state function – where the FSM should transition next
  - The output function
    - Function of the current state (Moore)
    - Function of the current state and the inputs (Mealy)



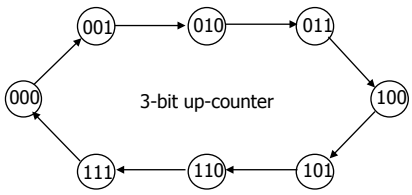
# FSM design procedure

- Draw the state diagram in all its glory (creative design)
  - List all inputs
  - List all outputs
  - Draw all the states
  - Draw all possible transitions from each state
    - One for each input combination
    - Use don't cares to reduce number
- Decide how each state should be represented using state bits
  - Choice may determine cost/speed of FSM implementation
- Convert state diagram to a state transition table (turn crank)
  - Truth table representation of state diagram
  - Truth table has next state function and output function
- Implement next state function and output function (old hat)



# Example FSM design procedure – 8-bit counter

- 8 states – 3 state bits
  - Use state to represent count (could use any encoding)
  - Output function is trivial
- State table has an entry for (states x inputs)
  - No inputs here, just states
  - Table output gives next state and output values




| current 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 |

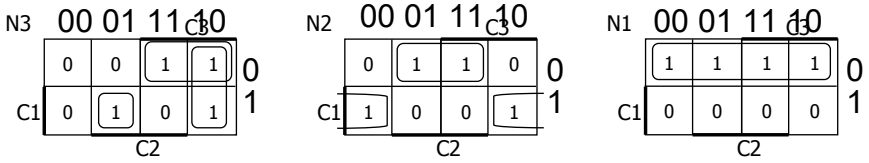
# 3-bit Counter Implementation

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

| 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 |

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

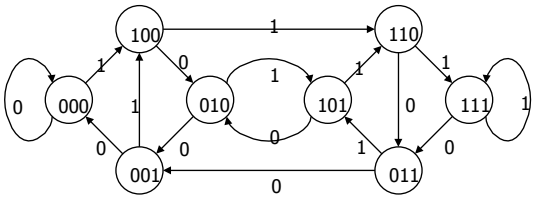
N1 <= C1'
<= C1 xor 1
N2 <= C1C2' + C1'C2
<= C1 xor C2
N3 <= C1C2C3' + C1'C3 + C2'C3
<= (C1C2)C3' + (C1' + C2')C3
<= (C1C2)C3' + (C1C2)'C3
<= (C1C2) xor C3



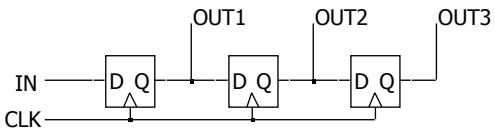



# Back to the shift register

- Input determines next state



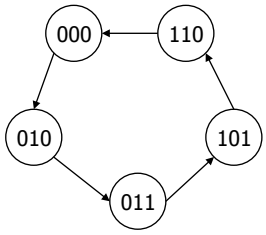

| In | C1 | C2 | C3 | N1 | N2 | N3 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1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

# 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



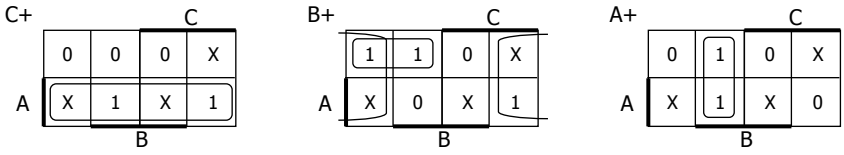

| Present State | | | Next State | | |
|---|---|---|---|---|---|
| C | B | A | C+ | B+ | A+ |
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | – | – | – |
| 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | – | – | – |
| 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | – | – | – |

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



# More complex counter example (cont'd)

- Step 3: K-maps for next state functions




C+ <= A

B+ <= B′ + A′C′

A+ <= BC′

# Self-starting counters (cont'd)

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

C+

| A \ BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

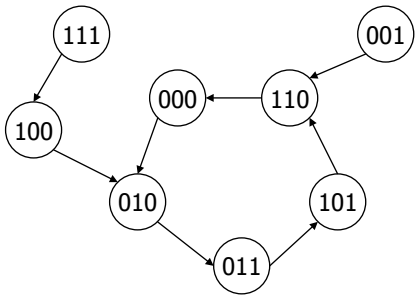
B+

| A \ BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |

A+

| A \ BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 |

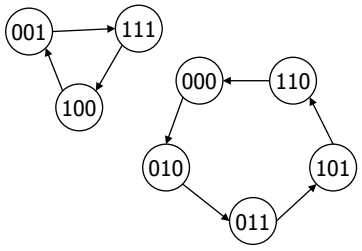
| Present State C | B | A | Next State C+ | B+ | A+ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 |

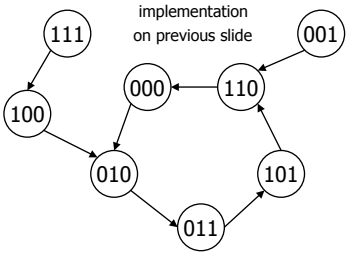





# Self-starting counters

- 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
    - this may or may not be acceptable
  - may limit exploitation of don't cares
- Or just use reset

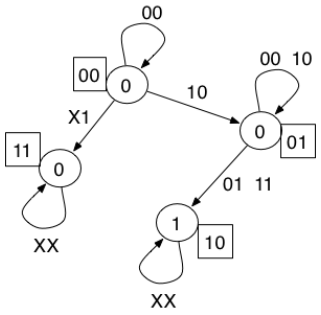


# Activity

- 2 inputs (A and B) and 1 output (+ reset)
- If A turns on first, and then B: Turn on output (until reset)
- If B turns on before A: Keep output off (until reset)

- Note that the output is a function of the state only (Moore)



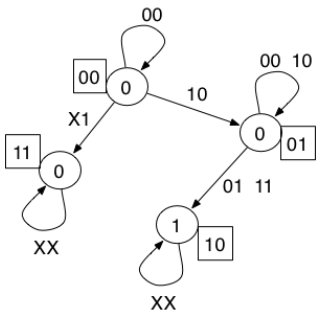

State Assignment

(Arbitrary – different encoding yields different circuits)



# Activity

- Convert state diagram to a State Table




State Table

| S1 | S0 | A | B | N1 | N0 | Out |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | - | - | 1 | 0 | 1 |
| 1 | 1 | - | - | 1 | 1 | 0 |

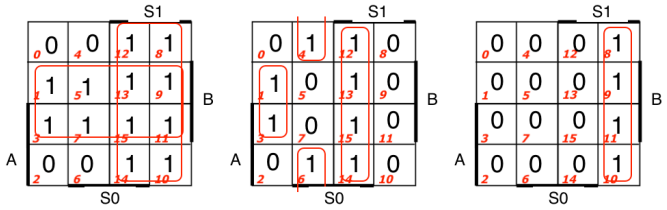
# Activity

- Implement next state and output functions

State Table

| S1 | S0 | A | B | N1 | N0 | Out |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | - | - | 1 | 0 | 1 |
| 1 | 1 | - | - | 1 | 1 | 0 |

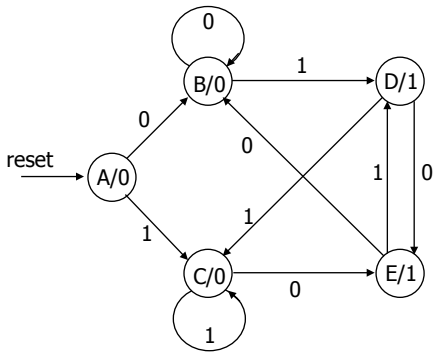
N1 <= S1 + B
N0 <= S1S0 + S1'S0'B + S1'S0A'
Out = S1S0'






# Edge Detector

- Implement an edge detector
  - Sample an input
  - Output a 1 when either the transition 0 -> 1, or 1 -> 0 is detected

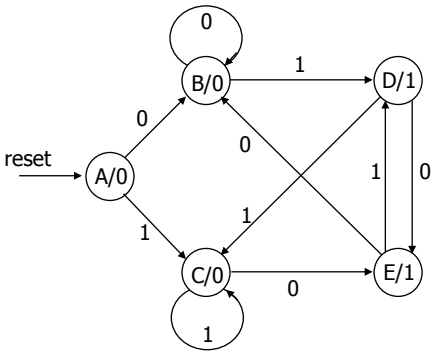



Note: Output value is associated with the state. This is a Moore machine.

State assignment has not been done. Symbolic values (A, B, C, D, E) have been used instead.

# Edge Detector

- State Table using symbolic states
- Next step: state assignment
- How many inputs do the next state and output functions have?
  - i.e. How large are the K-maps?

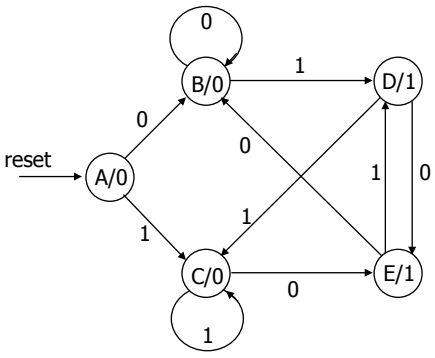



| reset | input | current state | next state | output |
|---|---|---|---|---|
| 1 | – | – | A | |
| 0 | 0 | A | B | 0 |
| 0 | 1 | A | C | 0 |
| 0 | 0 | B | B | 0 |
| 0 | 1 | B | D | 0 |
| 0 | 0 | C | E | 0 |
| 0 | 1 | C | C | 0 |
| 0 | 0 | D | E | 1 |
| 0 | 1 | D | C | 1 |
| 0 | 0 | E | B | 1 |
| 0 |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



# Edge Detector

- This is a Moore FSM
  - The output is 1 if the FSM is in state D or E
  - We can do state assignment so that one state bit is 1 only in state D and E
  - e.g. A=000, B=001, C=010, D=100, E=101

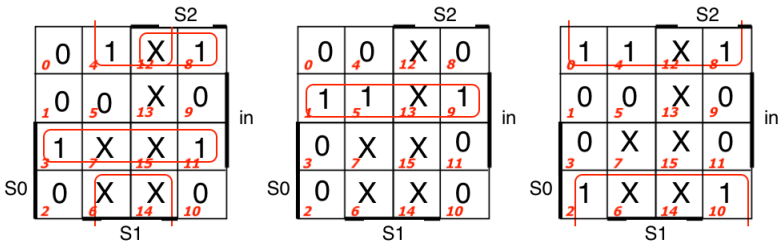



| reset | input | current state | next state | output |
|---|---|---|---|---|
| 1 | – | – | A | |
| 0 | 0 | A | B | 0 |
| 0 | 1 | A | C | 0 |
| 0 | 0 | B | B | 0 |
| 0 | 1 | B | D | 0 |
| 0 | 0 | C | E | 0 |
| 0 | 1 | C | C | 0 |
| 0 | 0 | D | E | 1 |
| 0 | 1 | D | C | 1 |
| 0 | 0 | E | B | 1 |
| 0 |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

# Edge Detector

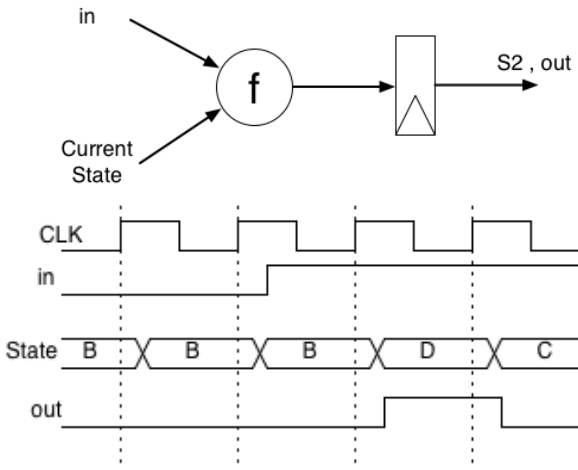
- N2 = S0in + S1in' + S2S0'in'
- N1 = S0'in
- N0 = in'

| reset | input | current state | next state | output |
|---|---|---|---|---|
| 1 | – | – | 0 0 0 | |
| 0 | 0 | 0 0 0 | 0 0 1 | 0 |
| 0 | 1 | 0 0 0 | 0 1 0 | 0 |
| 0 | 0 | 0 0 1 | 0 0 1 | 0 |
| 0 | 1 | 0 0 1 | 1 0 0 | 0 |
| 0 | 0 | 0 1 0 | 1 0 1 | 0 |
| 0 | 1 | 0 1 0 | 0 1 0 | 0 |
| 0 | 0 | 1 0 0 | 1 0 1 | 1 |
| 0 | 1 | 1 0 0 | 0 1 0 | 1 |
| 0 | 0 | 1 0 1 | 0 0 1 | 1 |
| 0 | 1 | 1 0 1 | 1 0 0 | 1 |





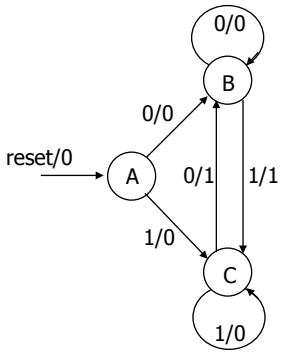

# Edge Detector: Implications of Moore Machine




Note that the output lags the input by a clock cycle
State register is between input and output

# Mealy machine

- Output is a function of both the current state and inputs
  - specify output on transition arc between states
  - Detector example
    - Note only 3 states are needed




| 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 |



# Mealy machine

- State assignment:
  - A=00, B=01, C=10

| reset | input | current state | next state | output |
|---|---|---|---|---|
| 1 | – | – | 0 0 | 0 |
| 0 | 0 | 0 0 | 0 1 | 0 |
| 0 | 1 | 0 0 | 1 0 | 0 |
| 0 | 0 | 0 1 | 0 1 | 0 |
| 0 | 1 | 0 1 | 1 0 | 1 |
| 0 | 0 | 1 0 | 0 1 | 1 |
| 0 | 1 | 1 0 | 1 0 | 0 |

N1 = in

| | S0 | | | |
|---|---|---|---|---|
| in | 0 | 0 | X | 0 |
| | 1 | 1 | X | 1 |
| | | | | S1 |

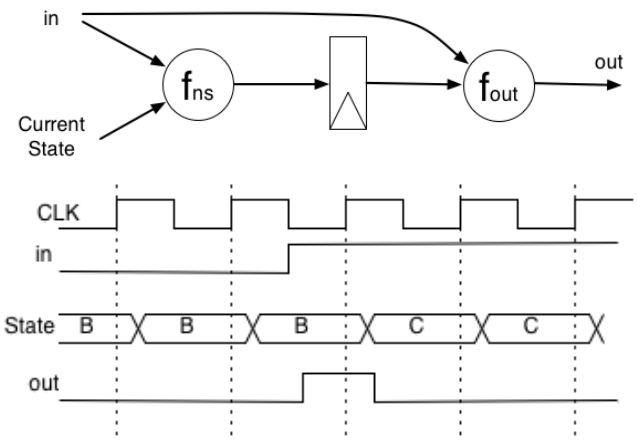
N0 = in'

| | S0 | | | |
|---|---|---|---|---|
| in | 1 | 1 | X | 1 |
| | 0 | 0 | X | 0 |
| | | | | S1 |

| | S0 | | | |
|---|---|---|---|---|
| in | 0 | 0 | X | 1 |
| | 0 | 1 | X | 0 |
| | | | | S1 |

out = S1in' + S0in

# Edge Detector – Implications of Mealy Machine

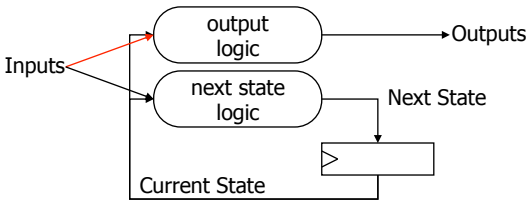



Note that the output changes as soon as the input changes.
But glitches on input get passed along!
Output reacts faster, but may add delay to critical path.



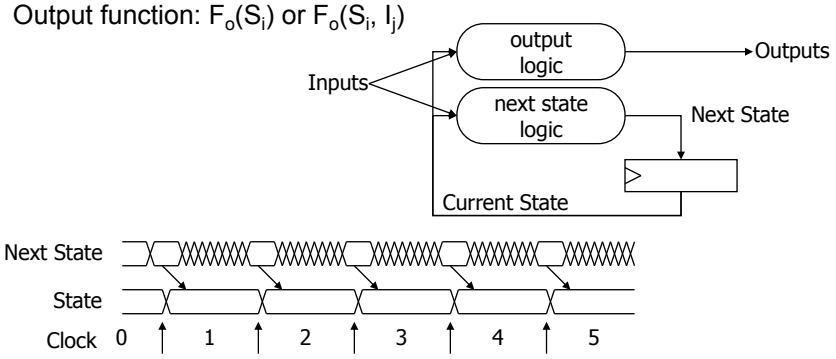
# General state machine model revisited

- 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)



# State machine model (cont'd)

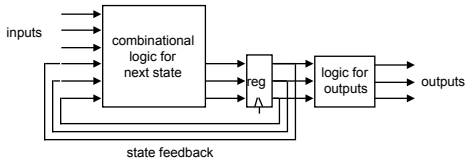
- States: $S_1$, $S_2$, ..., $S_k$
- Inputs: $I_1$, $I_2$, ..., $I_m$
- Outputs: $O_1$, $O_2$, ..., $O_n$
- Transition function: $F_s(S_i, I_j)$
- Output function: $F_o(S_i)$ or $F_o(S_i, I_j)$



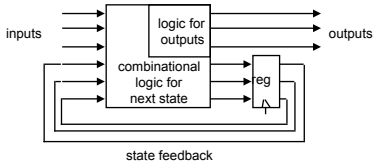



# Comparison of Mealy and Moore machines (cont'd)

- Moore




- Mealy



# Comparison of Mealy and Moore machines

- Mealy machines tend to have less states
  - outputs depend on arc taken from a state to another state ($n^2$) rather than just the state of the FSM (n)
- Moore machines are safer to use
  - outputs change at next clock edge
  - in Mealy machines, input change can cause async output change (after prop delay of logic) – a BIG problem when two machines are interconnected – asynchronous feedback may occur if one isn't careful (input to fsm1, changes output of fsm1, which is an input to fsm2, whose output changes, and turns out to be input to fsm1)
- Mealy machines advantage? – they 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 that are needed – more gate delays after clock edge

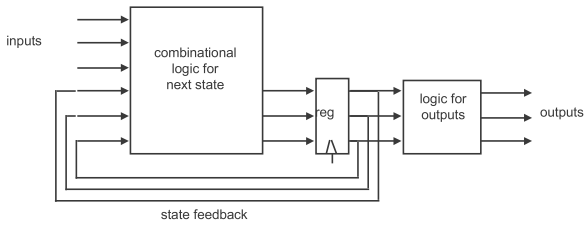
# Verilog for Finite State Machines

- Strongly recommended style for FSMs
- Works for both Mealy and Moore FSMs
- You can break the rules
  - But you have to live with the consequences

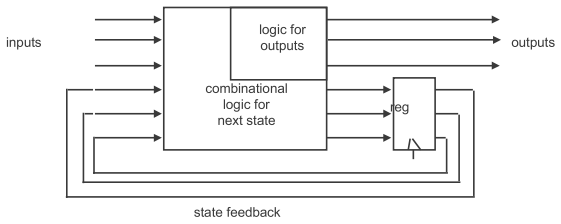


# Mealy and Moore machines

- Moore




- Mealy



## Constructing State Machines in Verilog

- We need register to hold
- the current state
  - always @(posedge clk) block
- We need next state function
  - Where do we go from each state given the inputs
  - state by state case analysis
    - next state determined by current state and inputs
- We need the output function
  - State by state analysis
  - Moore: output determined by current state only
  - Mealy: output determined by current state and inputs



## State Register

- Declare two values
  - state : current state – output of state register
  - nxtState : next state – input to state register
  - We rely on next state function to give us nxtState
- Declare symbols for states with state assignment

```
localparam IDLE=0, WAITFORB=1,
           DONE=2, ERROR=3;

reg [1:0] state,    // Current state
          nxtState; // Next state
```

## State Register

- Simple code for register
  - Define reset state
  - Otherwise, just move to nxtState on clock edge

```
localparam IDLE=0, WAITFORB=1,
           DONE=2, ERROR=3;
reg [1:0] state,    // Current state
          nxtState; // Next state

always @(posedge clk) begin
  if (reset) begin
    state <= IDLE;   // Initial state
  end else begin
    state <= nxtState;
  end
end
```



## Next State Function

- Combinational logic function
  - Inputs : state, inputs
  - Output : nxtState
- We could use assign statements
- We will use an **`always @(*)`** block instead
  - Allows us to use more complex statements
  - **`if`**
  - **`case`**