Finite State Machine

1. Review on counter design
2. State Diagrams for FSM
3. Moore & Mealy Models
4. State Minimization
5. Examples
6. HDL for FSM

Sequential Logic Implementation

- Models for representing sequential circuits
  - Finite-state machines (Moore and Mealy)
  - Representation of memory (states)
  - Changes in state (transitions)
- Design procedure
  - State diagrams
  - State transition table
  - Next state functions

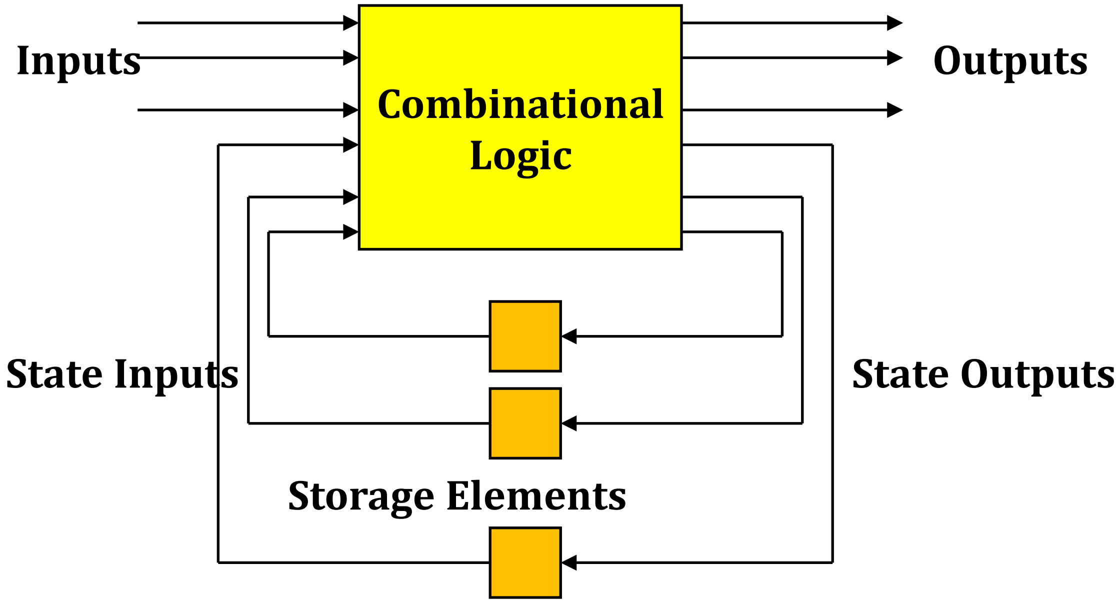
2 – STATE DIAGRAM FOR FSM

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

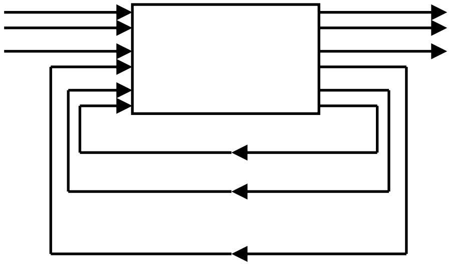
Abstraction of state elements

1. Divide circuit
   - combinational logic and state
2. Feedback loops
   - Localize
   - Break cycles
3. storage elements
   - sequential logic

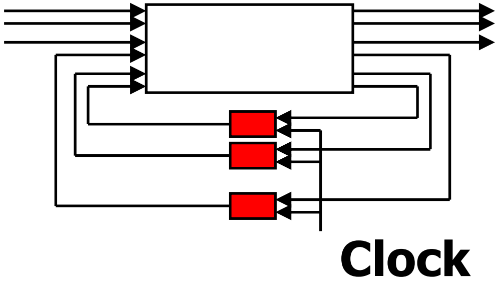


Forms of sequential logic

1. Asynchronous sequential logic
   - state changes occur whenever state inputs change
   - elements may be simple wires or delay elements

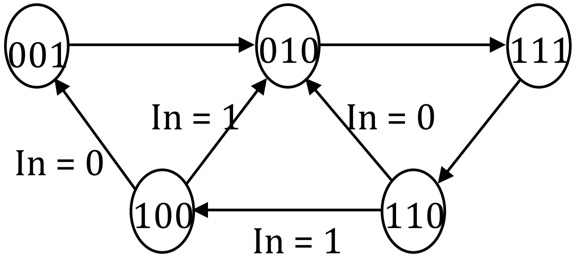


2. Synchronous sequential logic
   - state changes occur in lock step across all storage elements
   - using a periodic waveform - the clock



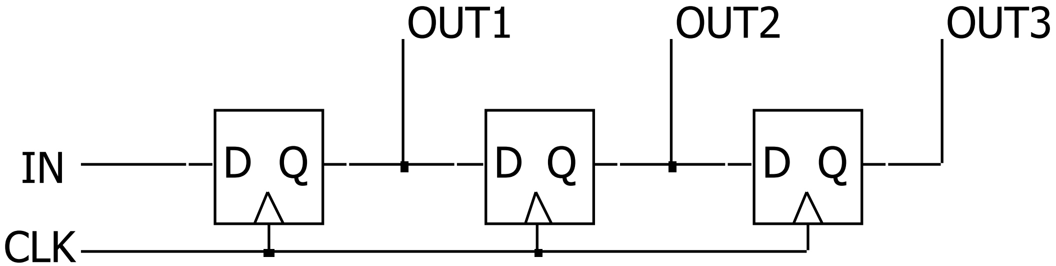
FSM Representations

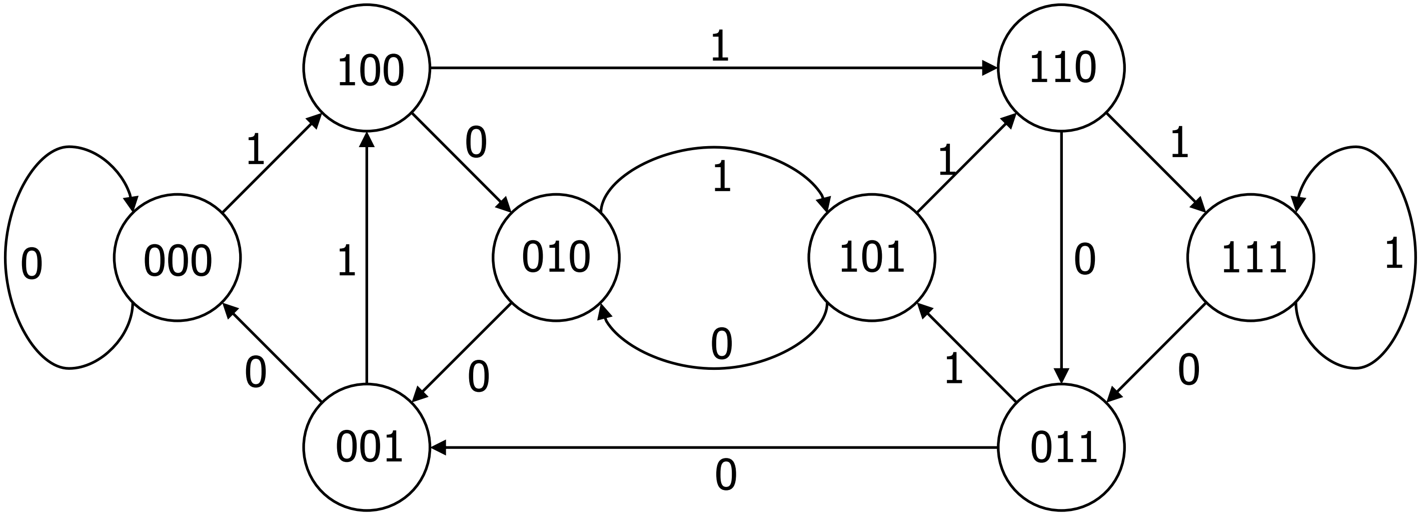
1. States
   - determined by possible values in sequential storage elements
2. Transitions
   - change of state
3. Clock
   - controls when state can change by controlling storage elements
4. Sequential logic
   - sequences through a series of states
   - based on sequence of values on input signals
   - clock period defines elements of sequence



Can any sequential system be represented with a state diagram?

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

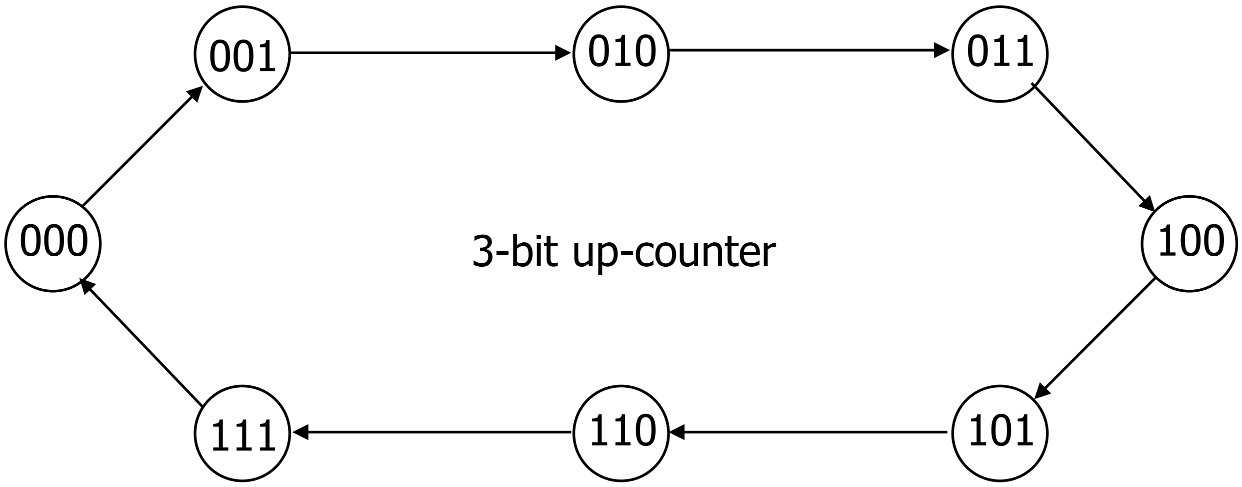




1- DESIGN A COUNTER

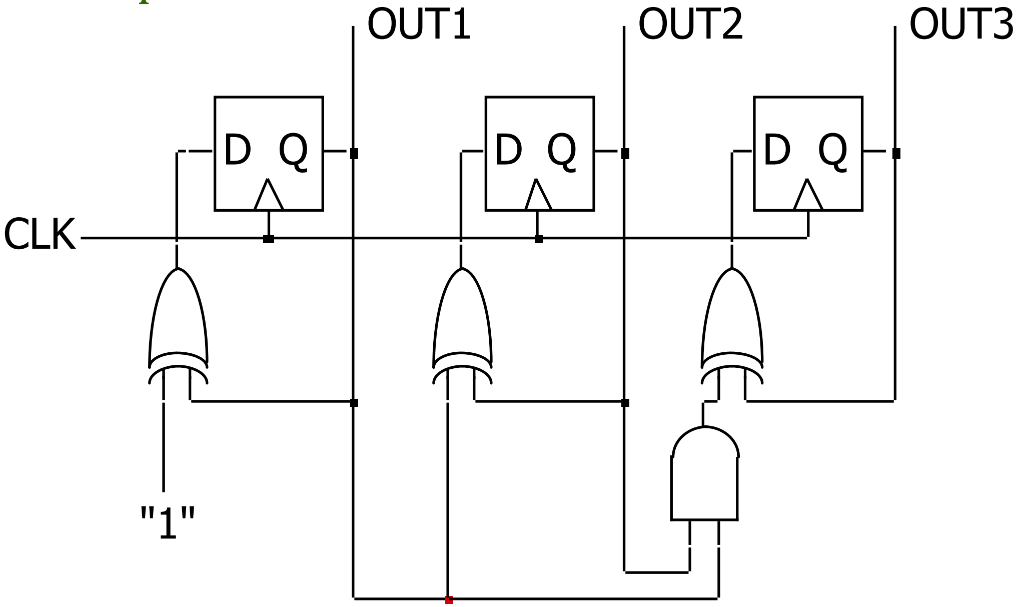
Counters are simple finite state machines

- 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, ...



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

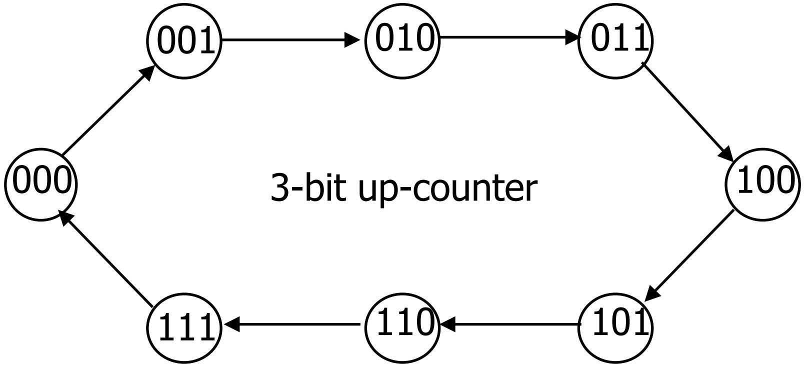


FSM design procedure

- 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

FSM design procedure: state diagram to encoded state transition table – counter case 1

- 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

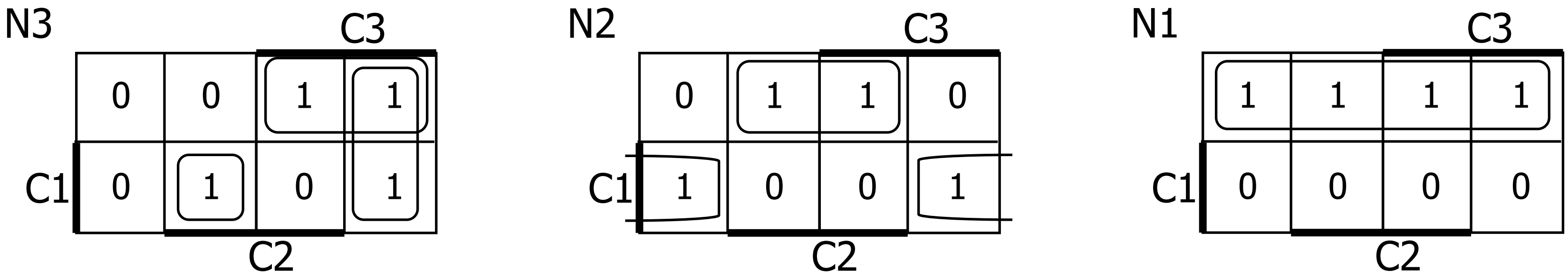
Implementation

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



N3

	0	0	1	1
C1	0	1	0	1

N2

	0	1	1	0
C1	1	0	0	1

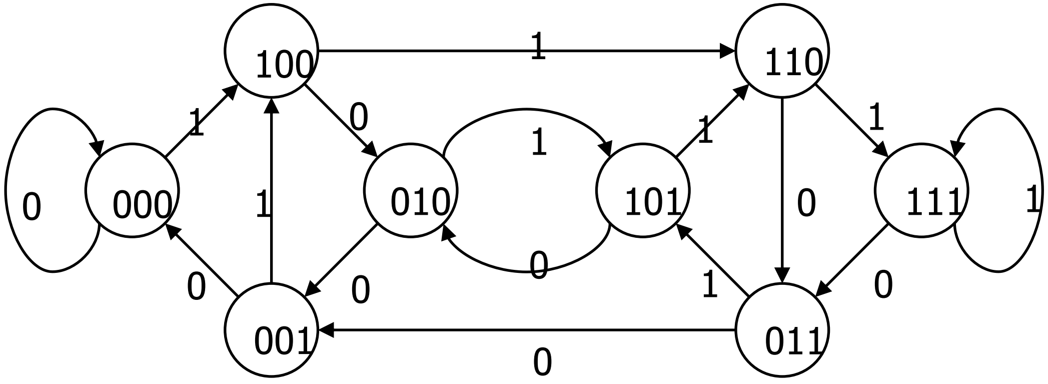
N1

	1	1	1	1
C1	0	0	0	0

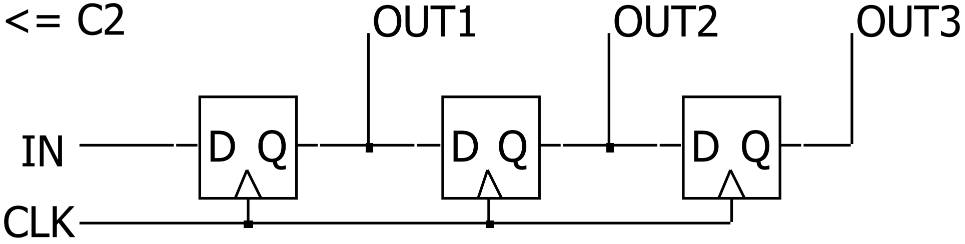
Back to the shift register - counter case 2

- 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

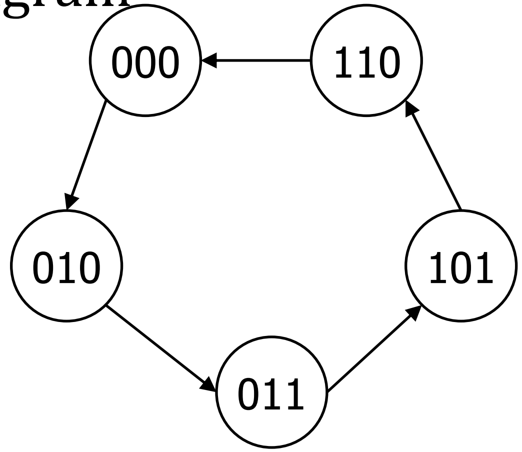


N1 <= In
N2 <= C1
N3 <= C2



More complex counter example - counter case 3

- 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

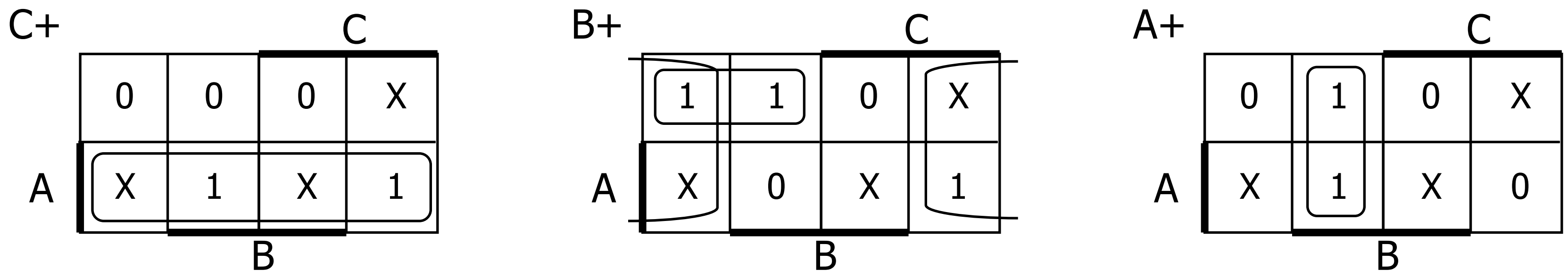


Present State			Next State		
C	B	A	C+	B+	A+
0	0	0	0	1	0
0	0	1	–	–	–
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	–	–	–
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	–	–	–

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) – counter case 3a

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

C+ (C, A, B)

0	0	0	0
1	1	1	1

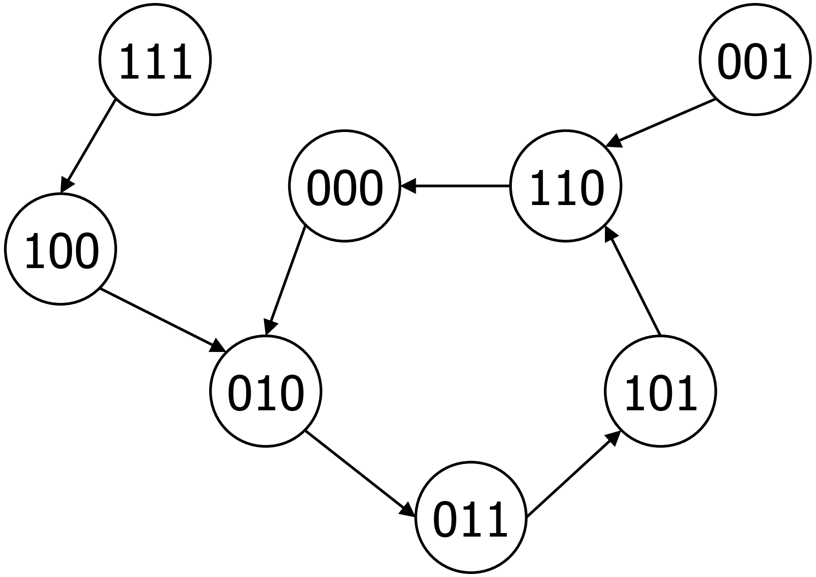
B+ (C, A, B)

1	1	0	1
1	0	0	1

A+ (C, A, B)

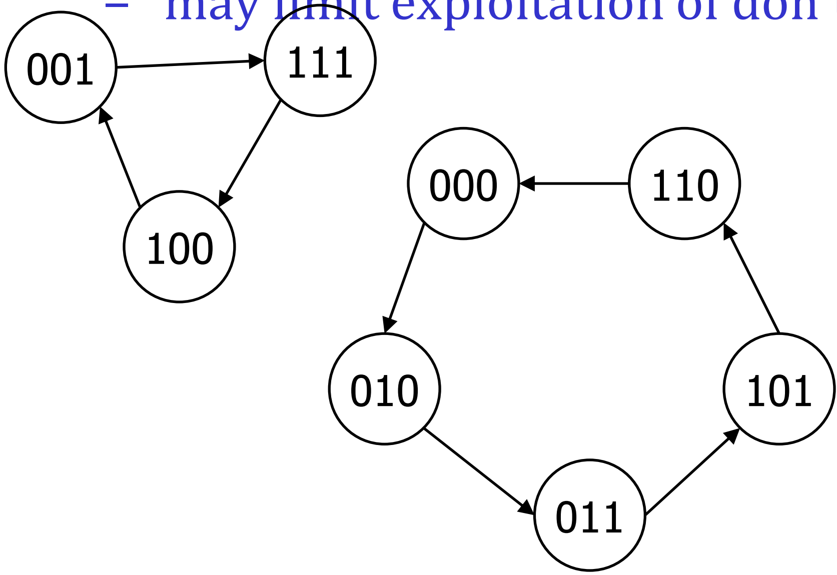
0	1	0	0
0	1	0	0

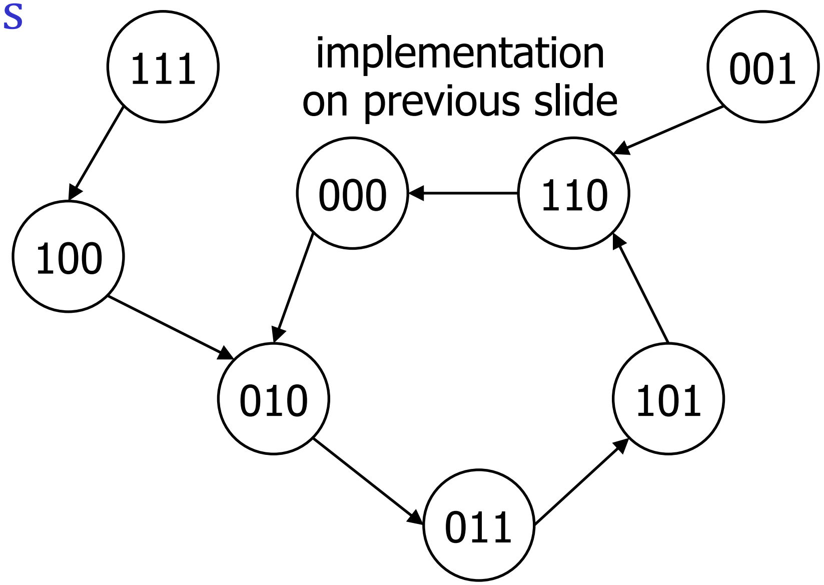
Present State			Next State		
C	B	A	C+	B+	A+
0	0	0	0	1	0
0	0	1	1	1	0
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	0	1	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	1	0	0



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





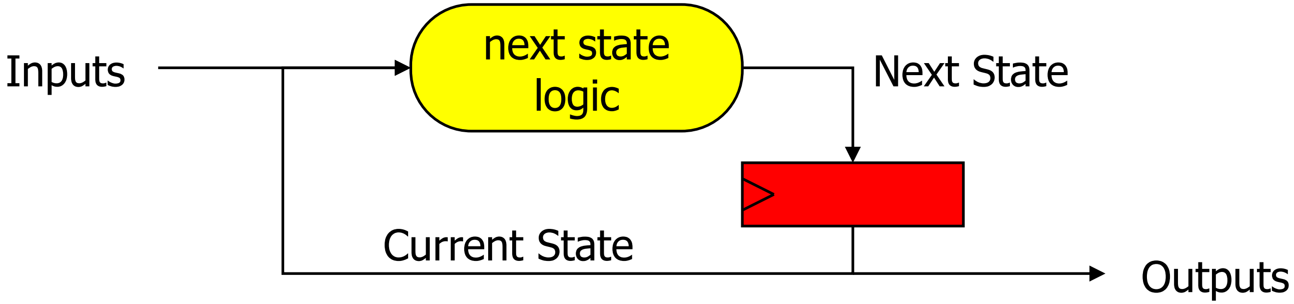
Activity

- 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)

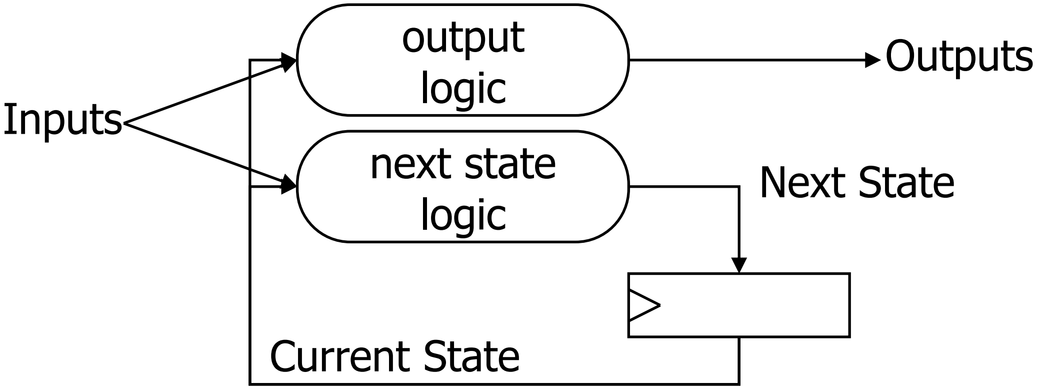
Counter/shift-register model

- 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



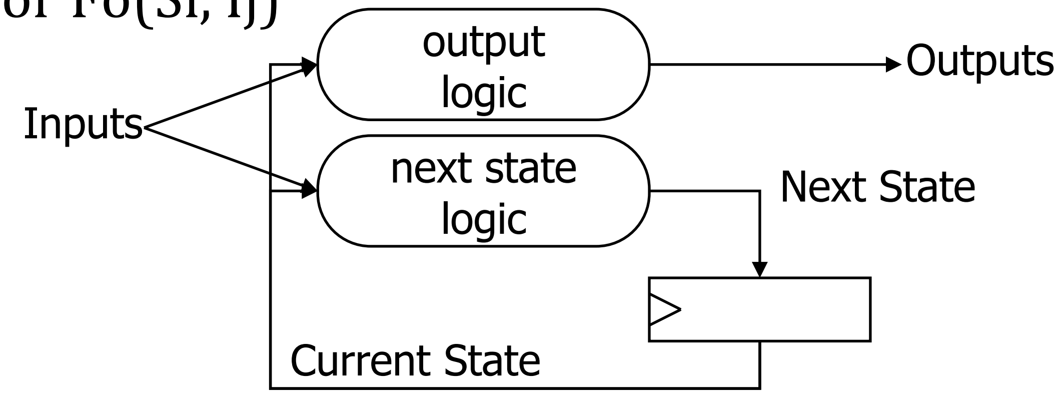
General state machine model

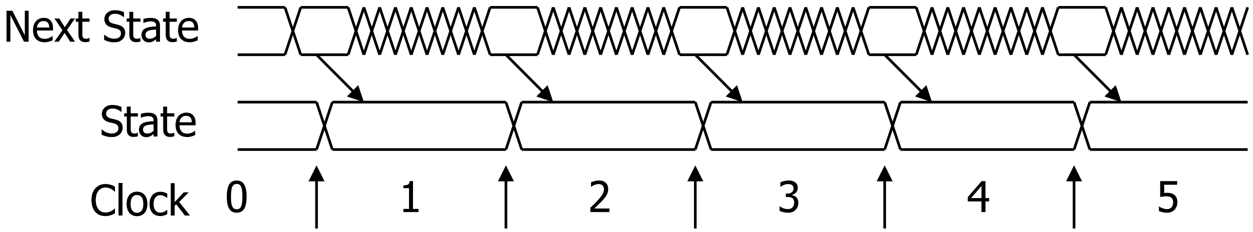
- Values stored in registers represent the state of the circuit
- Combinational logic computes:
  - next state
    - function of current state and inputs
  - outputs
    - function of current state and inputs (Mealy machine)
    - function of current state only (Moore machine)



State machine model (cont'd)

- States: S1, S2, ..., Sk
- Inputs: I1, I2, ..., Im
- Outputs: O1, O2, ..., On
- Transition function: Fs(Si, Ij)
- Output function: Fo(Si) or Fo(Si, Ij)





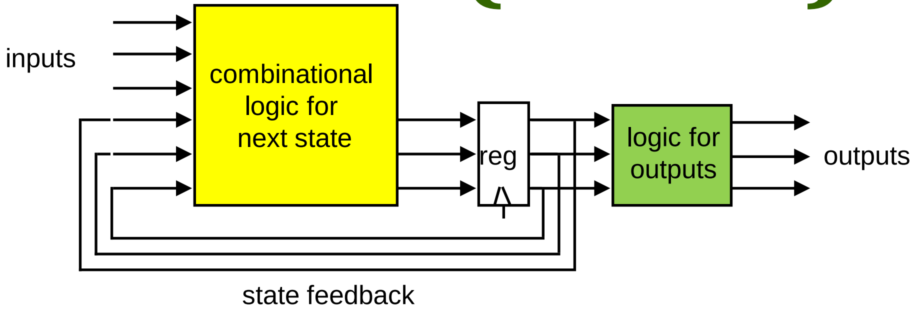
3 - MOORE & MEALY MACHINES

Mealy vs Moore Machines

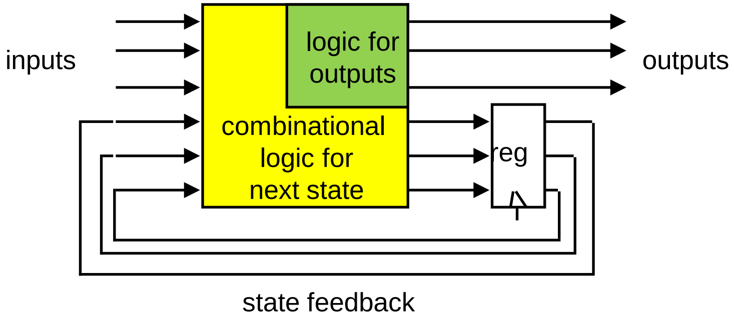
- Mealy machines tend to have less states
  - different outputs on arcs (n2) rather than states (n)
- Moore machines are safer to use
  - outputs change at clock edge (always one cycle later)
  - in Mealy machines, input change can cause output change as soon as logic is done – a big problem when two 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

Comparison of Mealy and Moore machines (cont'd)

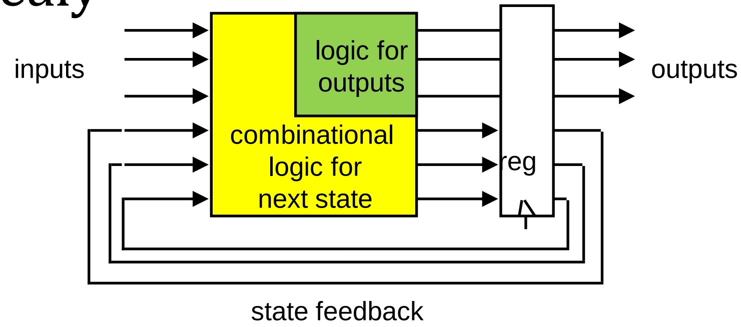
- Moore



- Mealy

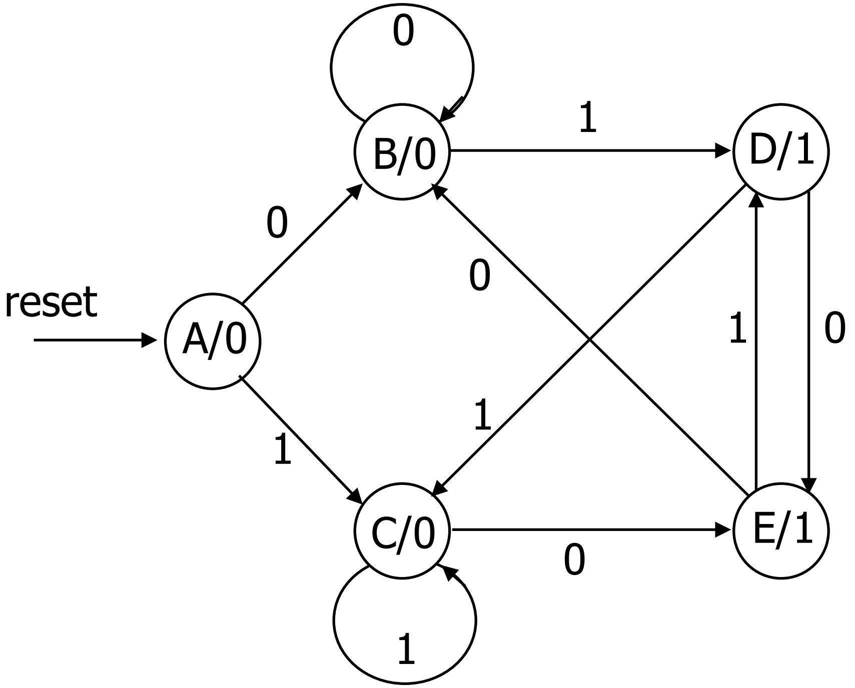


- Synchronous Mealy



Outputs for a Moore machine

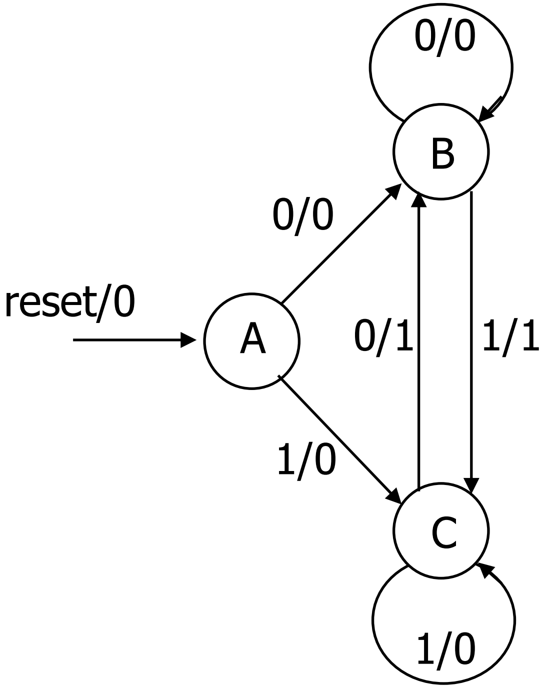
- Output is only function of state
  - specify in state bubble in state diagram
  - example: sequence detector for 01 or 10



		current	next	
reset	input	state	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

Outputs for a Mealy machine

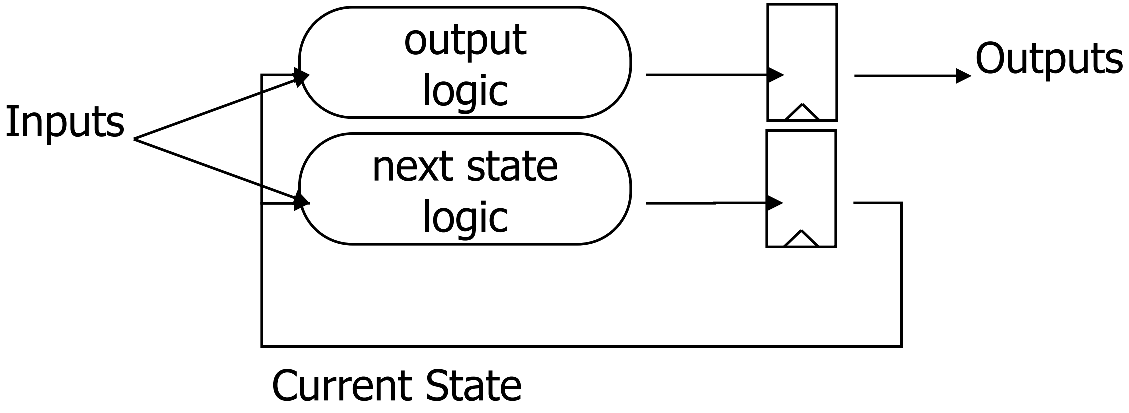
- Output is function of state and inputs
  - specify output on transition arc between states
  - example: sequence detector for 01 or 10



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

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



Activity

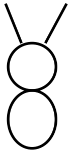
A 3-bit sequence detector will detect 111 and 001. All the input and output will be reset after each sequence had been detected. An example of input/output sequence that satisfies the conditions of the network specifications is as follows;

x = 0 1 1   1 1 1   0 1 0   1 0 0   0 0 1   0 1 1
z = 0 0 0   0 0 1   0 0 0   0 0 0   0 0 1   0 0 0

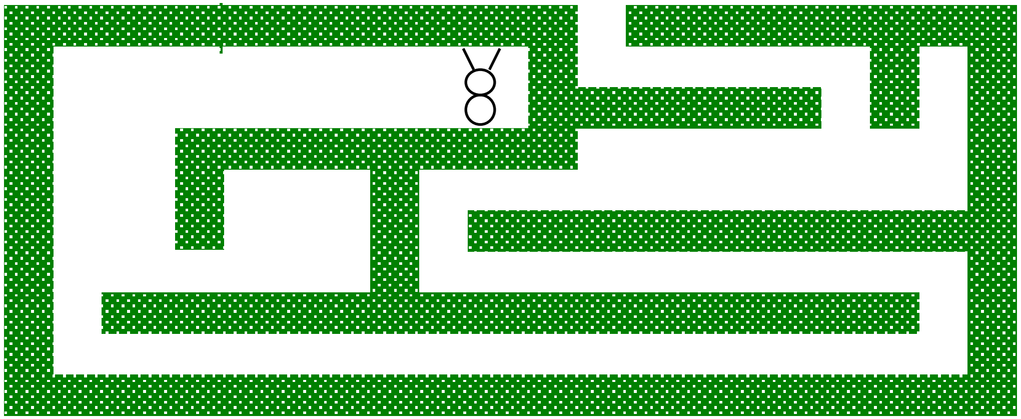
Draw the state transition diagram of the system using Mealy machine method.

Example: Ant Brain (Ward, MIT)

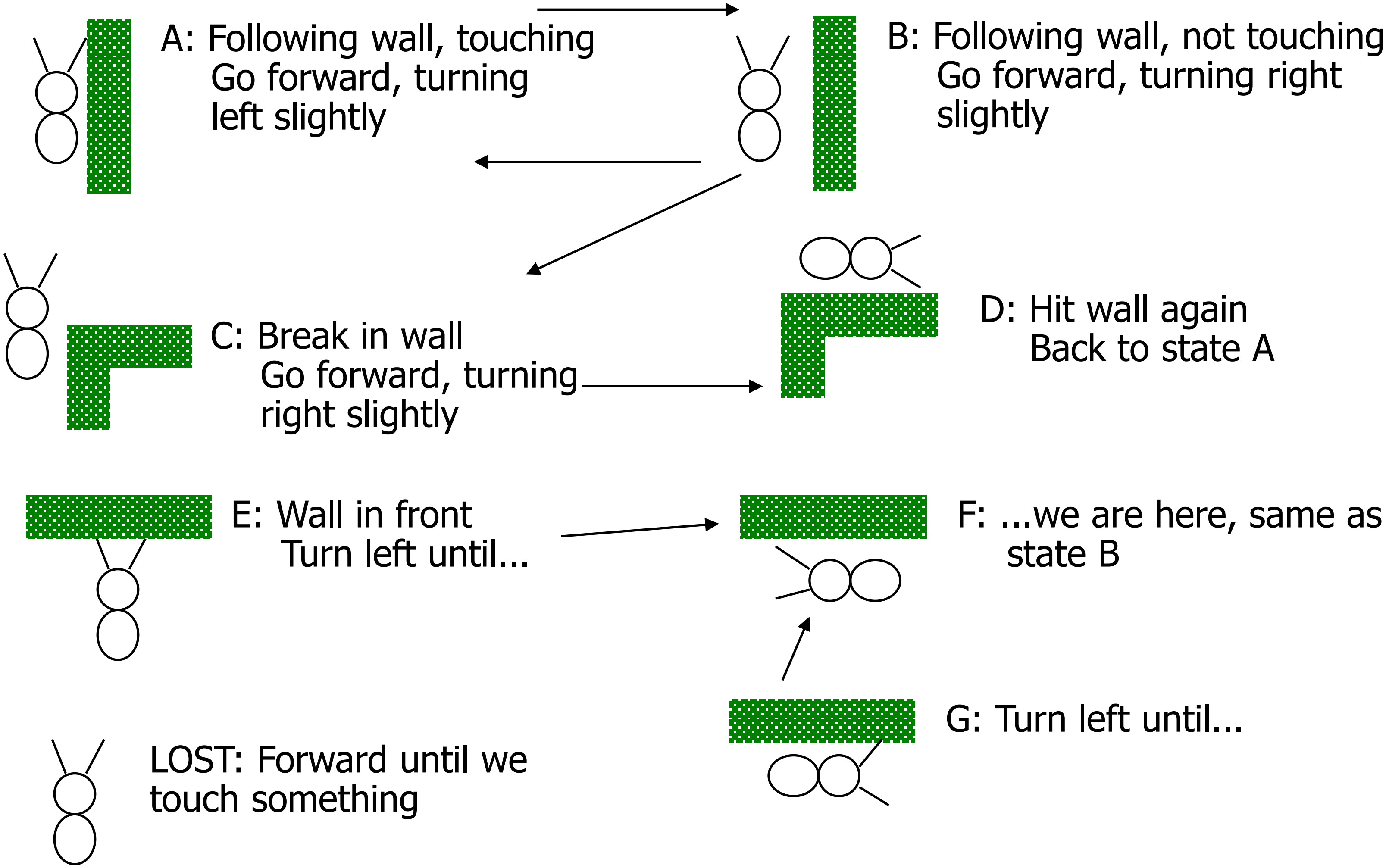
- Sensors:
  - L and R antennae, 1 if in touching wall
- Actuators:
  - F - forward step, TL/TR - turn left/right slightly



- Goal:
  - find way out of maze
- Strategy:
  - keep the wall on the right



Ant Behavior



A: Following wall, touching
Go forward, turning left slightly

B: Following wall, not touching
Go forward, turning right slightly

C: Break in wall
Go forward, turning right slightly

D: Hit wall again
Back to state A

E: Wall in front
Turn left until...

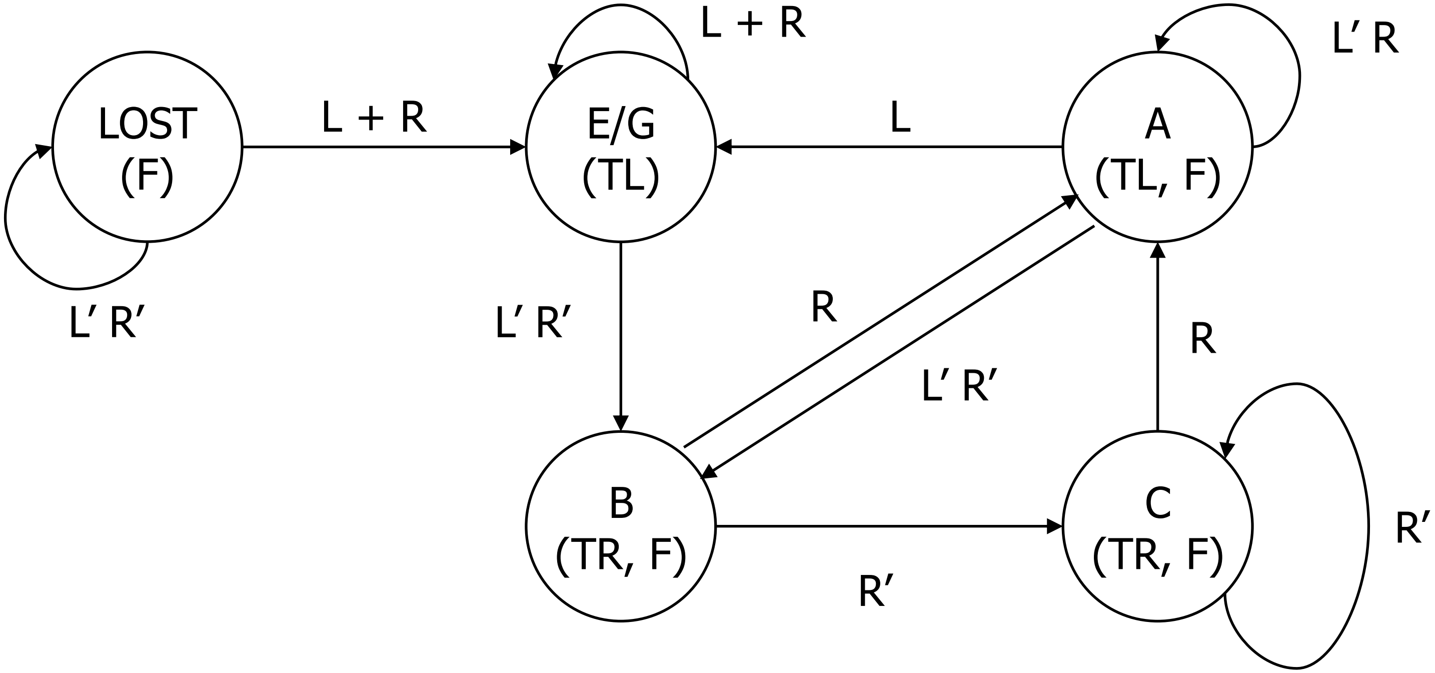
F: ...we are here, same as state B

G: Turn left until...

LOST: Forward until we touch something

Designing an Ant Brain

- State Diagram

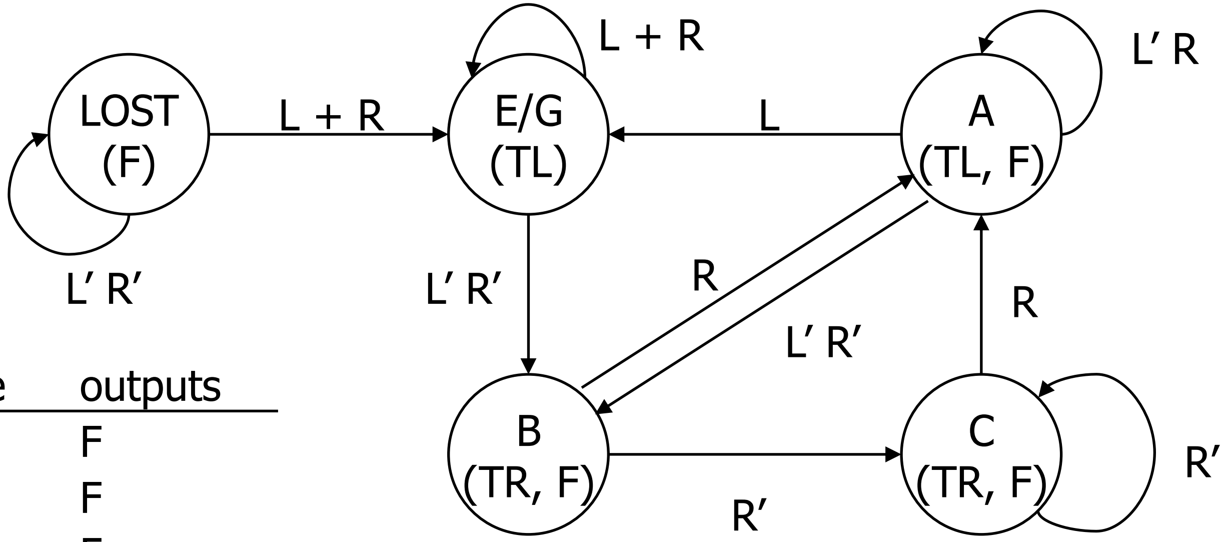


Synthesizing the Ant Brain Circuit

- Encode States Using a Set of State Variables
  - Arbitrary choice - may affect cost, speed
- Use Transition Truth Table
  - Define next state function for each state variable
  - Define output function for each output
- Implement next state and output functions using combinational logic
  - 2-level logic (ROM/PLA/PAL)
  - Multi-level logic
  - Next state and output functions can be optimized together

Transition Truth Table

- Using symbolic states and outputs



state	L	R	next state	outputs
LOST	0	0	LOST	F
LOST	–	1	E/G	F
LOST	1	–	E/G	F
A	0	0	B	TL, F
A	0	1	A	TL, F
A	1	–	E/G	TL, F
B	–	0	C	TR, F
B	–	1	A	TR, F
...	...	...	...	...

Synthesis

- 5 states : at least 3 state variables required (X, Y, Z)
  - State assignment (in this case, arbitrarily chosen)

LOST	-	000
E/G	-	001
A	-	010
B	-	011
C	-	100

state	L	R	next state	outputs		
X,Y,Z			X', Y', Z'	F	TR	TL
0 0 0	0	0	0 0 0	1	0	0
0 0 0	0	1	0 0 1	1	0	0
...	...	...	...	...		
0 1 0	0	0	0 1 1	1	0	1
0 1 0	0	1	0 1 0	1	0	1
0 1 0	1	0	0 0 1	1	0	1
0 1 0	1	1	0 0 1	1	0	1
0 1 1	0	0	1 0 0	1	1	0
0 1 1	0	1	0 1 0	1	1	0
...	...	...	...	...		

it now remains to synthesize these 6 functions

Synthesis of Next State and Output Functions

state	inputs		next state	outputs		
X,Y,Z	L	R	X^+,Y^+,Z^+	F	TR	TL
0 0 0	0	0	0 0 0	1	0	0
0 0 0	-	1	0 0 1	1	0	0
0 0 0	1	-	0 0 1	1	0	0
0 0 1	0	0	0 1 1	0	0	1
0 0 1	-	1	0 1 0	0	0	1
0 0 1	1	-	0 1 0	0	0	1
0 1 0	0	0	0 1 1	1	0	1
0 1 0	0	1	0 1 0	1	0	1
0 1 0	1	-	0 0 1	1	0	1
0 1 1	-	0	1 0 0	1	1	0
0 1 1	-	1	0 1 0	1	1	0
1 0 0	-	0	1 0 0	1	1	0
1 0 0	-	1	0 1 0	1	1	0

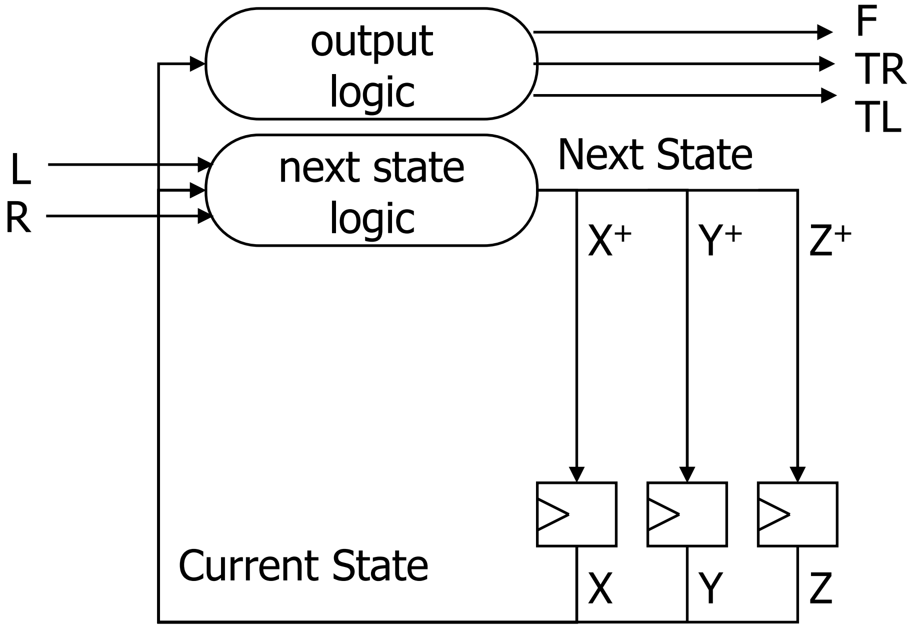
e.g.

$$TR = X + Y Z$$

$$X^+ = X R' + Y Z R' = R' TR$$

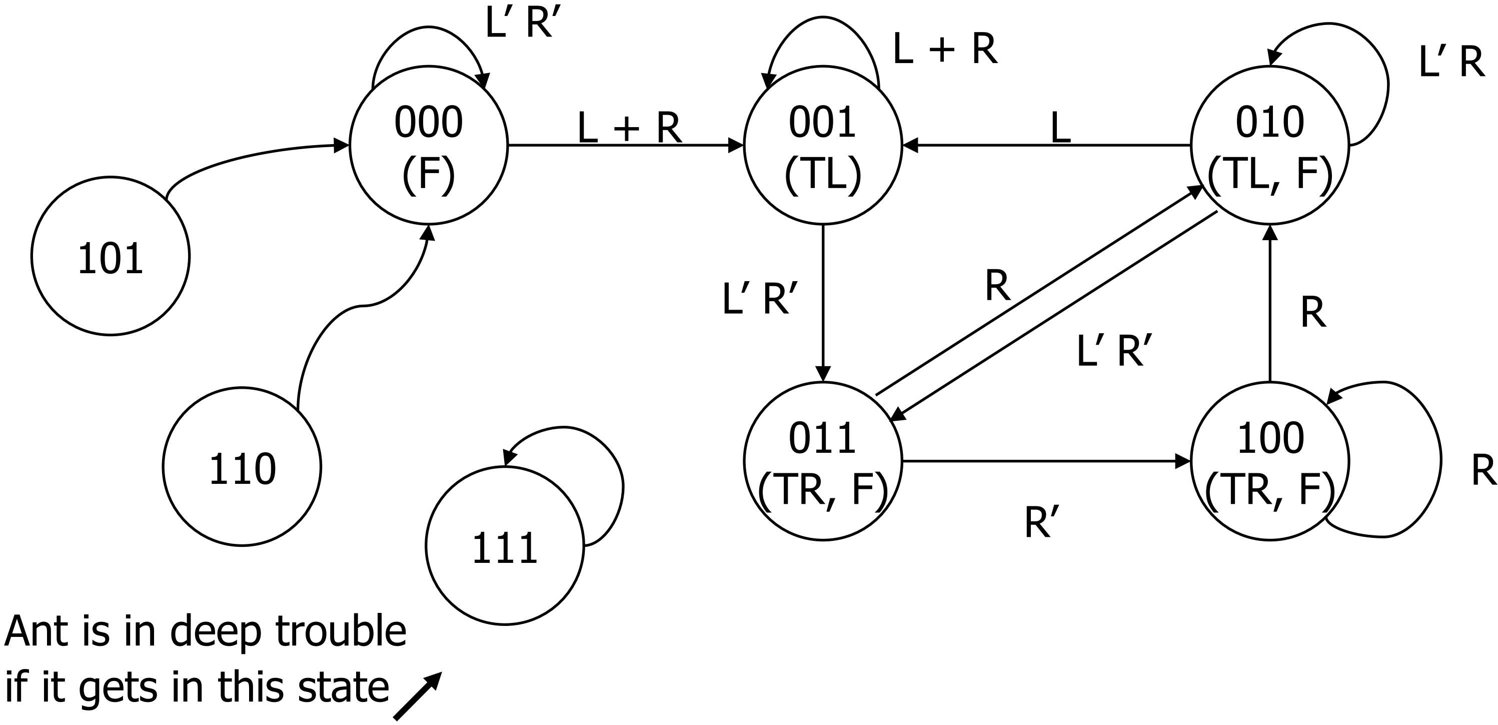
Circuit Implementation

- Outputs are a function of the current state only - Moore machine



Don’t Cares in FSM Synthesis

- What happens to the "unused" states (101, 110, 111)?
- Exploited as don't cares to minimize the logic
  - If states can't happen, then don't care what the functions do
  - if states do happen, we may be in trouble



Ant is in deep trouble
if it gets in this state

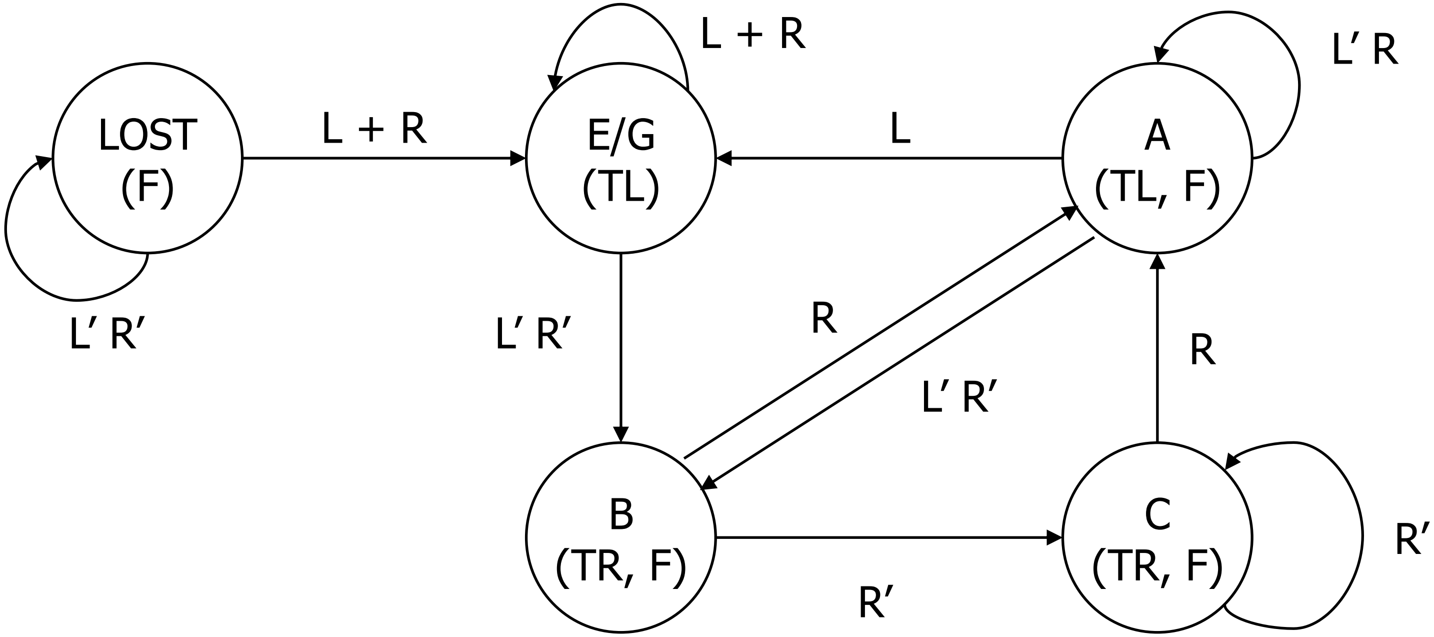
State Minimization

- Fewer states may mean fewer state variables
- High-level synthesis may generate many redundant states
- Two state are equivalent if they are impossible to distinguish from the outputs of the FSM, i. e., for any input sequence the outputs are the same

- Two conditions for two states to be equivalent:
  - 1) Output must be the same in both states
  - 2) Must transition to equivalent states for all input combinations

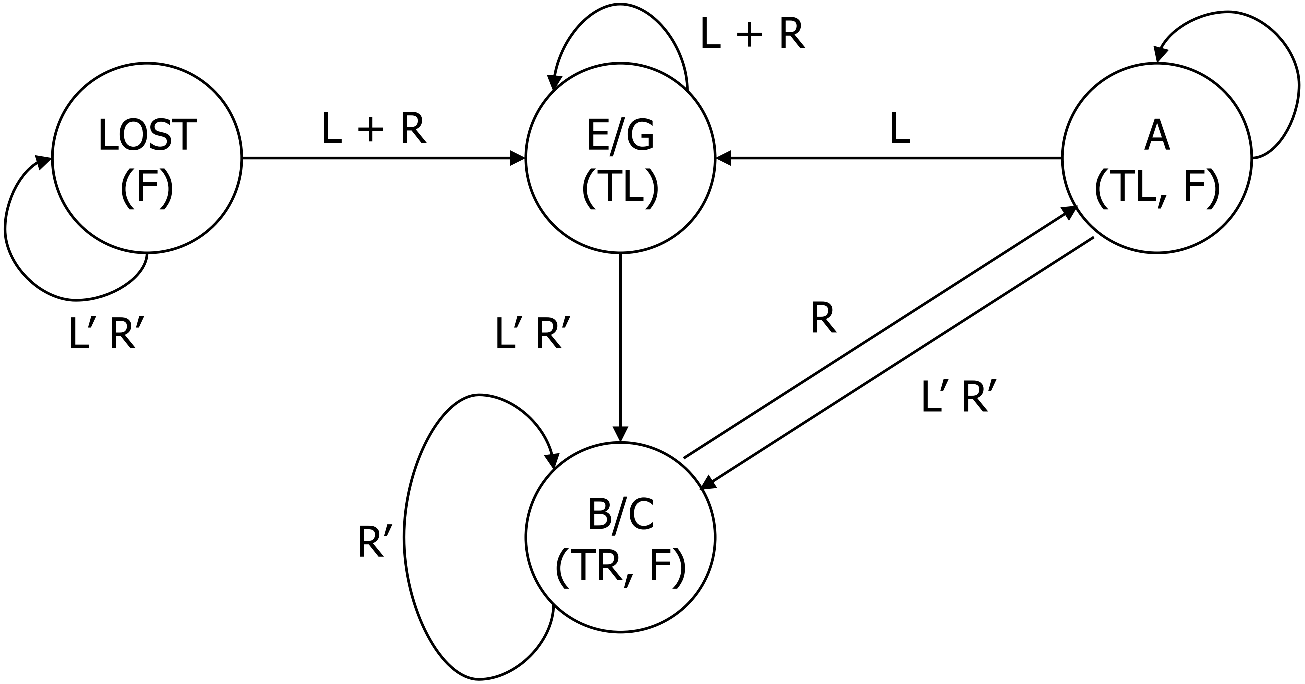
Ant Brain Revisited

- Any equivalent states?



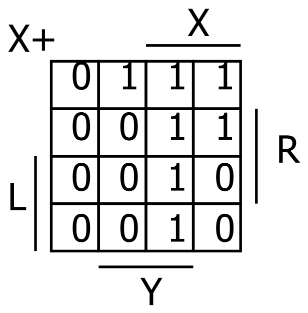
New Improved Brain

- Merge equivalent B and C states
- Behavior is exactly the same as the 5-state brain
- We now need only 2 state variables rather than 3

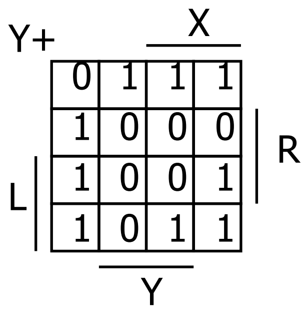


New Brain Implementation

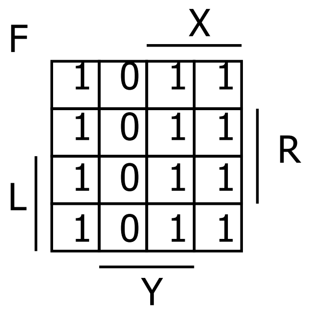
state	inputs		next state	outputs		
X,Y	L	R	X',Y'	F	TR	TL
0 0	0	0	0 0	1	0	0
0 0	-	1	0 1	1	0	0
0 0	1	-	0 1	1	0	0
0 1	0	0	1 1	0	0	1
0 1	-	1	0 1	0	0	1
0 1	1	-	0 1	0	0	1
1 0	0	0	1 1	1	0	1
1 0	0	1	1 0	1	0	1
1 0	1	-	0 1	1	0	1
1 1	-	0	1 1	1	1	0
1 1	-	1	1 0	1	1	0



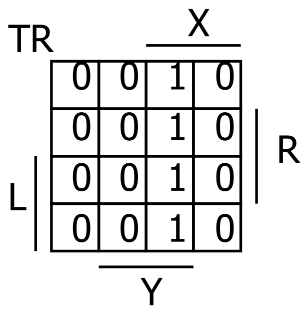
0	1	1	1
0	0	1	1
0	0	1	0
0	0	1	0



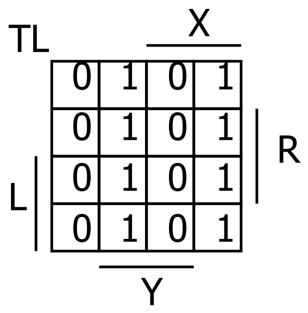
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	1



1	0	1	1
1	0	1	1
1	0	1	1
1	0	1	1



0	0	1	0
0	0	1	0
0	0	1	0
0	0	1	0



0	1	0	1
0	1	0	1
0	1	0	1
0	1	0	1

Description of Vending Machine Question

- The vending machine delivers a package of gum after it has received 15 cents in coins. The machine has a single coin slot that accepts nickels and dimes, one coin at a time. A mechanical sensor indicates to the control whether a dime or a nickel has been inserted into the coin slot. The controller's output causes a single package of gum to be released down a chute to the -customer.

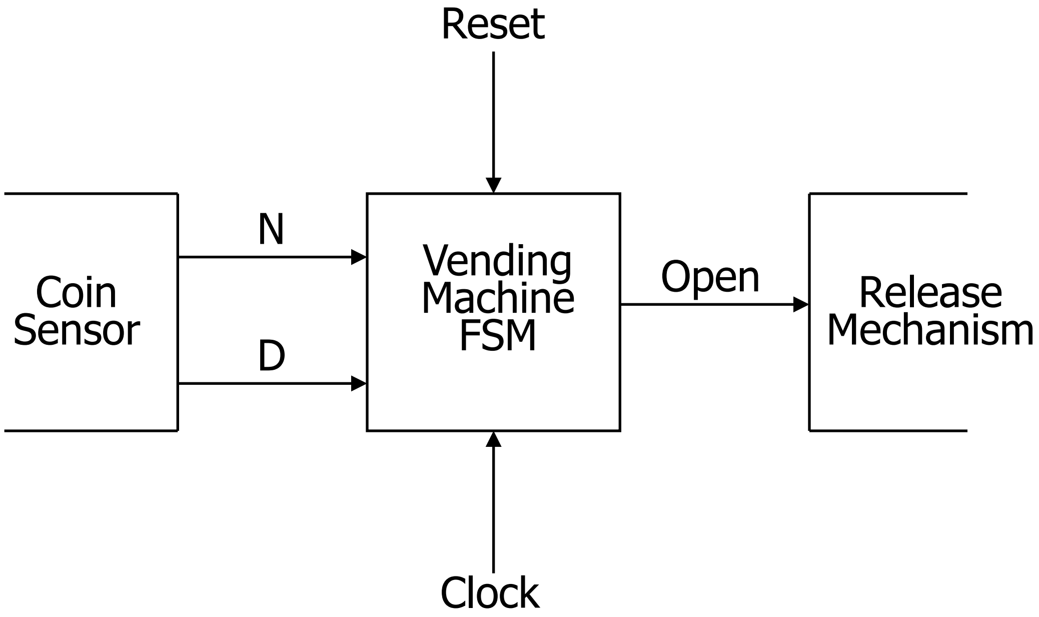
  One further specification: We will design our machine so it does not give change. A customer who pays with two dimes is out 5 cents!
- The specification may not completely define the behavior of the finite state machine. For example, what happens if someone inserts a penny into the coin slot? Or what happens after the gum is delivered to the customer? Sometimes we have to make reasonable assumptions. For the first question, we assume that the coin sensor returns any coins it does not recognize, leaving *N* and *D* unasserted. For the latter, we assume that external logic resets the machine after the gum is delivered.

Condition – particularly to this case

- Second we will expect our machine **to be reset before each new use.** This might be accomplished through **separates mechanism** the customer uses to select the item to be purchace

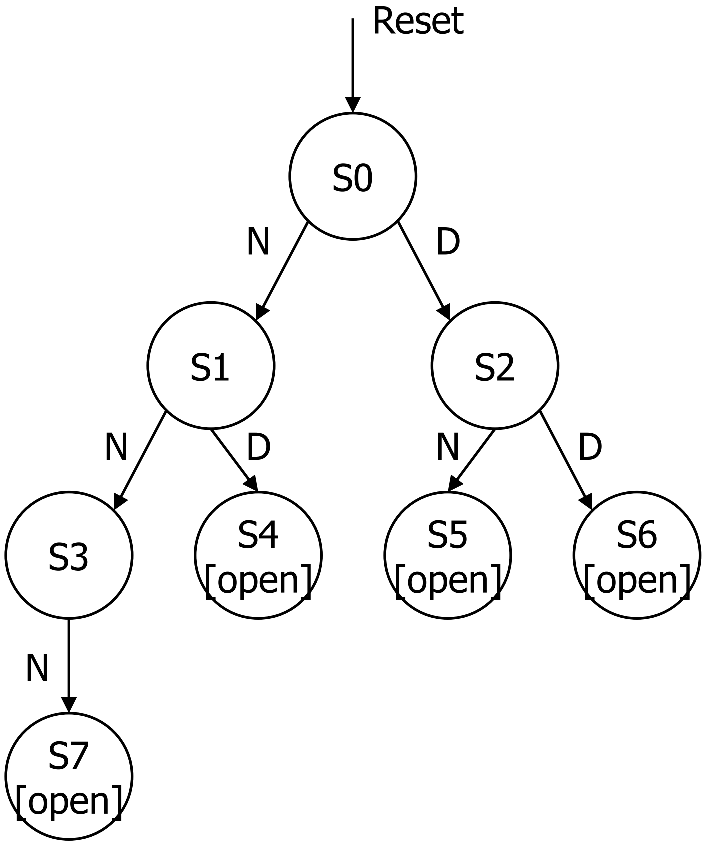
Example: Vending Machine

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



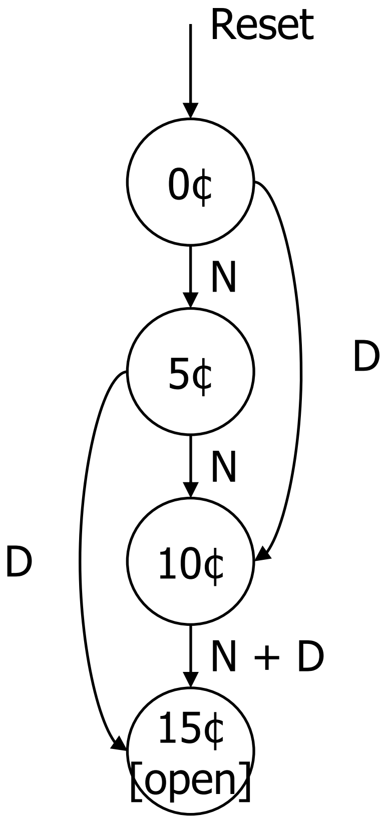
Example: Vending Machine (cont'd)

- Suitable Abstract Representation
  - Tabulate typical input sequences:
    - 3 nickels
    - nickel, dime
    - dime, nickel
    - two dimes
  - 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)



Example: Vending Machine (cont’d)

- Minimize number of states - reuse states whenever possible



present state	inputs D	inputs 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

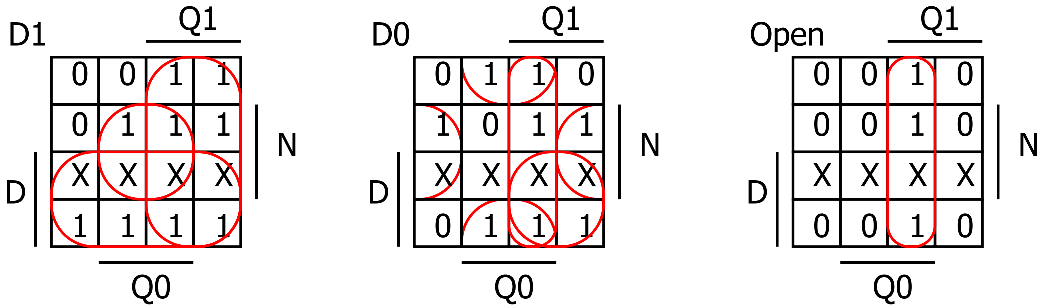
Example: Vending Machine (cont’d)

- 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

Example: Vending Machine (cont'd)

- Mapping to Logic



D1

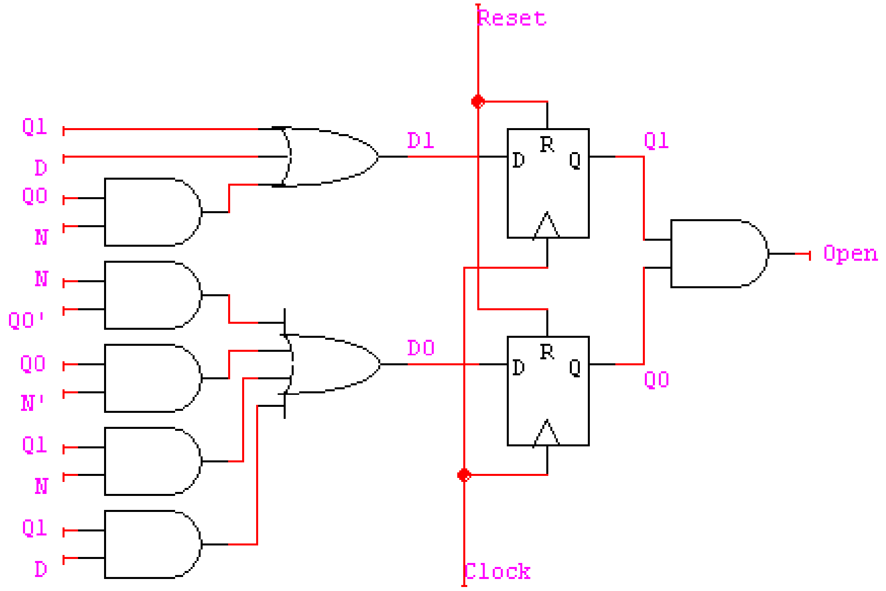
0	0	1	1
0	1	1	1
X	X	X	X
1	1	1	1

D0

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

Open

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



D1 = Q1 + D + Q0 N

D0 = Q0′ N + Q0 N′ + Q1 N + Q1 D

OPEN = Q1 Q0

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

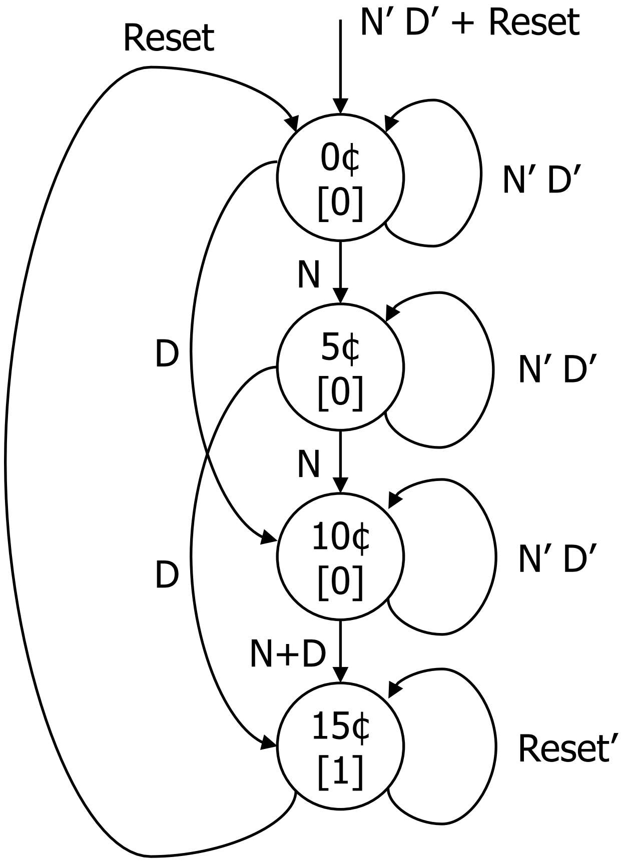
OPEN = Q3

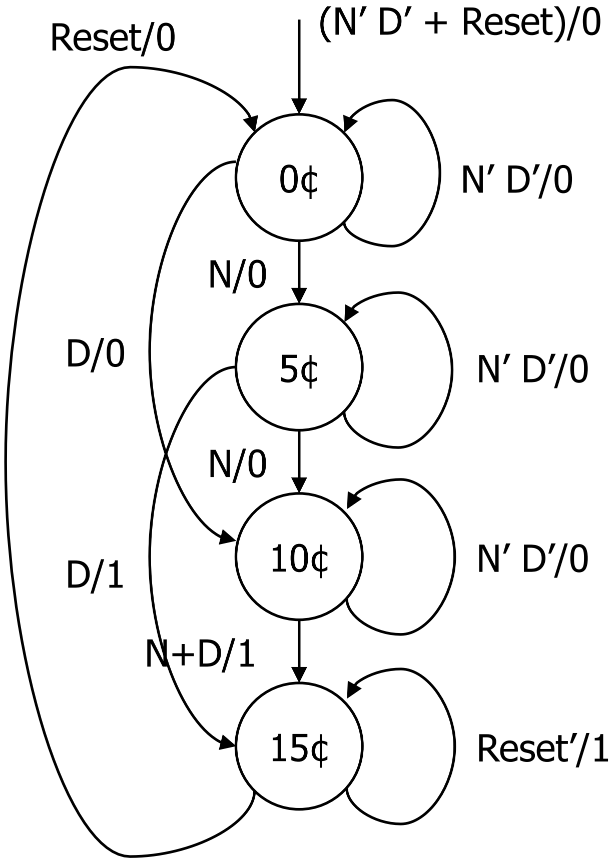
Equivalent Mealy and Moore State Diagrams

- Moore machine
  - outputs associated with state

Mealy machine

outputs associated with transitions





Example: Traffic Light Controller

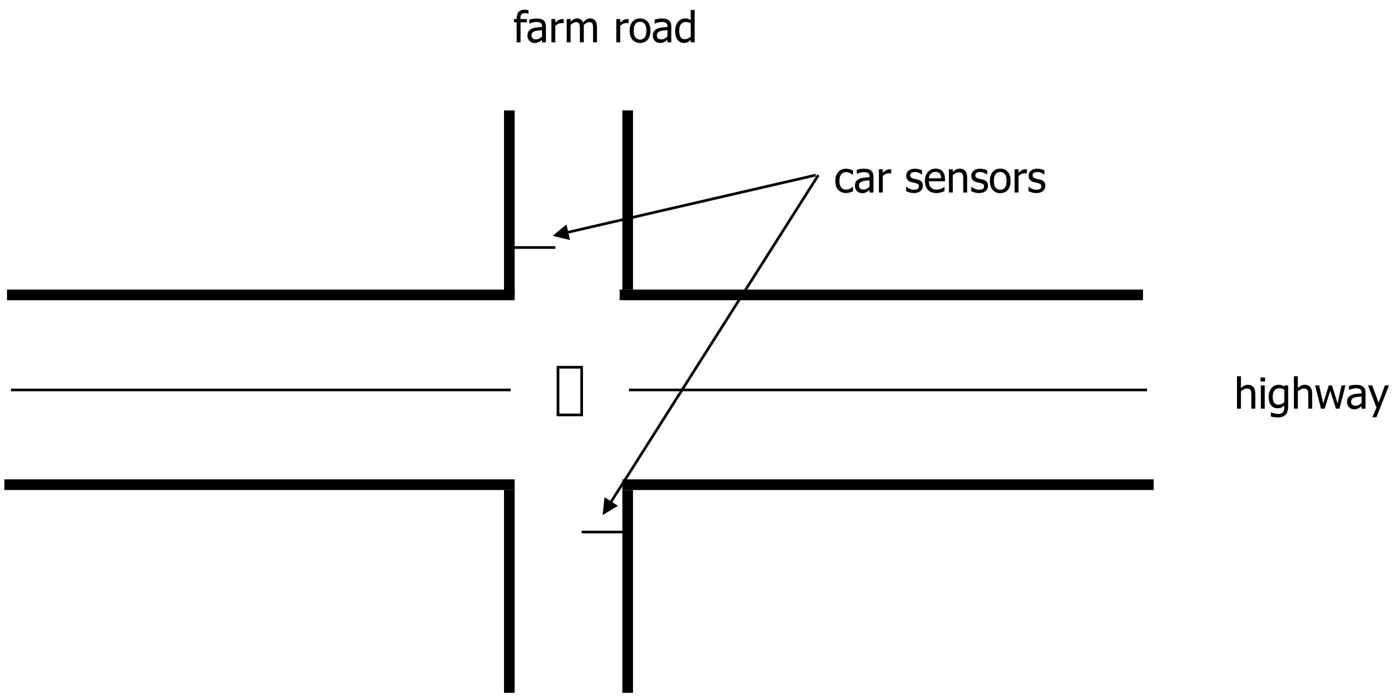
- A busy highway is intersected by a little used farmroad
- Detectors C sense the presence of cars waiting on the farmroad
  - with no car on farmroad, light remain green in highway direction
  - if vehicle on farmroad, highway lights go from Green to Yellow to Red, allowing the farmroad lights to become green
  - these stay green only as long as a farmroad car is detected but never longer than a set interval
  - when these are met, farm lights transition from Green to Yellow to Red, allowing highway to return to green
  - even if farmroad vehicles are waiting, highway gets at least a set interval as green

Example: Traffic Light Controller

- Assume you have an interval timer that generates:
  - a short time pulse (TS) and
  - a long time pulse (TL),
  - in response to a set (ST) signal.
  - TS is to be used for timing yellow lights and TL for green lights

Example: Traffic Light Controller (cont')

- Highway/farm road intersection



Example: Traffic Light Controller (cont')

- Tabulation of Inputs and Outputs

inputs	description	outputs	description
reset	place FSM in initial state	HG, HY, HR	assert green/yellow/red highway lights
C	detect vehicle on the farm road	FG, FY, FR	assert green/yellow/red highway lights
TS	short time interval expired	ST	start timing a short or long interval
TL	long time interval expired		

- Tabulation of unique states – some light configurations imply others

state	description
S0	highway green (farm road red)
S1	highway yellow (farm road red)
S2	farm road green (highway red)
S3	farm road yellow (highway red)

Example: Traffic Light Controller (cont')

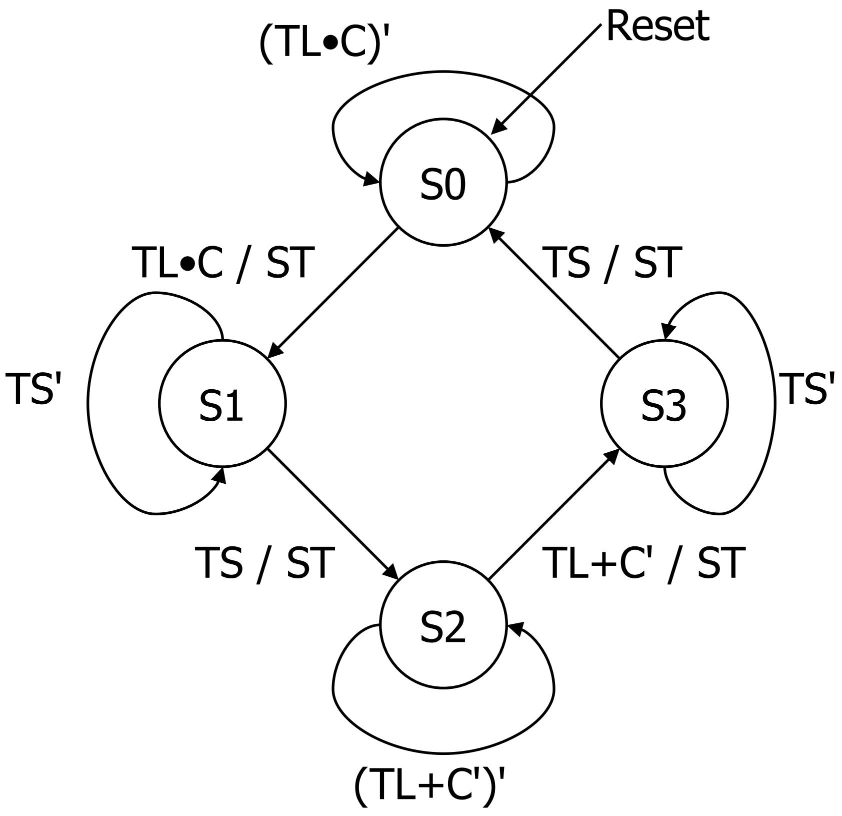
- State Diagram

S0: HG

S1: HY

S2: FG

S3: FY



Example: Traffic Light Controller (cont’)

- Generate state table with symbolic states
- Consider state assignments

output encoding – similar problem
to state assignment
(Green = 00, Yellow = 01, Red = 10)

Inputs			Present State	Next State	Outputs		
C	TL	TS			ST	H	F
0	–	–	HG	HG	0	Green	Red
–	0	–	HG	HG	0	Green	Red
1	1	–	HG	HY	1	Green	Red
–	–	0	HY	HY	0	Yellow	Red
–	–	1	HY	FG	1	Yellow	Red
1	0	–	FG	FG	0	Red	Green
0	–	–	FG	FY	1	Red	Green
–	1	–	FG	FY	1	Red	Green
–	–	0	FY	FY	0	Red	Yellow
–	–	1	FY	HG	1	Red	Yellow

SA1:	HG = 00	HY = 01	FG = 11	FY = 10	
SA2:	HG = 00	HY = 10	FG = 01	FY = 11	
SA3:	HG = 0001	HY = 0010	FG = 0100	FY = 1000	(one-hot)

Logic for Different State Assignments

- SA1

  NS1 = C•TL'•PS1•PS0 + TS•PS1'•PS0 + TS•PS1•PS0' + C'•PS1•PS0 + TL•PS1•PS0
  NS0 = C•TL•PS1'•PS0' + C•TL'•PS1•PS0 + PS1'•PS0

  ST = C•TL•PS1'•PS0' + TS•PS1'•PS0 + TS•PS1•PS0' + C'•PS1•PS0 + TL•PS1•PS0
  H1 = PS1 H0 = PS1'•PS0
  F1 = PS1' F0 = PS1•PS0'

- SA2

  NS1 = C•TL•PS1' + TS'•PS1 + C'•PS1'•PS0
  NS0 = TS•PS1•PS0' + PS1'•PS0 + TS'•PS1•PS0

  ST = C•TL•PS1' + C'•PS1'•PS0 + TS•PS1
  H1 = PS0 H0 = PS1•PS0'
  F1 = PS0' F0 = PS1•PS0

- SA3

  NS3 = C'•PS2 + TL•PS2 + TS'•PS3 NS2 = TS•PS1 + C•TL'•PS2
  NS1 = C•TL•PS0 + TS'•PS1 NS0 = C'•PS0 + TL'•PS0 + TS•PS3

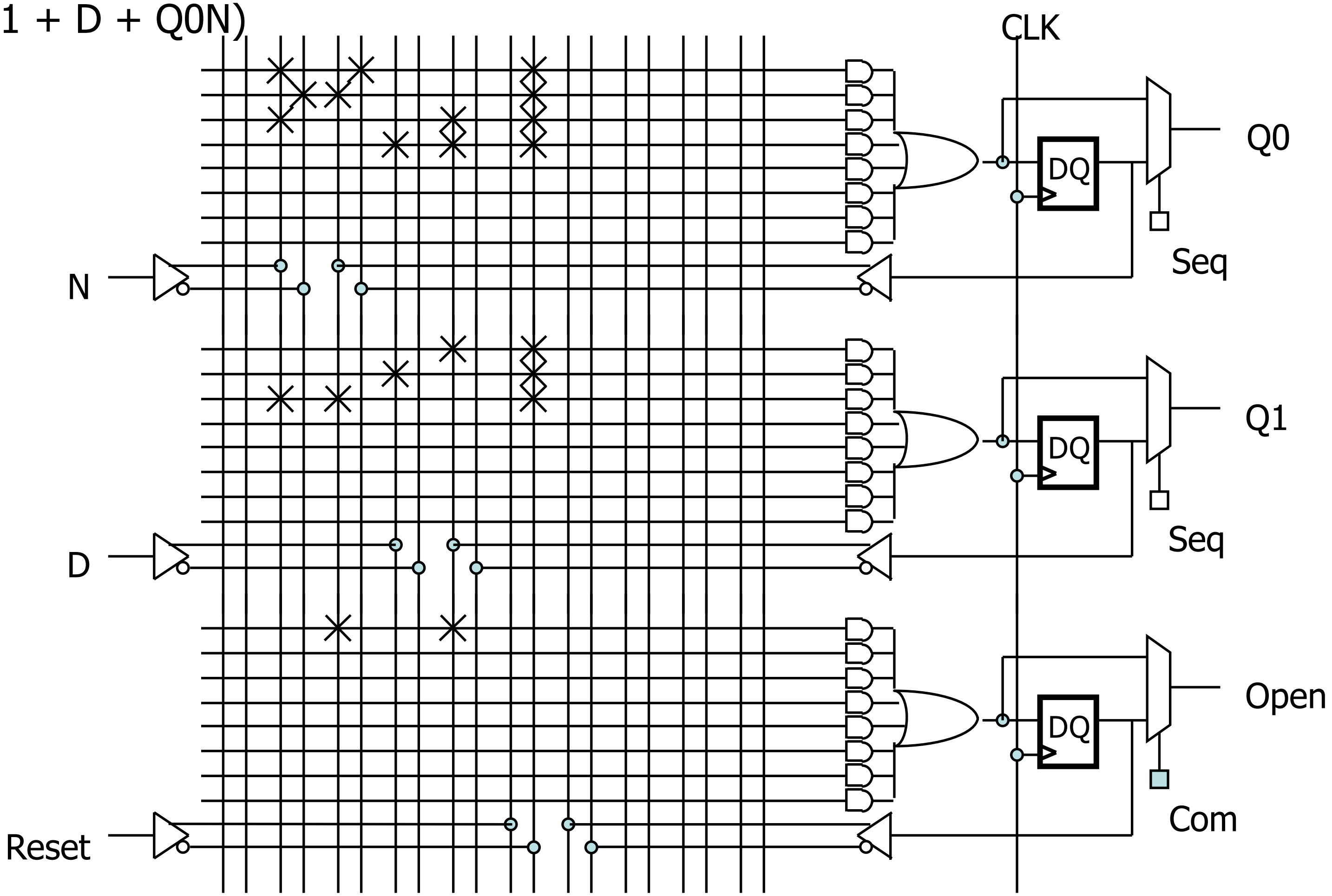
  ST = C•TL•PS0 + TS•PS1 + C'•PS2 + TL•PS2 + TS•PS3
  H1 = PS3 + PS2 H0 = PS1
  F1 = PS1 + PS0 F0 = PS3

Vending Machine Example (PLD mapping)

D0 = reset'(Q0'N + Q0N' + Q1N + Q1D)

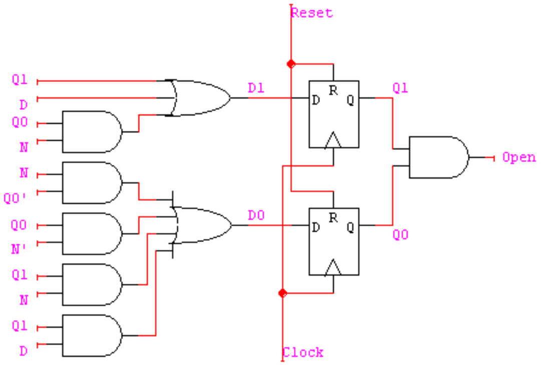
D1 = reset'(Q1 + D + Q0N)

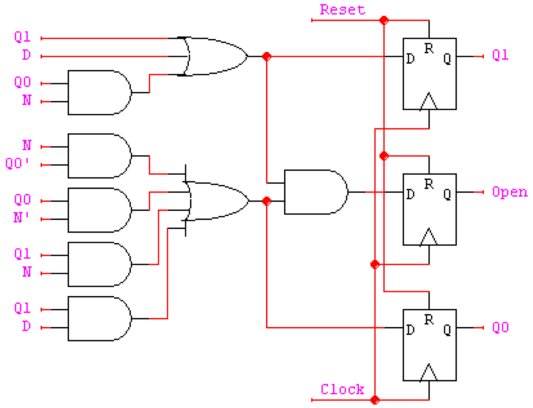
OPEN = Q1Q0



Vending Machine (cont’d)

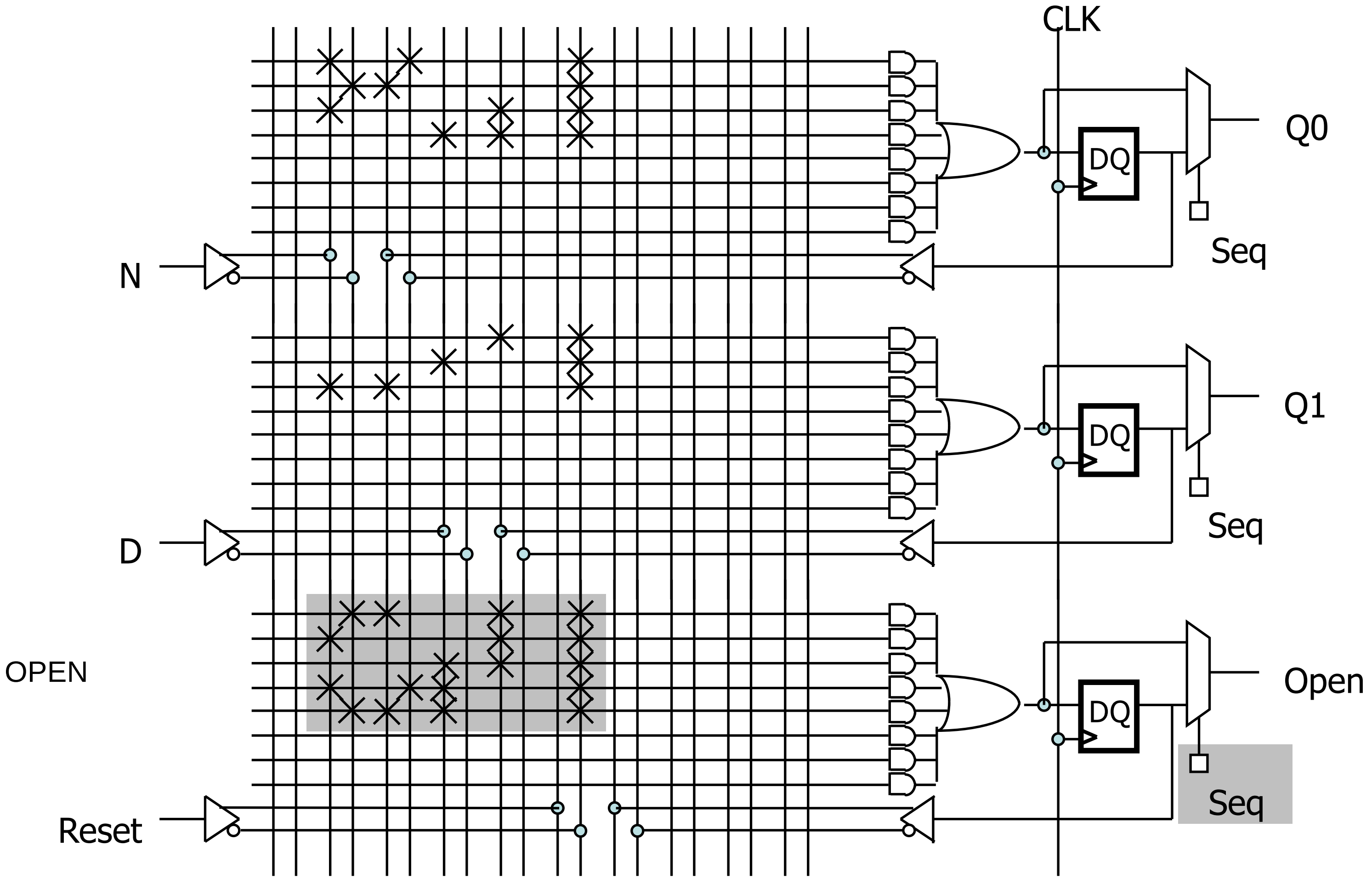
- OPEN = Q1Q0 creates a combinational delay after Q1 and Q0 change
- This can be corrected by retiming, i.e., move flip-flops and logic through each other to improve delay
- OPEN = reset'(Q1 + D + Q0N)(Q0'N + Q0N' + Q1N + Q1D)
  = reset'(Q1Q0N' + Q1N + Q1D + Q0'ND + Q0N'D)
- Implementation now looks like a synchronous Mealy machine
  - Common for programmable devices to have FF at end of logic





Vending Machine (Retimed PLD Mapping)

OPEN = reset'(Q1Q0N' + Q1N + Q1D + Q0'ND + Q0N'D)



STATE MINIMIZATION

Finite State Machine Optimization

- Should look over this one
- Make sure you know the motivation for this
  - State Minimization
    - Try to merge state with same behavior so total # of states are less
  - State Encoding
    - Less combinational logic involved in implementing if correct state encoding is chosen.
    - Less Flip flops used if a more dense encoding is used.
      - # of flip flop could range from log2(n) to n
- Make sure you know the algorithm for both

Finite State Machine Optimization

- State Minimization
  - Fewer states require fewer state bits
  - Fewer bits require fewer logic equations
- Encodings: State, Inputs, Outputs
  - State encoding with fewer bits has fewer equations to implement
    - However, each may be more complex
  - State encoding with more bits (e.g., one-hot) has simpler equations
    - Complexity directly related to complexity of state diagram
  - Input/output encoding may or may not be under designer control

Algorithmic Approach to State Minimization

- Goal – identify and combine states that have equivalent behavior
- Equivalent States:
  - Same output
  - For all input combinations, states transition to same or equivalent states
- Algorithm Sketch
  - 1. Place all states in one set
  - 2. Initially partition set based on output behavior
  - 3. Successively partition resulting subsets based on next state transitions
  - 4. Repeat (3) until no further partitioning is required
    - states left in the same set are equivalent
  - Polynomial time procedure

State Minimization

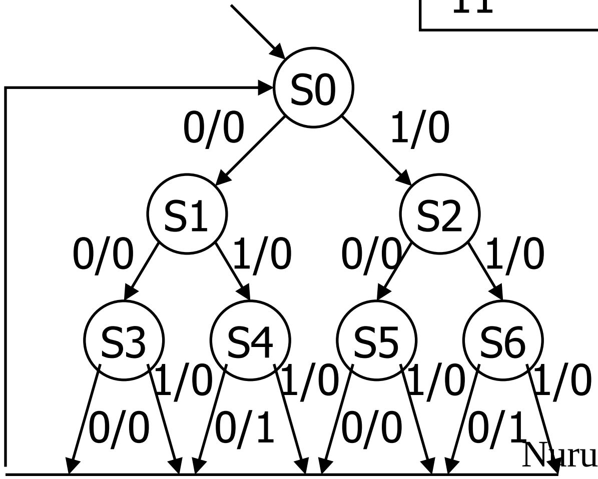
2 Methods:

1. Row Matching
2. Implication Chart

Implication Chart Method

- Sequence detector for 010 or 110

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



Implication Chart Method

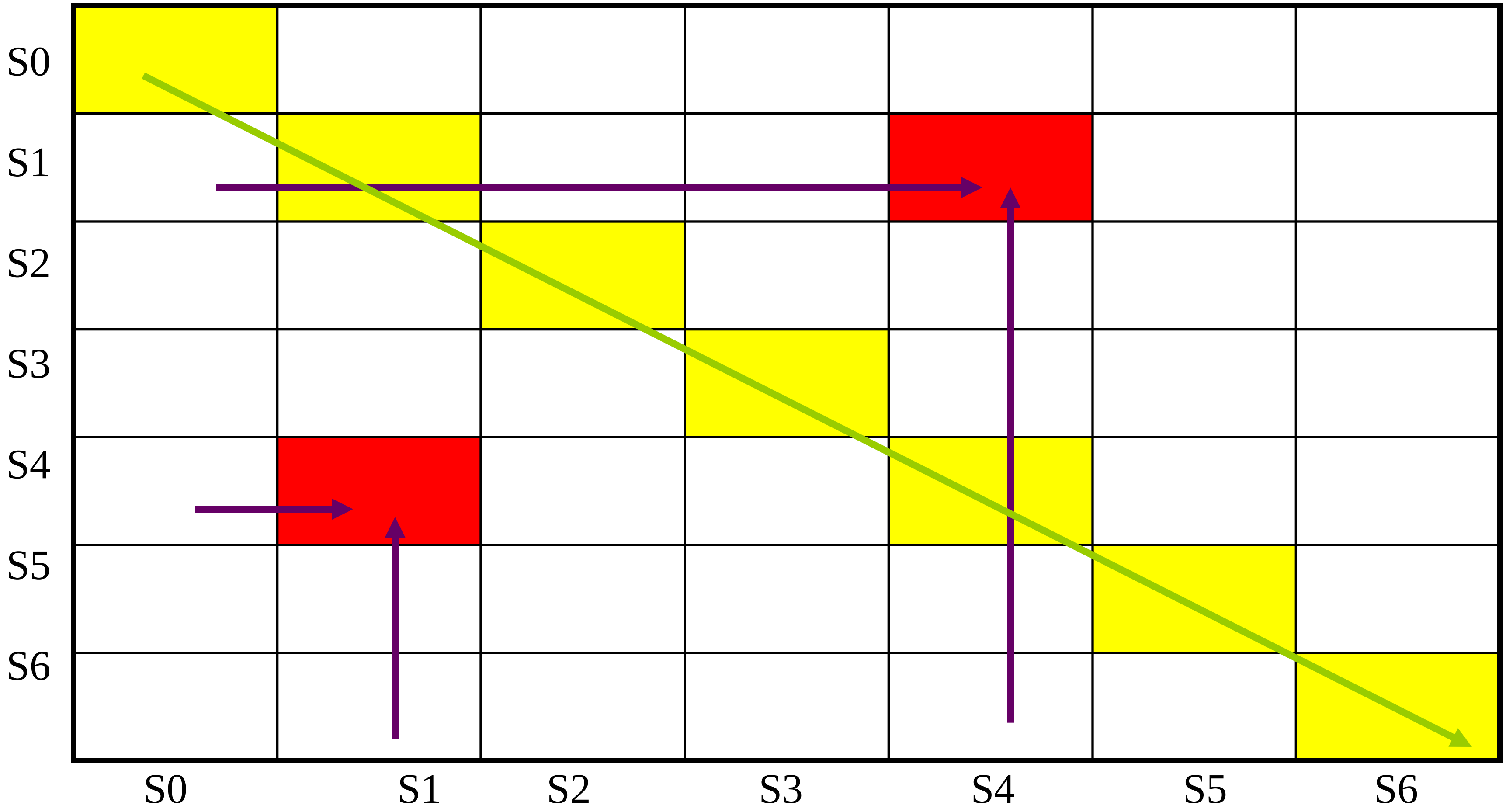


Figure 1 : Chart with entry of every pairs of states

Implication Chart Method

	S0	S1	S2	S3	S4	S5	S6
S0	🟦	🟦	🟦	🟦	🟦	🟦	🟦
S1		🟦	🟦	🟦	🟦	🟦	🟦
S2			🟦	🟦	🟦	🟦	🟦
S3				🟦	🟦	🟦	🟦
S4					🟦	🟦	🟦
S5						🟦	🟦
S6							🟦

Implication Chart Method

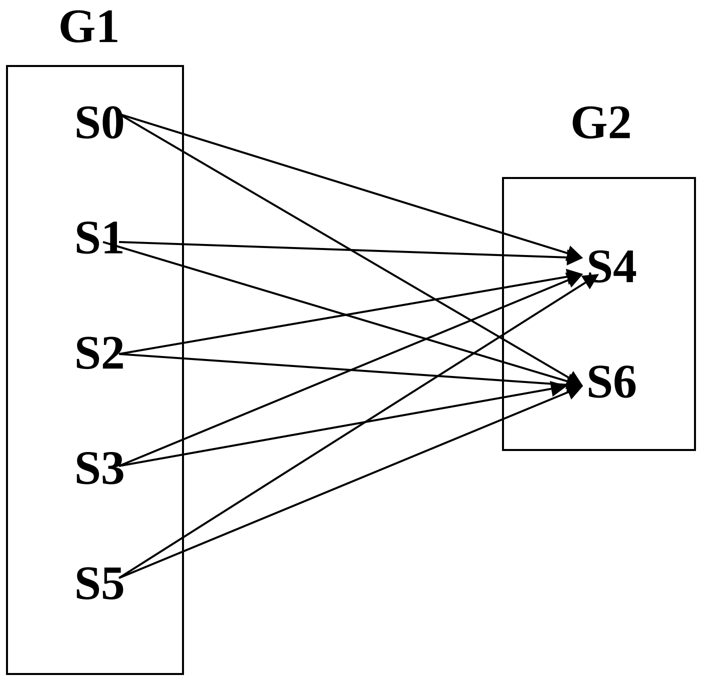
IMPLIED STATE PAIRS

S0							
S1	S1-S3 S2-S4						
S2	S1-S5 S2-S6	S3-S5 S4-S6					
S3	S1-S0 S2-S0	S3-S0 S4-S0	S5-S0 S6-S0				
S4	X	X	X	X			
S5	S1-S0 S2-S0	S3-S0 S4-S0	S5-S0 S6-S0	S0-S0 S0-S0	X		
S6	X	X	X	X	S0-S0 S0-S0	X	
	S0	S1	S2	S3	S4	S5	S6

S0, S1, S2, S3, S5➔same output (group1)

S4, S6 ➔ same output (group2) any combination across g1 and g2 indicate by X

Implied State Pairs



S0, S1, S2, S3, S5➔same output (group1)
S4, S6 ➔ same output (group2) any combination across g1 and g2 indicate by X

Implied State Pairs

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

Rechecked Implied State Pairs

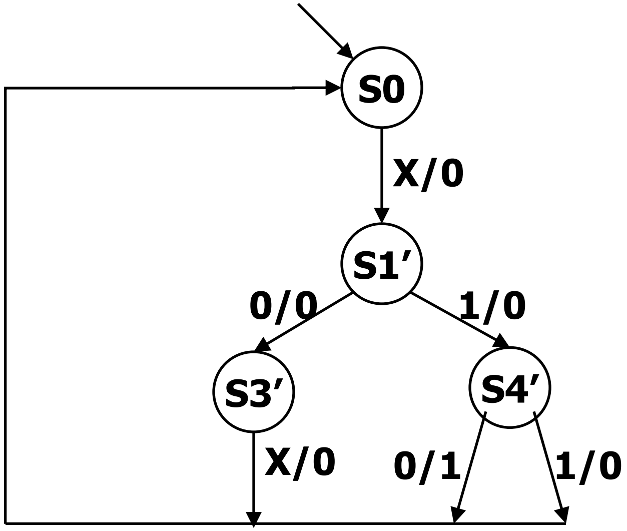
	S0	S1	S2	S3	S4	S5	S6
S0							
S1	X						
S2	X	S3-S5 S4-S6					
S3	X	X	X				
S4	X	X	X	X			
S5	X	X	X	S0-S0 S0-S0	X		
S6	X	X	X	X	S0-S0 S0-S0	X	

IF square Si and Sj contains the ISP Sm-Sn and square Sm,Sn contains an X, then Mark Si,Sj with an X as well.

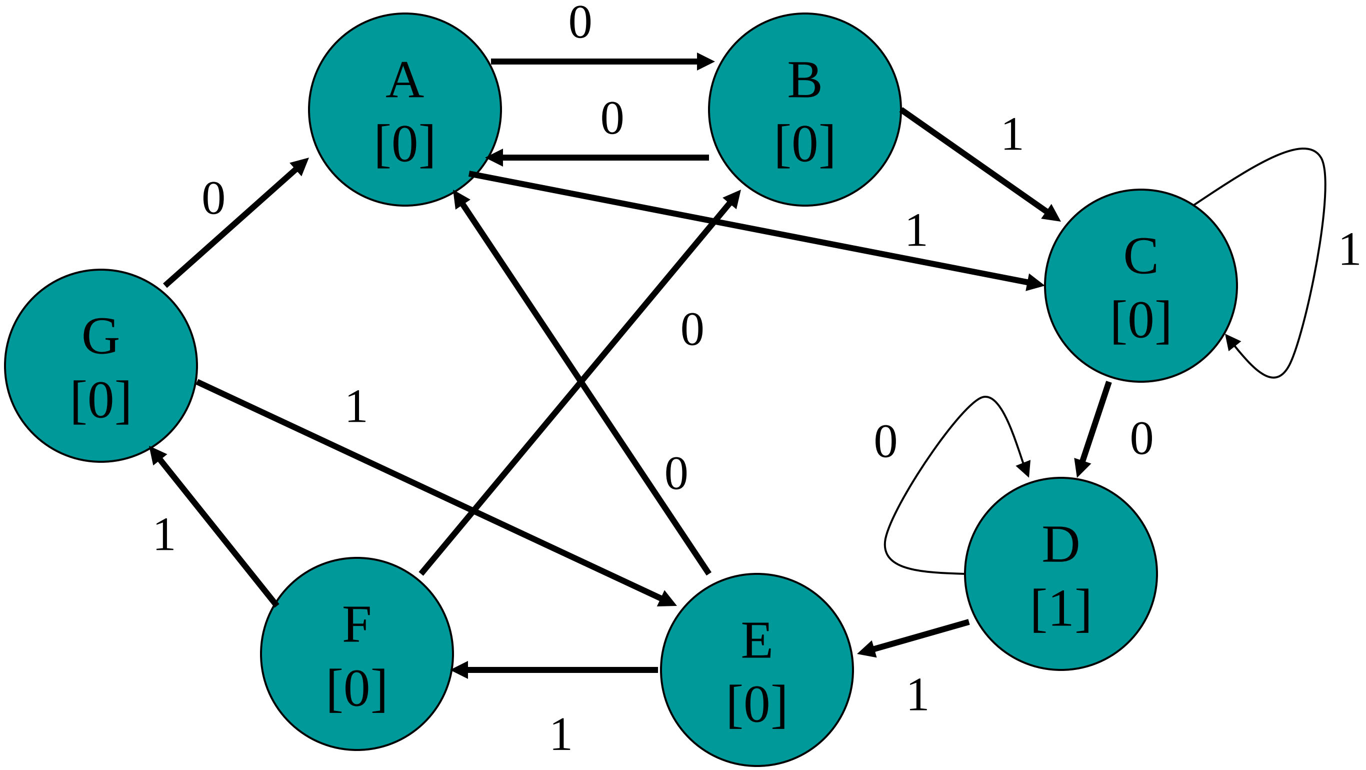
Implication Chart Method

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S0	S1'	S1'	0	0
0 + 1	S1'	S3'	S4'	0	0
X0	S3'	S0	S0	0	0
X1	S4'	S0	S0	1	0

Simplified Table



Another Example



NurulHazlina/BEE2243/FSM

Continue

Present state	Next state		output
	0	1	
A	B	C	0
B	A	C	0
C	D	C	0
D	D	E	1
E	A	F	0
F	B	G	0
G	A	E	0

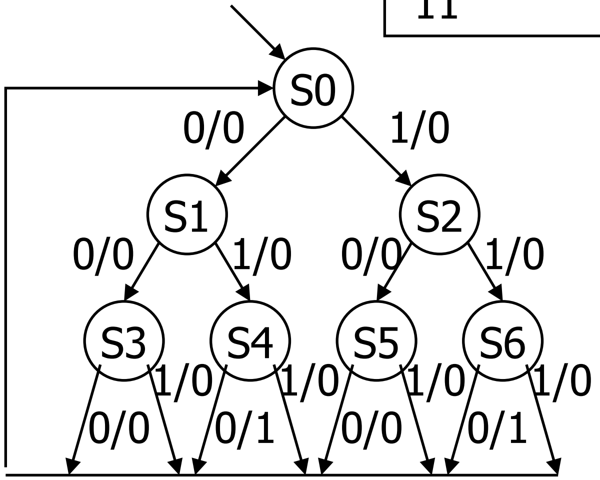
Divide into 2 groups;

A,B,C,E,F,G and D based on Output value

State Minimization Example

- Sequence Detector for 010 or 110

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



Method of Successive Partitions Row Matching

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output 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

(S0 S1 S2 S3 S4 S5 S6)

(S0 S1 S2 S3 S5) (S4 S6)

(S0 S3 S5) (S1 S2) (S4 S6)

(S0) (S3 S5) (S1 S2) (S4 S6)

S1 is equivalent to S2

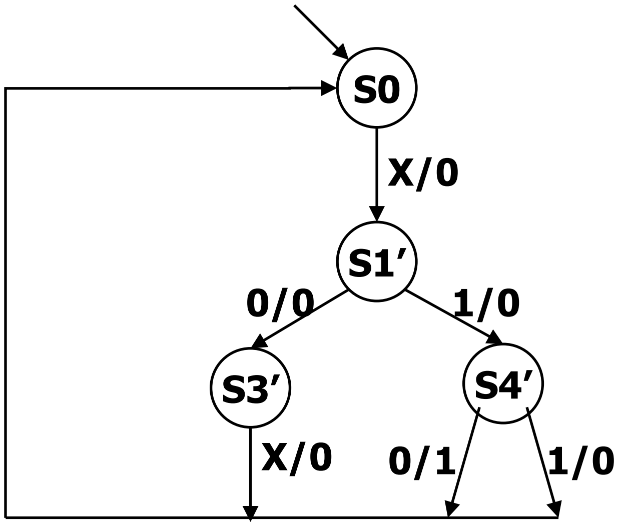
S3 is equivalent to S5

S4 is equivalent to S6

Minimized FSM

- State minimized sequence detector for 010 or 110

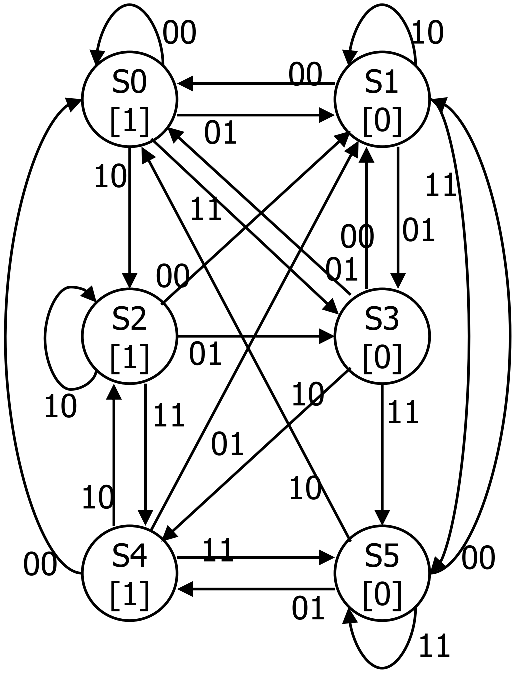
Input Sequence	Present State	Next State		Output	
		X=0	X=1	X=0	X=1
Reset	S0	S1'	S1'	0	0
0 + 1	S1'	S3'	S4'	0	0
X0	S3'	S0	S0	0	0
X1	S4'	S0	S0	1	0



Implication Chart

More Complex State Minimization

- Multiple input example



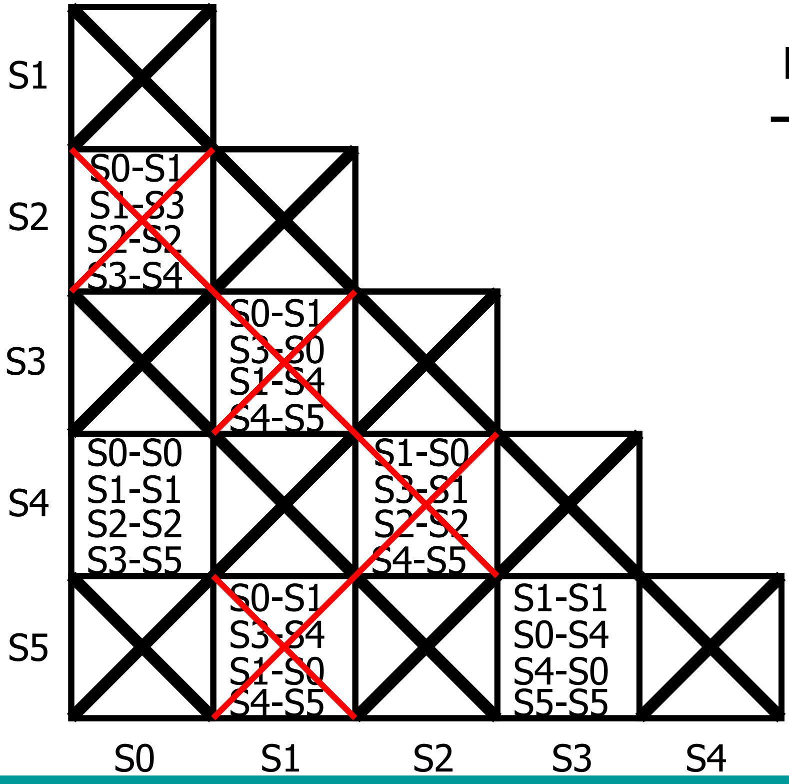
inputs here

present state	next state				output
	00	01	10	11	
S0	S0	S1	S2	S3	1
S1	S0	S3	S1	S4	0
S2	S1	S3	S2	S4	1
S3	S1	S0	S4	S5	0
S4	S0	S1	S2	S5	1
S5	S1	S4	S0	S5	0

symbolic state transition table

Minimized FSM

- Implication Chart Method
  - Cross out incompatible states based on outputs
  - Then cross out more cells if indexed chart entries are already crossed out



	S0	S1	S2	S3	S4
S1	☒				
S2	~~S0-S1~~ ~~S1-S3~~ ~~S2-S2~~ ~~S3-S4~~	☒			
S3	☒	~~S0-S1~~ ~~S3-S0~~ ~~S1-S4~~ ~~S4-S5~~	☒		
S4	S0-S0 S1-S1 S2-S2 S3-S5	☒	~~S1-S0~~ ~~S3-S1~~ ~~S2-S2~~ ~~S4-S5~~	☒	
S5	☒	~~S0-S1~~ ~~S3-S4~~ ~~S1-S0~~ ~~S4-S5~~	☒	S1-S1 S0-S4 S4-S0 S5-S5	☒

present state	next state 00	01	10	11	output
S0'	S0'	S1	S2	S3'	1
S1	S0'	S3'	S1	S3'	0
S2	S1	S3'	S2	S0'	1
S3'	S1	S0'	S0'	S3'	0

minimized state table
(S0==S4) (S3==S5)

Minimizing Incompletely Specified FSMs

- Equivalence of states is transitive when machine is fully specified
- But its not transitive when don't cares are present

e.g.,

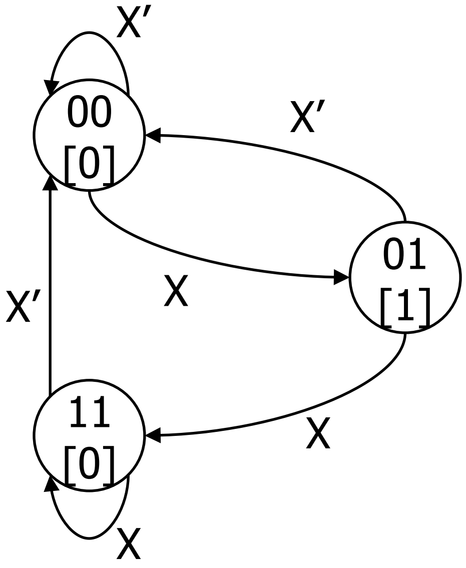
state	output
S0	– 0
S1	1 –
S2	– 1

S1 is compatible with both S0 and S2 but S0 and S2 are incompatible

- No polynomial time algorithm exists for determining best grouping of states into equivalent sets that will yield the smallest number of final states

Minimizing States May Not Yield Best Circuit

- Example: edge detector - outputs 1 when last two input changes from 0 to 1



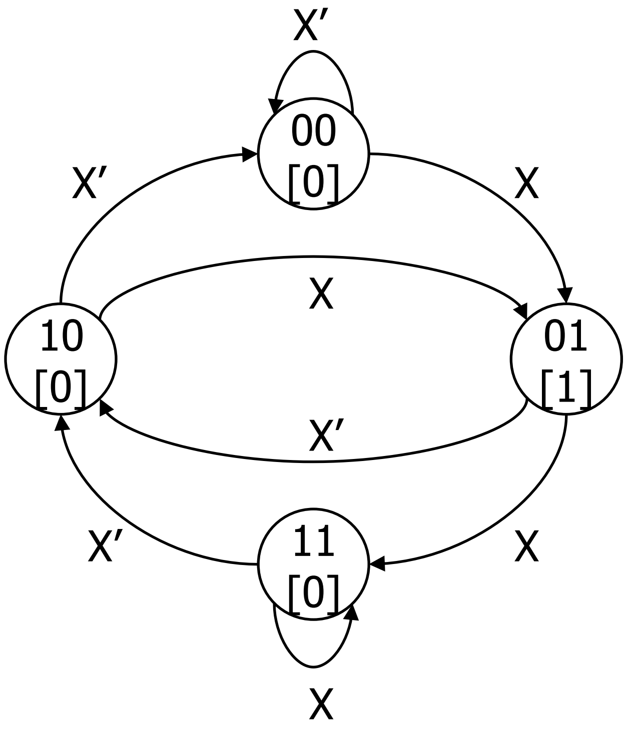
X	Q_1	Q_0	Q_1^+	Q_0^+
0	0	0	0	0
0	0	1	0	0
0	1	1	0	0
1	0	0	0	1
1	0	1	1	1
1	1	1	1	1
–	1	0	0	0

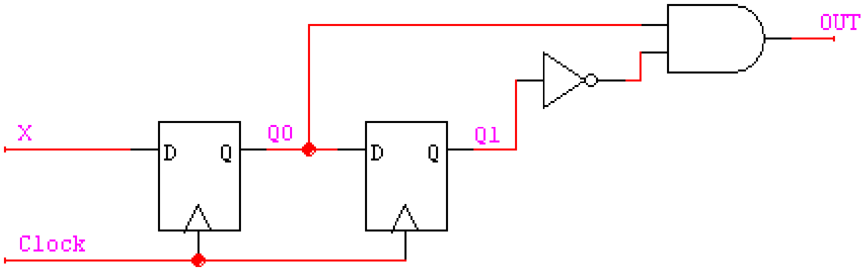
$$Q_1^+ = X\ (Q_1 \text{ xor } Q_0)$$

$$Q_0^+ = X\ Q_1'\ Q_0'$$

Another Implementation of Edge Detector

- "Ad hoc" solution - not minimal but cheap and fast





Sequential Logic Implementation 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
- Implementation of sequential logic
  - State minimization
  - State assignment
  - Support in programmable logic devices

Finite State Machine Concept

- FSMs are the decision making logic of digital designs
- Partitioning designs into datapath and control elements
- When inputs are sampled and outputs asserted

Basic Design Approach: 4-step Design Process

Implementation Examples and Case Studies

- Finite-string pattern recognizer – mini project a
- Traffic light controller – mini project b
- Door combination lock – mini project c

SEQUENTIAL LOGIC EXAMPLES

General FSM Design Procedure

1. Determine inputs and outputs
2. Determine possible states of machine
   - State minimization
3. Encode states and outputs into a binary code
   - State assignment or state encoding
   - Output encoding
   - Possibly input encoding (if under our control)
4. Realize logic to implement functions for states and outputs
   - Combinational logic implementation and optimization
   - Choices in steps 2 and 3 have large effect on resulting logic

Finite String Pattern Recognizer (Step 1)

MINI PROJECT 3a

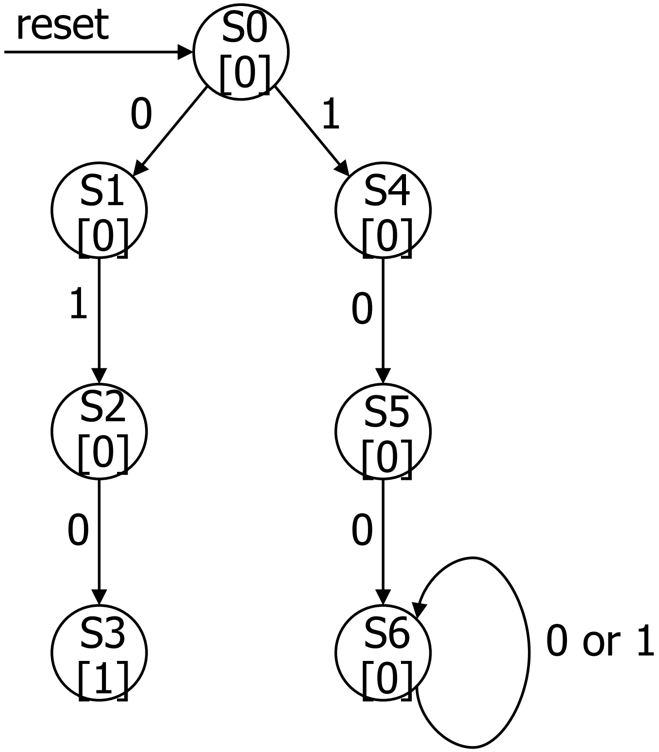
- Finite String Pattern Recognizer
  - One input (X) and one output (Z)
  - Output is asserted whenever the input sequence …010… has been observed, as long as the sequence 100 has never been seen

- Step 1: Understanding the Problem Statement
  - Sample input/output behavior:

```
X:  0 0 1 0 1 0 1 0 0 1 0 …
Z:  0 0 0 1 0 1 0 1 0 0 0 …

X:  1 1 0 1 1 0 1 0 0 1 0 …
Z:  0 0 0 0 0 0 0 1 0 0 0 …
```

Finite String Pattern Recognizer (Step 2)

- Step 2: Draw State Diagram
  - For the strings that must be recognized, i.e., 010 and 100
  - Moore implementation

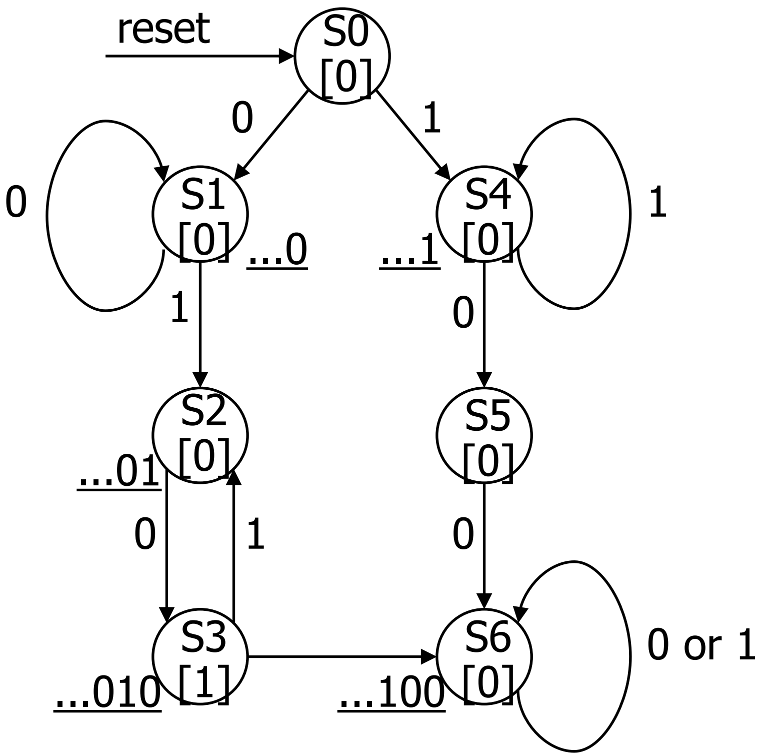


Finite String Pattern Recognizer (Step 2, cont'd)

- Exit conditions from state S3: have recognized …010
  - If next input is 0 then have …0100 = ...100 (state S6)
  - If next input is 1 then have …0101 = …01 (state S2)

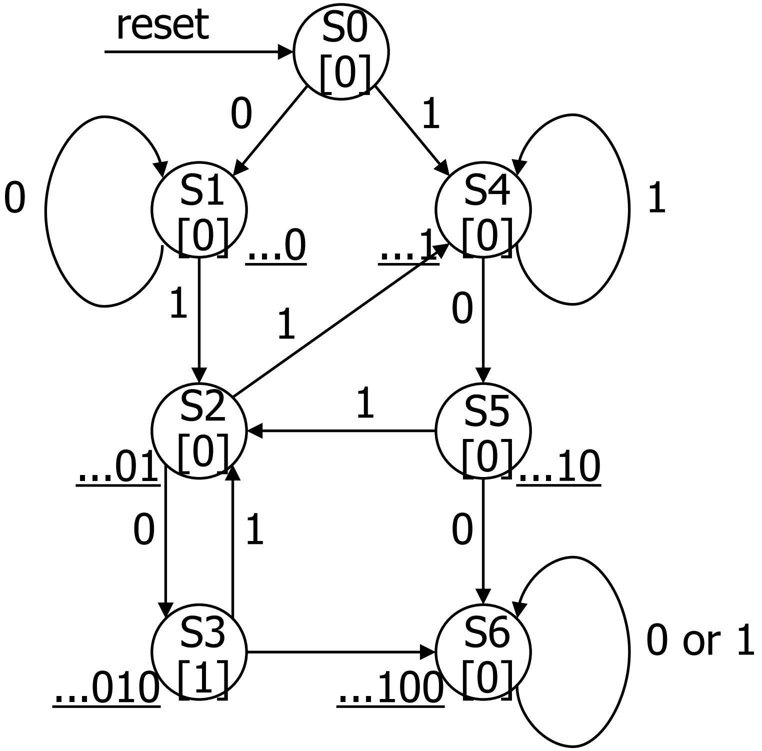
Exit conditions from S1:
recognizes strings of form …0
(no 1 seen);
loop back to S1 if input is 0

Exit conditions from S4:
recognizes strings of form …1
(no 0 seen);
loop back to S4 if input is 1



Finite String Pattern Recognizer (Step 2, cont'd)

- S2 and S5 still have incomplete transitions
  - S2 = ...01; If next input is 1,
    then string could be prefix of (01)1(00)
    S4 handles just this case
  - S5 = ...10; If next input is 1,
    then string could be prefix of (10)1(0)
    S2 handles just this case
- Reuse states as much as possible
  - Look for same meaning
  - State minimization leads to smaller number of bits to represent states
- Once all states have complete set of transitions we have final state diagram



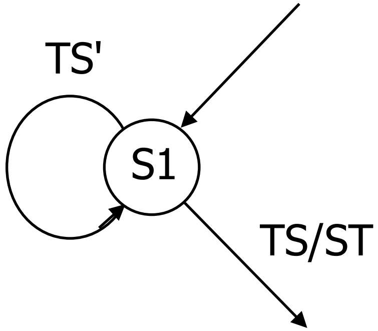
Finite String Pattern Recognizer

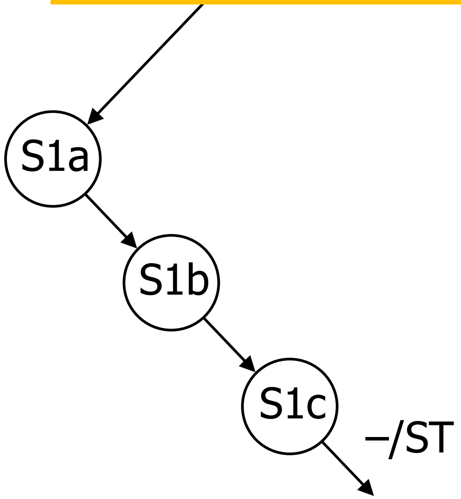
- Review of Process
  - Understanding problem
    - Write down sample inputs and outputs to understand specification
  - Derive a state diagram
    - Write down sequences of states and transitions for sequences to be recognized
  - Minimize number of states
    - Add missing transitions; reuse states as much as possible
  - State assignment or encoding
    - Encode states with unique patterns
  - Simulate realization
    - Verify I/O behavior of your state diagram to ensure it matches specification

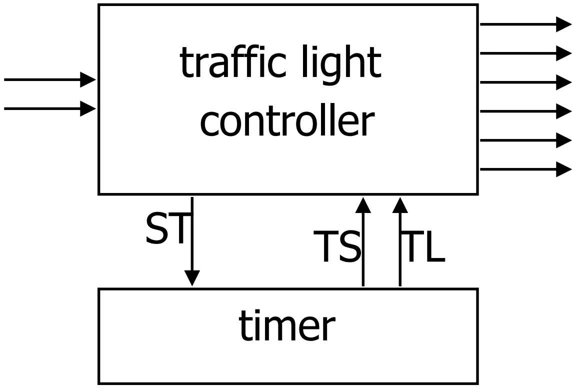
Traffic Light Controller as Two Communicating FSMs

MINI PROJECT 3b

- Without Separate Timer
  - S0 would require 7 states
  - S1 would require 3 states
  - S2 would require 7 states
  - S3 would require 3 states
  - S1 and S3 have simple transformation
  - S0 and S2 would require many more arcs
    - C could change in any of seven states
- By Factoring Out Timer
  - Greatly reduce number of states
    - 4 instead of 20
  - Counter only requires seven or eight states
    - 12 total instead of 20

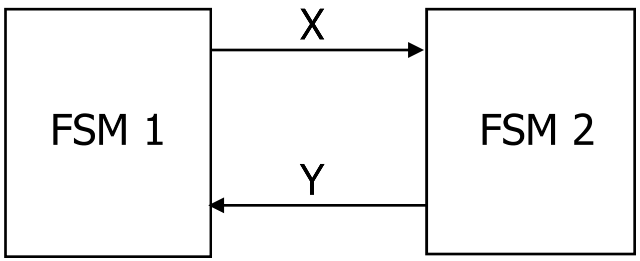


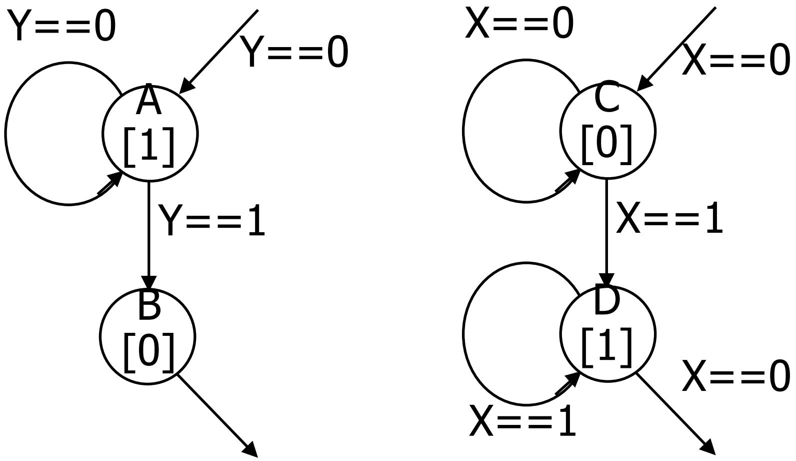


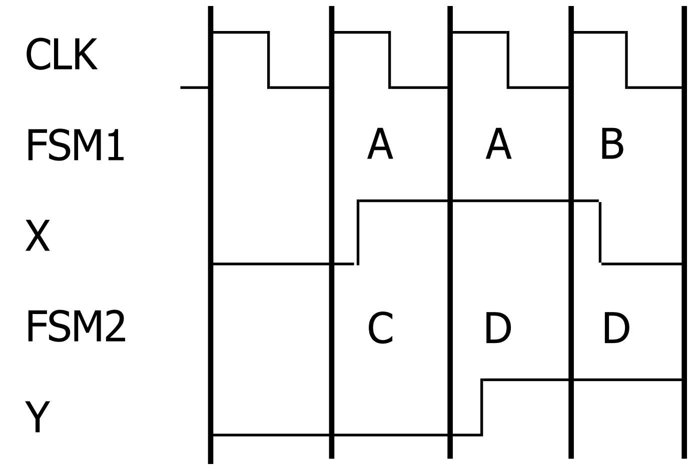


Communicating Finite State Machines

- One machine's output is another machine's input



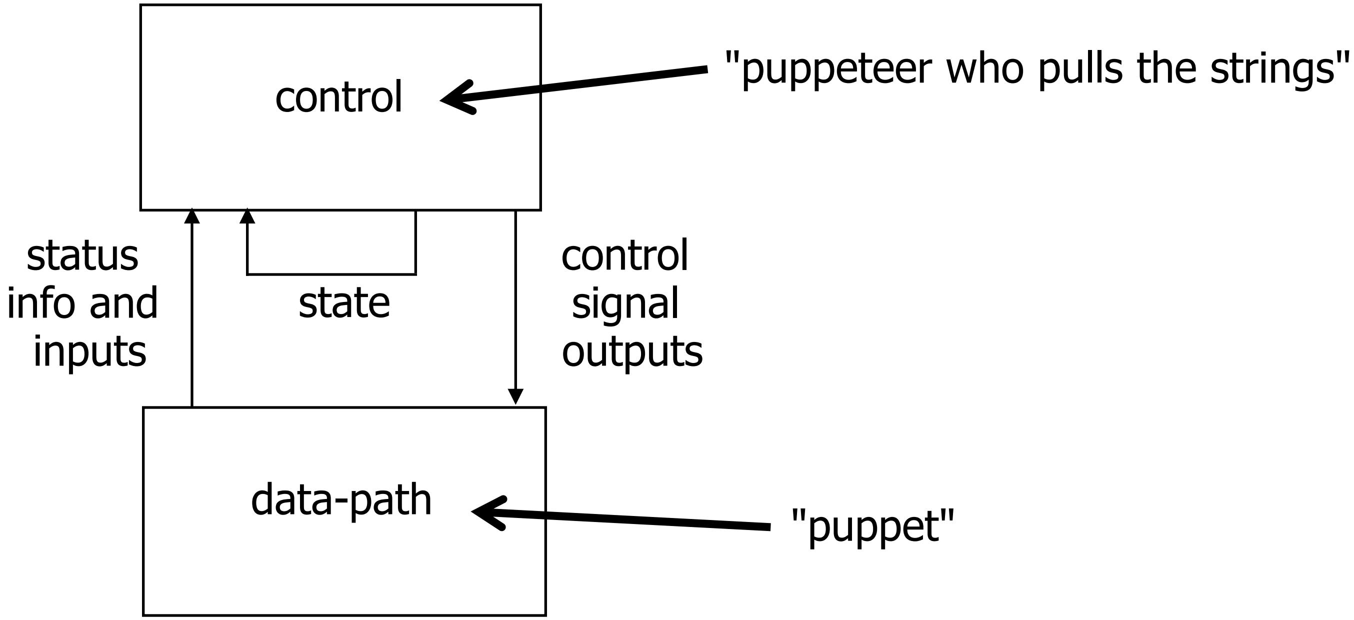




machines advance in lock step
initial inputs/outputs: X = 0, Y = 0

Datapath and Control

- Digital hardware systems = data-path + control
  - Datapath: registers, counters, combinational functional units (e.g., ALU), communication (e.g., busses)
  - Control: FSM generating sequences of control signals that instructs datapath what to do next



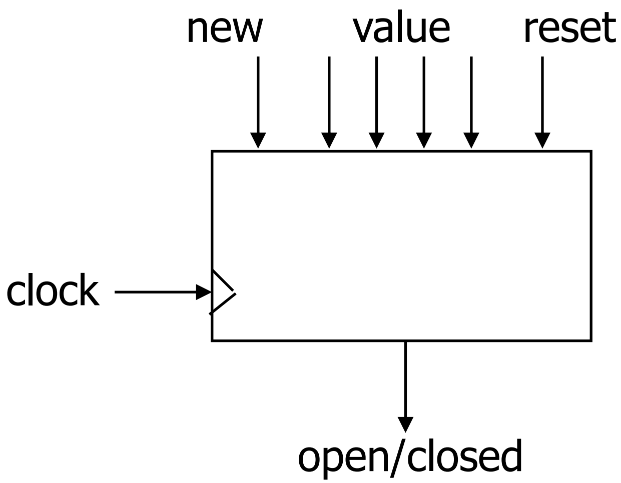
Digital Combinational Lock

MINI PROJECT 3c

- Door Combination Lock:
  - Punch in 3 values in sequence and the door opens; if there is an error the lock must be reset; once the door opens the lock must be reset
  - Inputs: sequence of input values, reset
  - Outputs: door open/close
  - Memory: must remember combination or always have it available
  - Open questions: how do you set the internal combination?
    - Stored in registers (how loaded?)
    - Hardwired via switches set by user

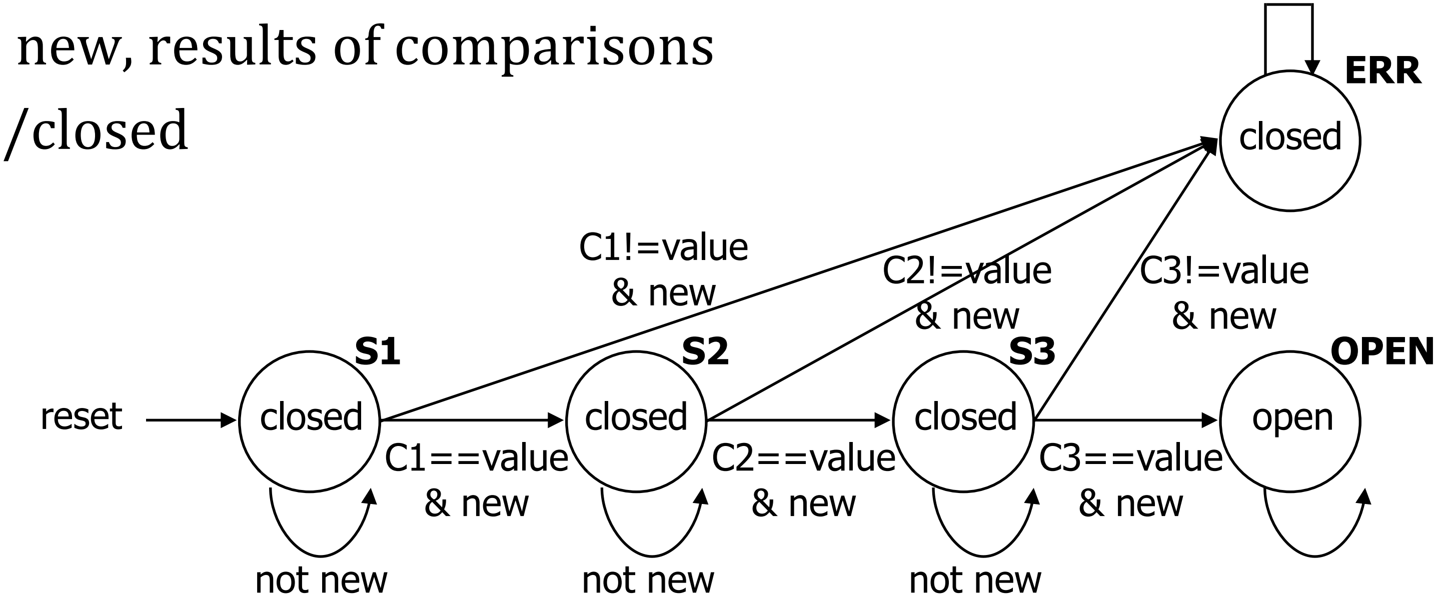
Determining Details of the Specification

- How many bits per input value?
- How many values in sequence?
- How do we know a new input value is entered?
- What are the states and state transitions of the system?



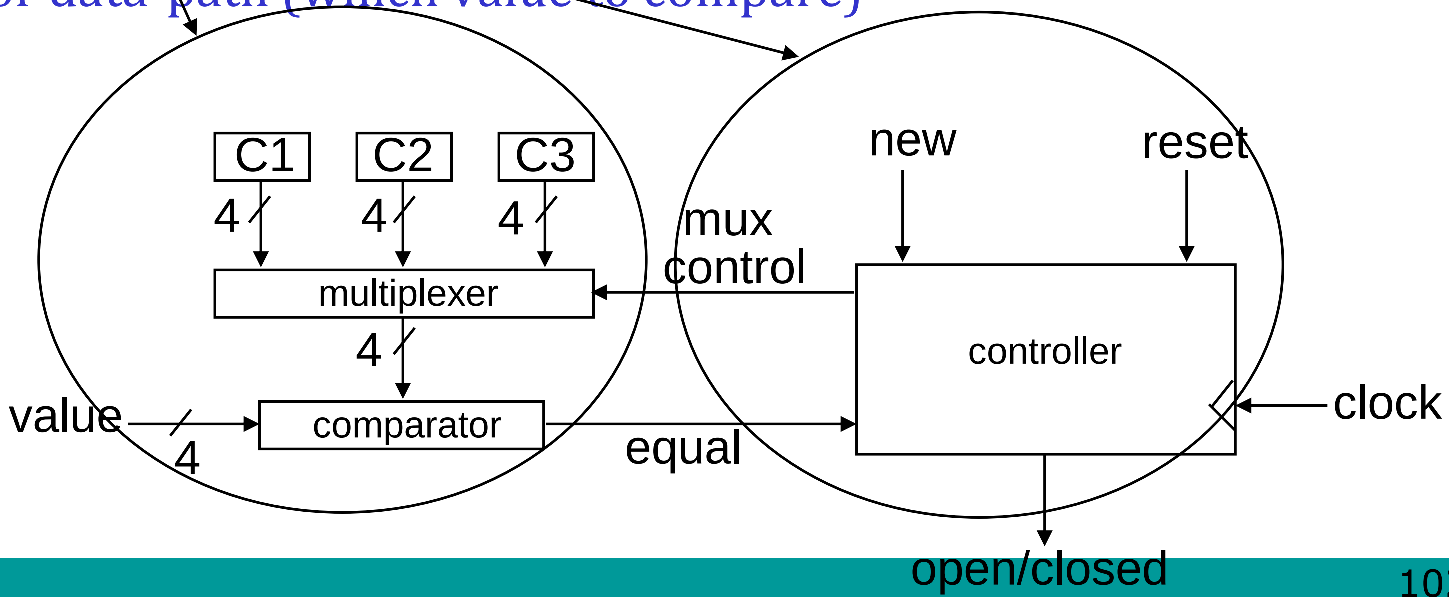
Digital Combination Lock State Diagram

- States: 5 states
  - Represent point in execution of machine
  - Each state has outputs
- Transitions: 6 from state to state, 5 self transitions, 1 global
  - Changes of state occur when clock says its ok
  - Based on value of inputs
- Inputs: reset, new, results of comparisons
- Output: open/closed



Datapath and Control Structure

- Datapath
  - Storage registers for combination values
  - Multiplexer
  - Comparator
- Control
  - Finite-state machine controller
  - Control for data-path (which value to compare)



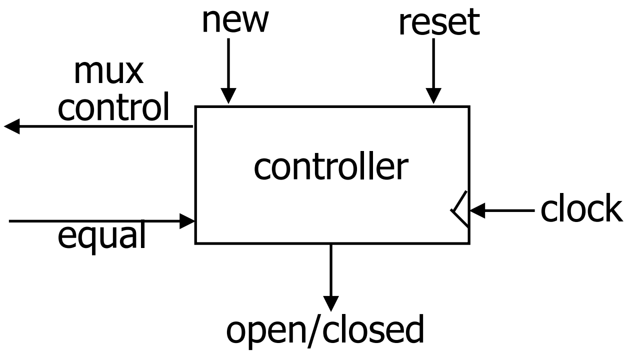
State Table for Combination Lock

- Finite-State Machine
  - Refine state diagram to take internal structure into account
  - State table ready for encoding

reset	new	equal	state	next state	mux	open/closed
1	–	–	–	S1	C1	closed
0	0	–	S1	S1	C1	closed
0	1	0	S1	ERR	–	closed
0	1	1	S1	S2	C2	closed
...						
0	1	1	S3	OPEN	–	open
...						

Encodings for Combination Lock

- Encode state table
  - State can be: S1, S2, S3, OPEN, or ERR
    - Needs at least 3 bits to encode: 000, 001, 010, 011, 100
    - And as many as 5: 00001, 00010, 00100, 01000, 10000
    - Choose 4 bits: 0001, 0010, 0100, 1000, 0000
  - Output mux can be: C1, C2, or C3
    - Needs 2 to 3 bits to encode
    - Choose 3 bits: 001, 010, 100
  - Output open/closed can be: open or closed
    - Needs 1 or 2 bits to encode
    - Choose 1 bit: 1, 0

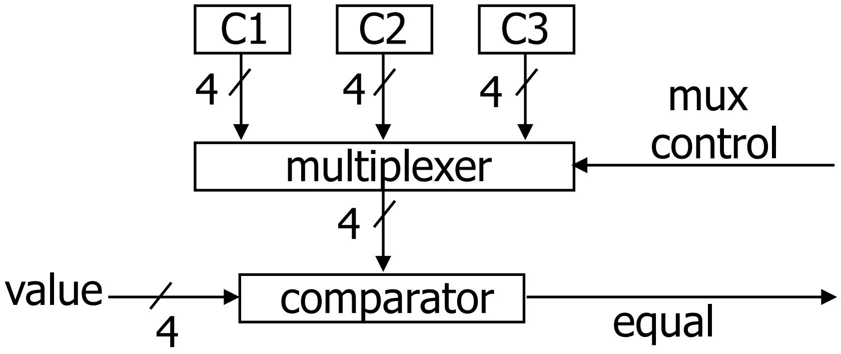


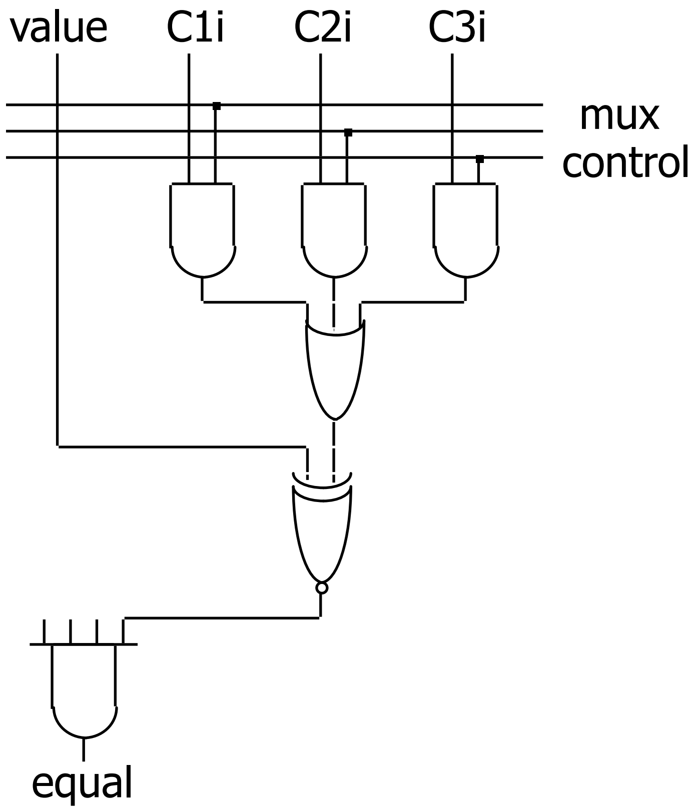
reset	new	equal	state	next state	mux	open/closed
1	–	–	–	0001	001	0
0	0	–	0001	0001	001	0
0	1	0	0001	0000	–	0
0	1	1	0001	0010	010	0
...						
0	1	1	0100	1000	–	1
...						

mux is identical to last 3 bits of state
open/closed is identical to first bit of state
therefore, we do not even need to implement
FFs to hold state, just use outputs

Datapath Implementation for Combination Lock

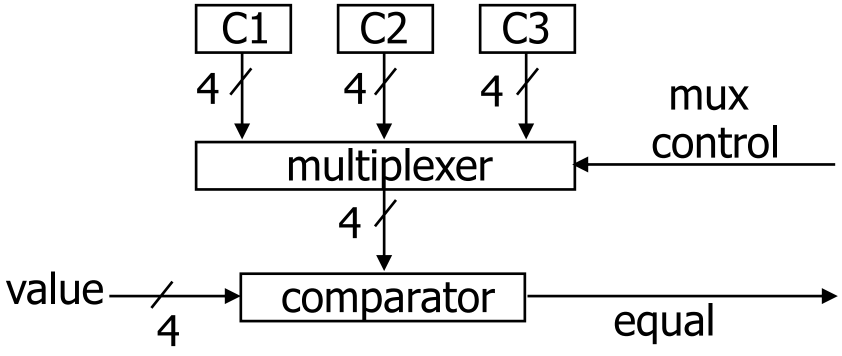
- Multiplexer
  - Easy to implement as combinational logic when few inputs
  - Logic can easily get too big for most PLDs

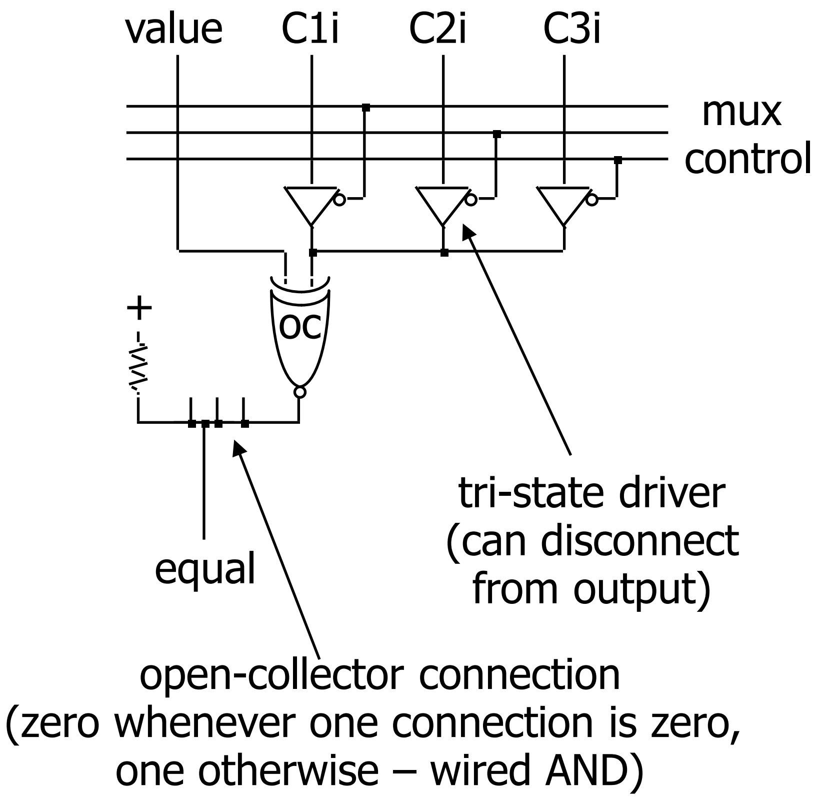




Datapath Implementation (cont'd)

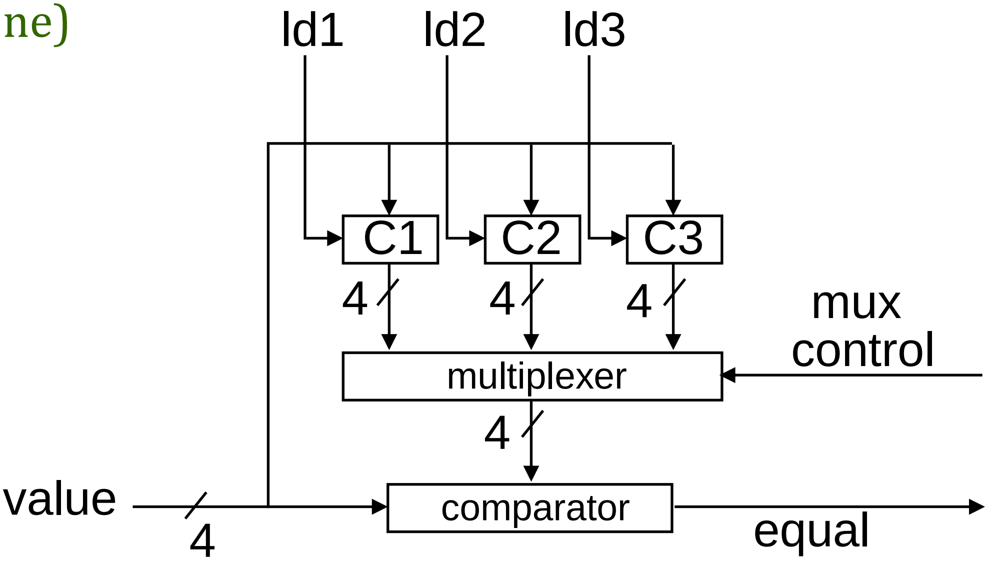
- Tri-State Logic
  - Utilize a third output state: "no connection" or "float"
  - Connect outputs together as long as only one is "enabled"
  - Open-collector gates can only output 0, not 1
    - Can be used to implement logical AND with only wires




value
C1i
C2i
C3i
mux control
OC
+
equal
tri-state driver (can disconnect from output)
open-collector connection (zero whenever one connection is zero, one otherwise – wired AND)

Digital Combination Lock (New Datapath)

- Decrease number of inputs
- Remove 3 code digits as inputs
  - Use code registers
  - Make them loadable from value
  - Need 3 load signal inputs (net gain in input (4*3)−3=9)
    - Could be done with 2 signals and decoder (ld1, ld2, ld3, load none)



Section Summary

- FSM Design
  - Understanding the problem
  - Generating state diagram
  - Implementation using synthesis tools
  - Iteration on design/specification to improve qualities of mapping
  - Communicating state machines
- Four case studies
  - Understand I/O behavior
  - Draw diagrams
  - Enumerate states for the "goal"
  - Expand with error conditions
  - Reuse states whenever possible

FSM AND HDL

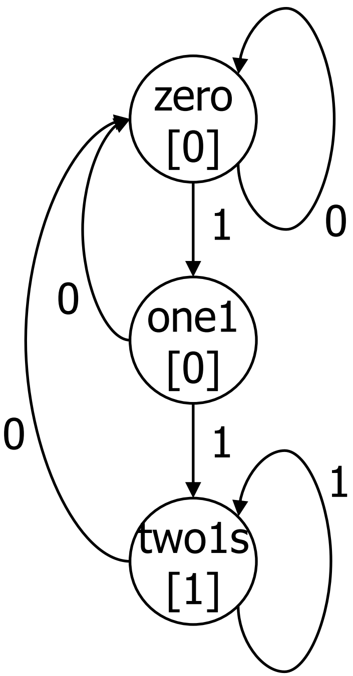
Hardware Description Languages and Sequential Logic

- 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

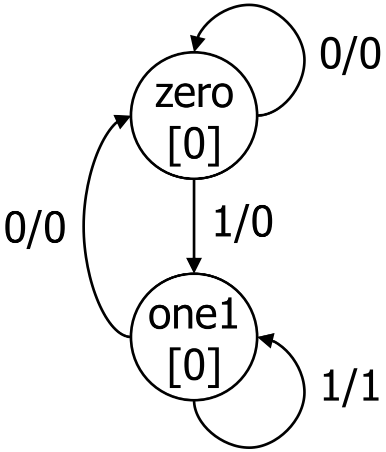
Example: reduce-1-string-by-1

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

Moore



Mealy

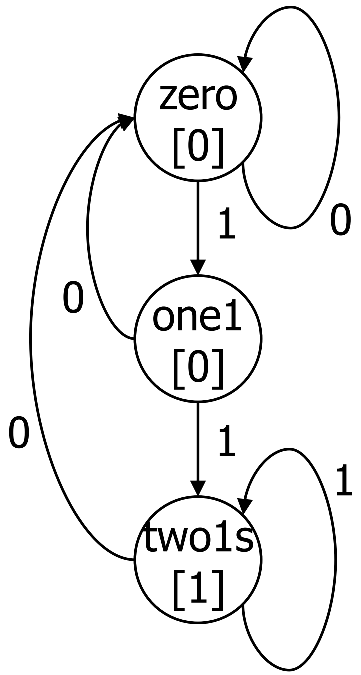


Verilog FSM - Reduce 1s example

- Moore machine

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

```
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;
```



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
```

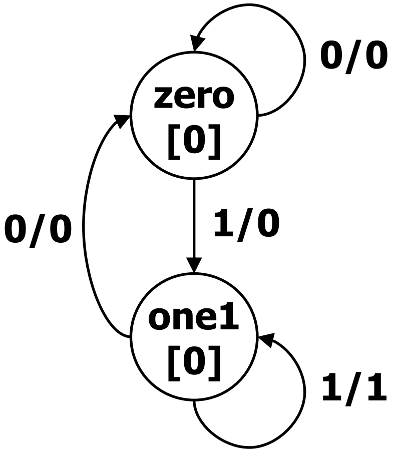
crucial to include
all signals that are
input to state determination

note that output
depends only on state

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

endmodule
```

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
```



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
```