inst.eecs.berkeley.edu/~eecs151

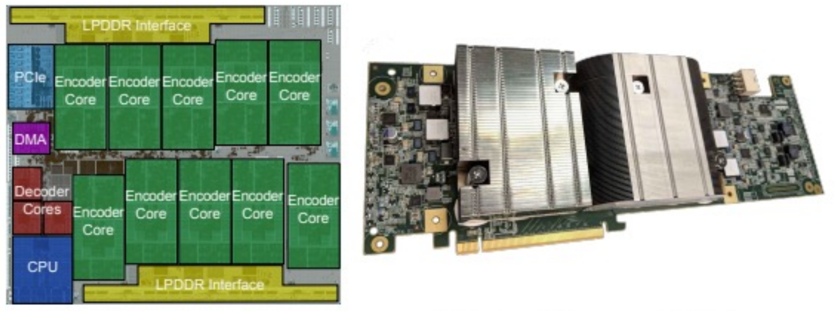
EECS151/251A Introduction to Digital Design and ICs

Lecture 6: CL and Finite State Machine

Sophia Shao

Google is now building its own video-transcoding chips

Google has decided that YouTube demands such a huge transcoding workload tha
it needs to build its own server chips. The company detailed its new "Argos" chips
a YouTube blog post, a CNET interview, and in a paper for ASPLOS,
the Architectural Support for Programming Languages and Operating Systems
Conference. Just as there are GPUs for graphics workloads and Google's TPU
(tensor processing unit) for AI workloads, the YouTube infrastructure team says it
has created the "VCU" or "Video (trans)Coding Unit," which helps YouTube
transcode a single video into over a dozen versions that it needs to provide a
smooth, bandwidth-efficient, profitable video site.



(a) Chip floorplan (b) Two chips on a PCBA

Figure 5: Pictures of the VCU

https://arstechnica.com/gadgets/2021/04/youtube-is-now-building-its-own-video-transcoding-chips/

- Combinational Logic
  - Introduction
  - Boolean Algebra
    - DeMorgan's Law
    - Sum of Products
    - Product of Sums
  - Logic Simplification
    - Boolean Simplification
    - Karnaugh Map

Why Logic simplification?

- Minimize number of gates in circuit
  - Gates take area
- Minimize amount of wiring in circuit
  - Wiring takes space and is difficult to route
  - Physical gates have limited number of inputs
- Minimize number of gate levels
  - Faster is better
- How to systematically simplify Boolean logics?
  - Use tools!

Practical methods for Boolean simplification

- Still based on Boolean algebra, but more systematic
- 2-level simplification -> multilevel
- Key tool: The Uniting Theorem

$$xy' + xy = x(y' + y) = x(1) = x$$

ab	f
00	0
01	0
10	1
11	1

$$f = ab' + ab = a(b'+b) = a$$

b values change within rows

a values don't change

b is eliminated, a remains

Example: Full Adder (FA) Carry out

Cout = a’bc + ab’c + abc’ + abc

ci	a	b	r	co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$c_{out} = a_i c_{in} + a_i b_i + b_i c_{in}$$
$$= c_{in}(a_i + b_i) + a_i b_i$$

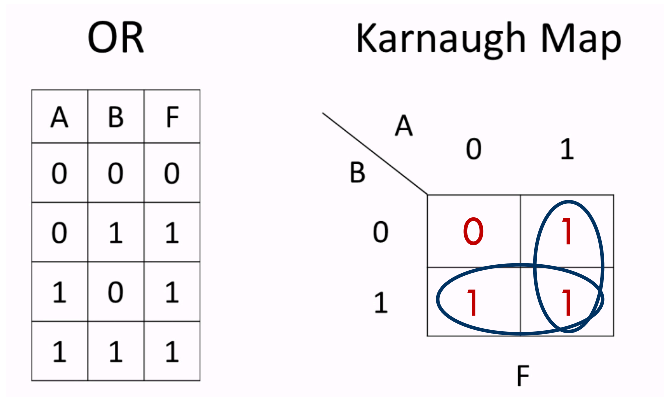
Example: Full Adder (FA) Carry out

$$
\begin{aligned}
\text{Cout} &= a'bc + ab'c + abc' + abc \\
&= a'bc + ab'c + abc' + abc + abc \\
&= a'bc + abc + ab'c + abc' + abc \\
&= (a' + a)bc + ab'c + abc' + abc \\
&= (1)bc + ab'c + abc' + abc \\
&= bc + ab'c + abc' + abc + abc \\
&= bc + ab'c + abc + abc' + abc \\
&= bc + a(b' + b)c + abc' + abc \\
&= bc + a(1)c + abc' + abc \\
&= bc + ac + ab(c' + c) \\
&= bc + ac + ab(1) \\
&= bc + ac + ab
\end{aligned}
$$

ci	a	b	r	co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$
\begin{aligned}
c_{out} &= a_i c_{in} + a_i b_i + b_i c_{in} \\
&= c_{in}(a_i + b_i) + a_i b_i
\end{aligned}
$$

Karnaugh Map Method



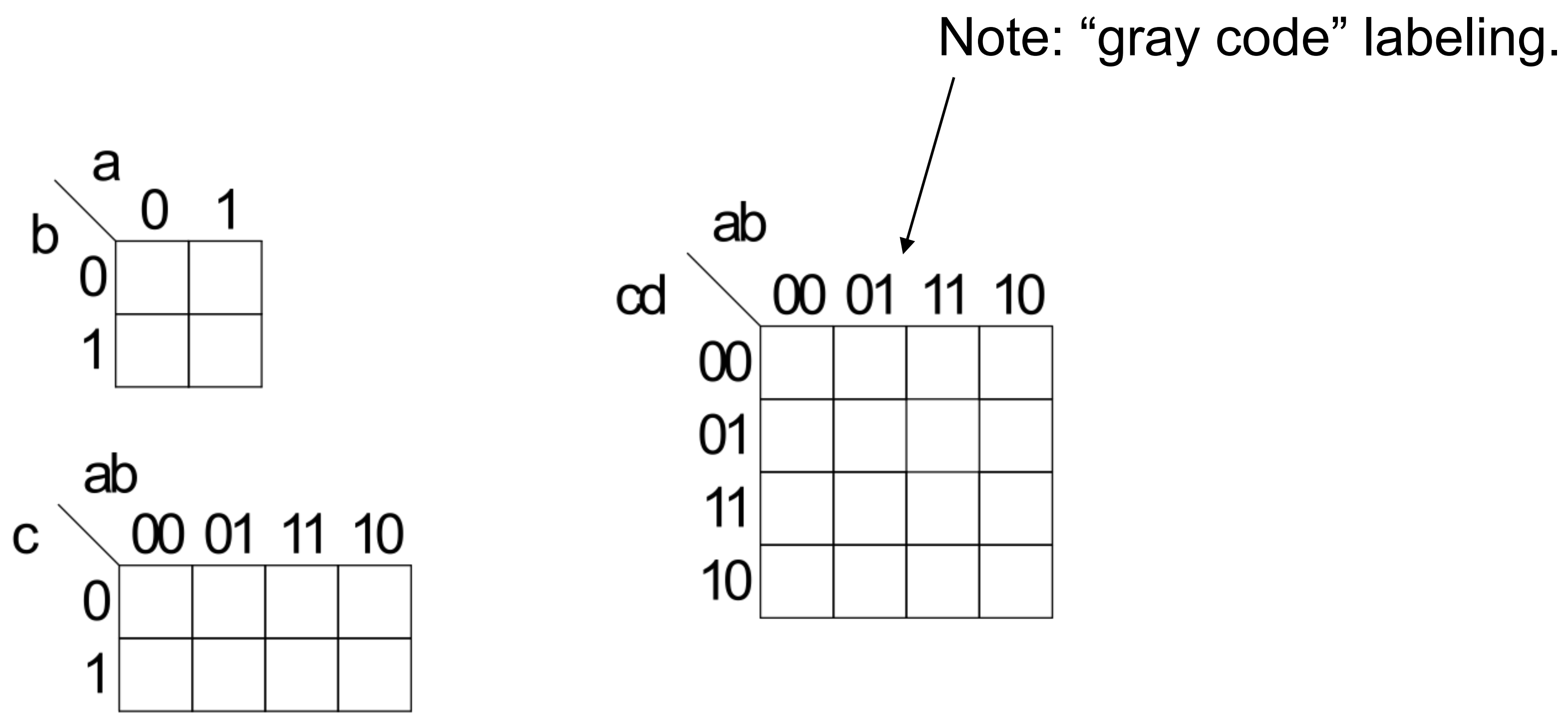
A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

F = A + B

1. Draw K-map of the appropriate number of variables.
2. Fill in map with function values from truth table.
3. Form groups of 1's.
   - ✓ Dimensions of groups must be even powers of two (1x1, 1x2, 1x4, …, 2x2, 2x4, …)
   - ✓ <u>Form as large as possible groups and as few groups as possible.</u>
   - ✓ Groups can overlap (this helps make larger groups)
   - ✓ Remember K-map is periodical in all dimensions (groups can cross over edges of map and continue on other side)
4. For each group write a product term.
   - the term includes the "constant" variables (use the uncomplemented variable for a constant 1 and complemented variable for constant 0)
5. Form Boolean expression as sum-of-products.

Karnaugh Map Method

- K-map is an alternative method of representing **the truth table** and to help visual the **adjacencies**.



Karnaugh Map Method

- Adjacent groups of 1's represent product terms

b\a	0	1
0	**0**	**1**
1	**0**	**1**

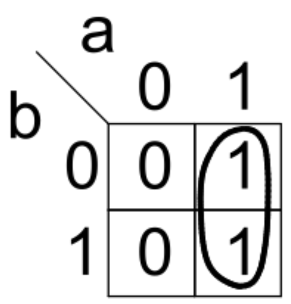
b\a	0	1
0	**1**	**1**
1	**0**	**0**

c\ab	00	01	11	10
0	**0**	**0**	**1**	**0**
1	**0**	**1**	**1**	**1**

c\ab	00	01	11	10
0	**0**	**0**	**1**	**1**
1	**0**	**0**	**1**	**1**

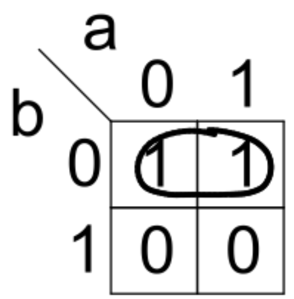
Karnaugh Map Method

- Adjacent groups of 1's represent product terms



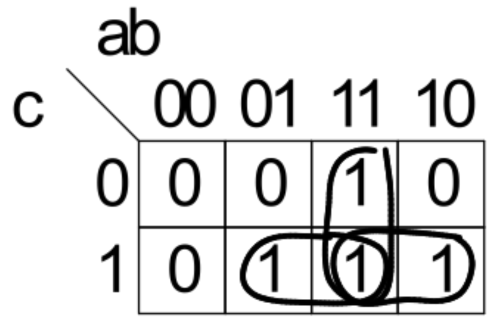
b \ a	0	1
0	0	1
1	0	1

$f = a$



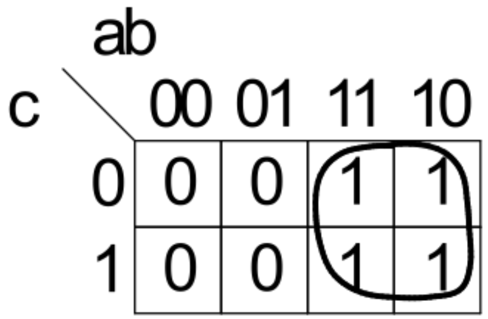
b \ a	0	1
0	1	1
1	0	0

$g = b'$



c \ ab	00	01	11	10
0	0	0	1	0
1	0	1	1	1

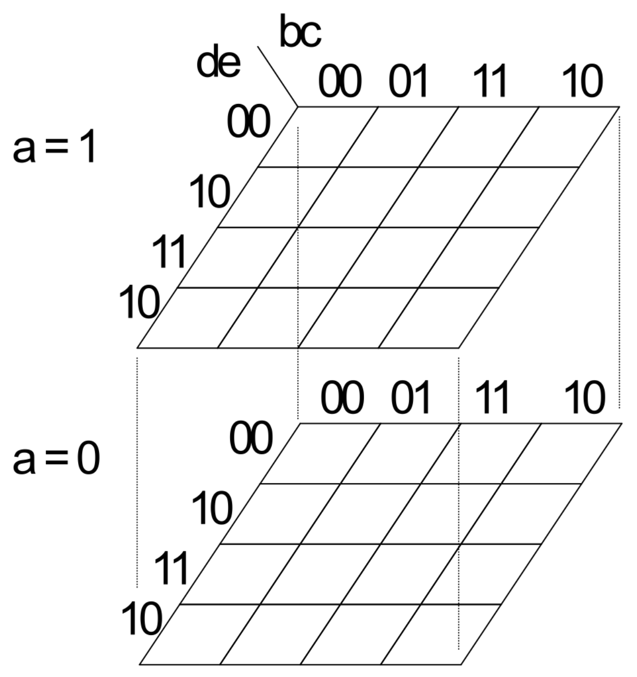
$cout = ab + bc + ac$

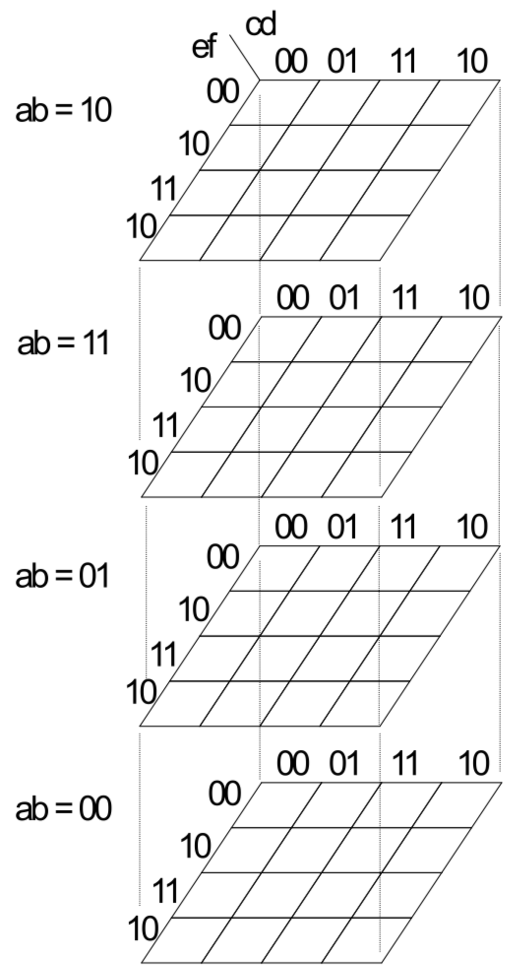


c \ ab	00	01	11	10
0	0	0	1	1
1	0	0	1	1

$f = a$

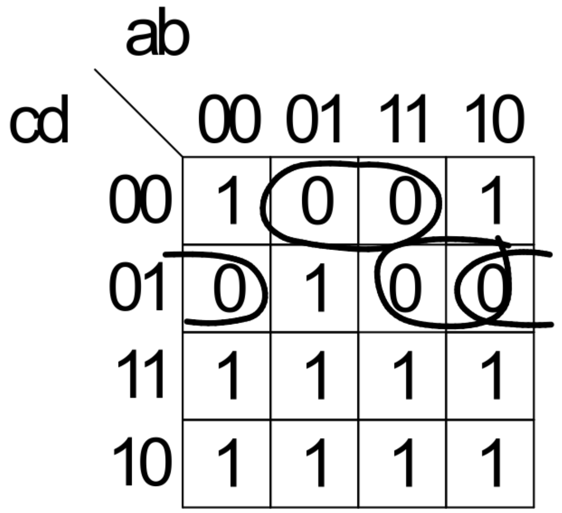
Higher Dimensional K-maps





Product-of-Sums Version

1. Form groups of 0’s instead of 1’s.
2. For each group write a sum term.
   - the term includes the “constant” variables (use the uncomplemented variable for a constant 0 and complemented variable for constant 1)
3. Form Boolean expression as product-of-sums.



cd \ ab	00	01	11	10
00	1	0	0	1
01	0	1	0	0
11	1	1	1	1
10	1	1	1	1

$$f = (b' + c + d)(a' + c + d')(b + c + d')$$

Summary

- Combinational circuits:
  - The outputs only depend on the current values of the inputs (memoryless).
  - The functional specification of a combinational circuit can be expressed as:
    - A truth table
    - A Boolean equation
- Boolean algebra
  - Deal with variables that are either True or False.
  - Map naturally to hardware logic gates.
  - Use theorems of Boolean algebra and Karnaugh maps to simplify equations.
- Common job interview questions ☺

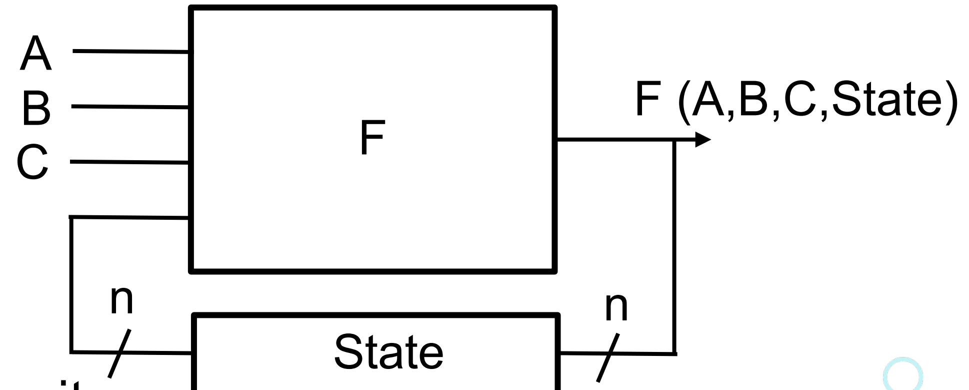
Administrivia

- Hope you enjoyed Lab 2!
  - Wrap up Lab 2 if you haven't.
  - Don't fall behind.
- Lab 3 starts this week.
- HW 2 due this week.
  - HW3 will be released this week.

- **Finite State Machine**
  - **Introduction**
  - **Moore vs Mealy FSM**

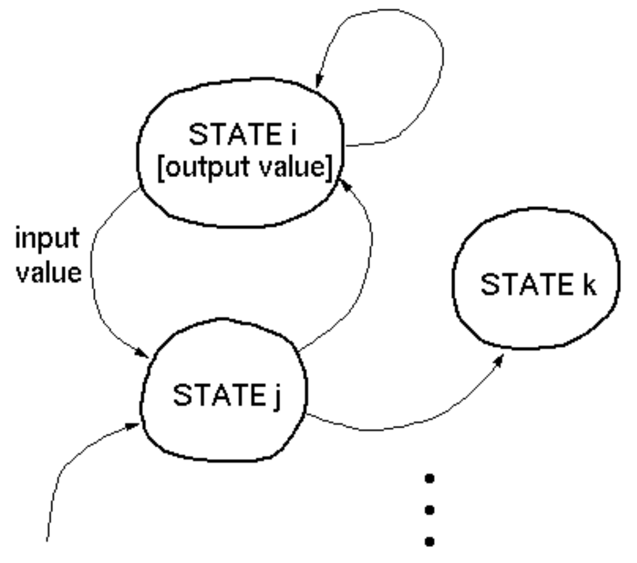
Sequential logic

- Combinational logic:
  - Memoryless: the outputs only dependent on the current inputs.
- Sequential logic:
  - Memory: the outputs depend on both current and previous values of the inputs.
    - Distill the prior inputs into a smaller amount of information, i.e., states.
  - State: the information about a circuit
    - Influences the circuit’s future behavior
    - Stored in Flip-flops and Latches
  - Finite State Machines:
    - Useful representation for designing sequential circuits
    - As with all sequential circuits: output depends on present and past inputs
    - We will first learn how to design by hand then how to implement in Verilog.



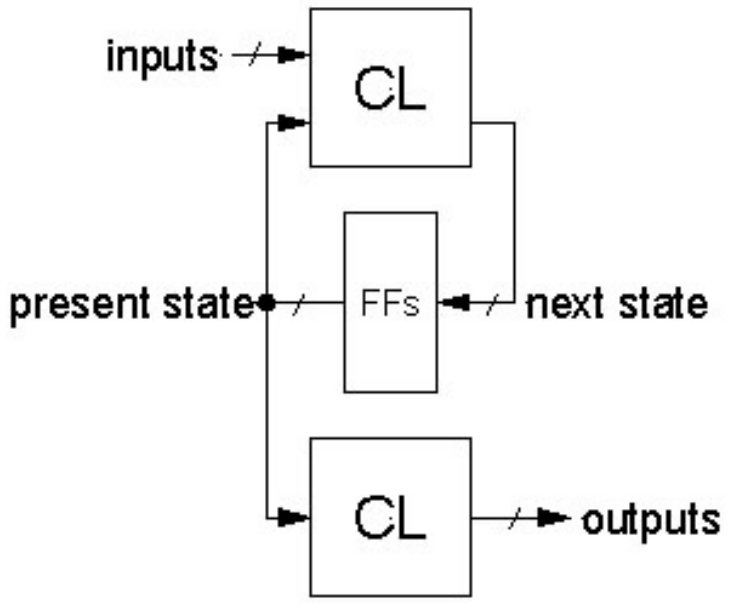
Finite State Machines

- A sequential circuit which has
  - External inputs
  - Externally visible outputs
  - Internal states



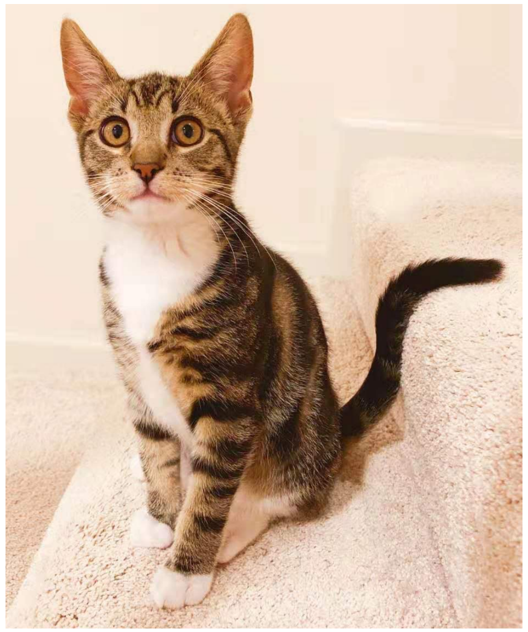
State Transition Diagram

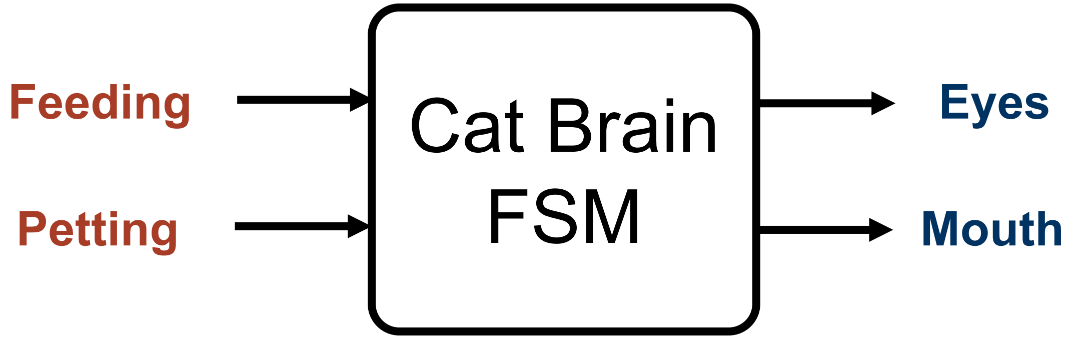
- Consists of:
  - State register
    - Stores current state
    - Loads previously calculated next state
    - # of states <= 2^(# of FFs)
  - Combinational logic
    - Computes the next state
    - Computes the outputs



FSM Example

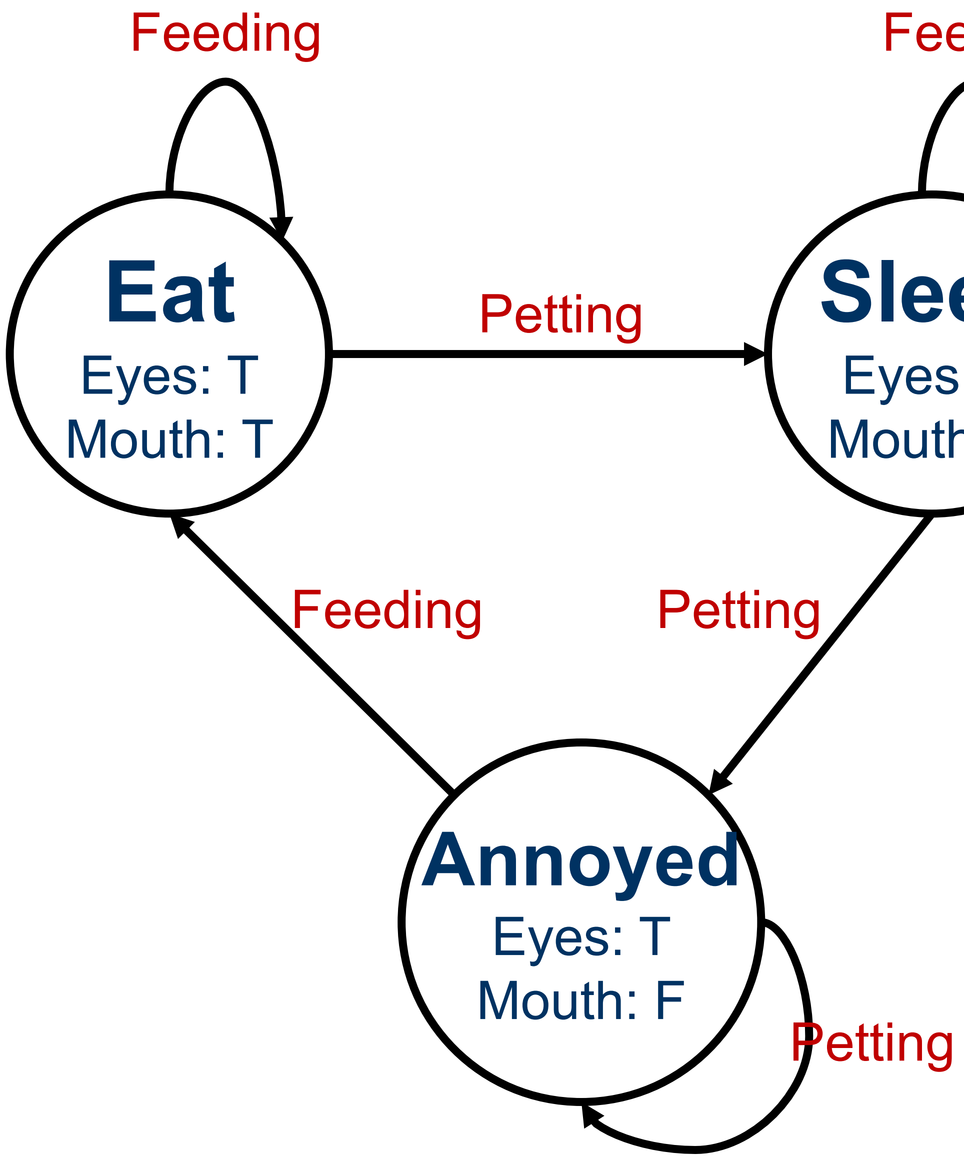
- Cat Brain (Simplified…)
  - Inputs:
    - Feeding
    - Petting
  - Outputs:
    - Eyes: open or close
    - Mouth: open or close
  - States:
    - Eating
    - Sleeping
    - Annoyed…





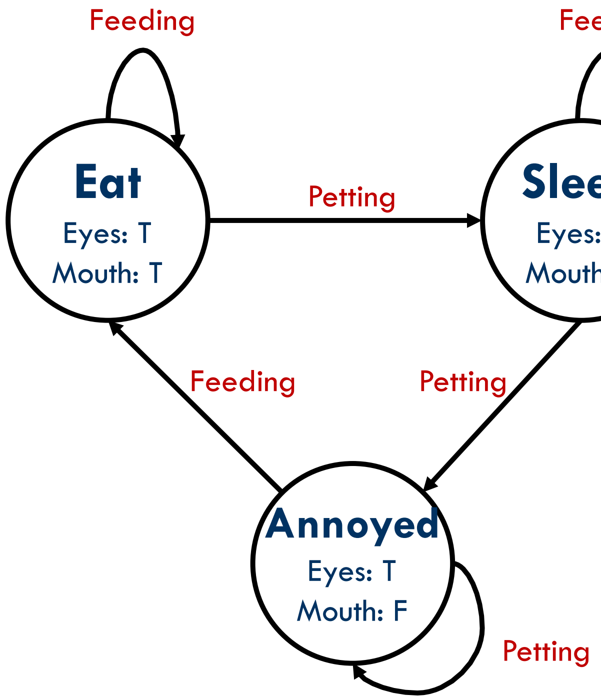
FSM State Transition Diagram

- States:
  - Circles
- Outputs:
  - Labeled in each state
  - Arcs
- Inputs:
  - Arcs



FSM Symbolic State Transition Table

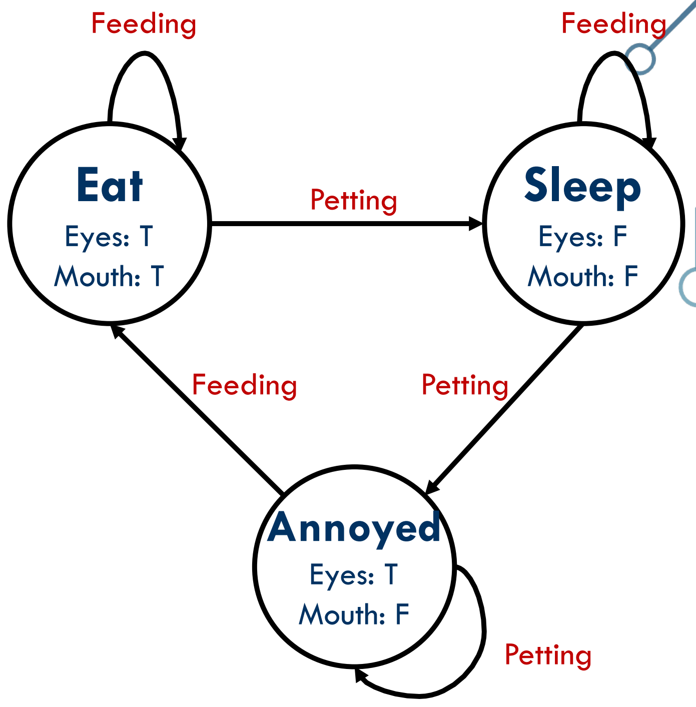
Current State	Inputs	Next State
Eat	Feeding	Eat
Eat	Petting	Sleep
Sleep	Feeding	Sleep
Sleep	Petting	Annoyed
Annoyed	Feeding	Eat
Annoyed	Petting	Annoyed



FSM Encoded State Transition Table

State	Encoding
Eat	00
Sleep	01
Annoyed	10

Current State		Input	Next State	
S1	S0	X	S1’	S0’
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	0
1	0	1	1	0



Current State	Inputs	Next State
Eat	Feeding	Eat
Eat	Petting	Sleep
Sleep	Feeding	Sleep
Sleep	Petting	Annoyed
Annoyed	Feeding	Eat
Annoyed	Petting	Annoyed

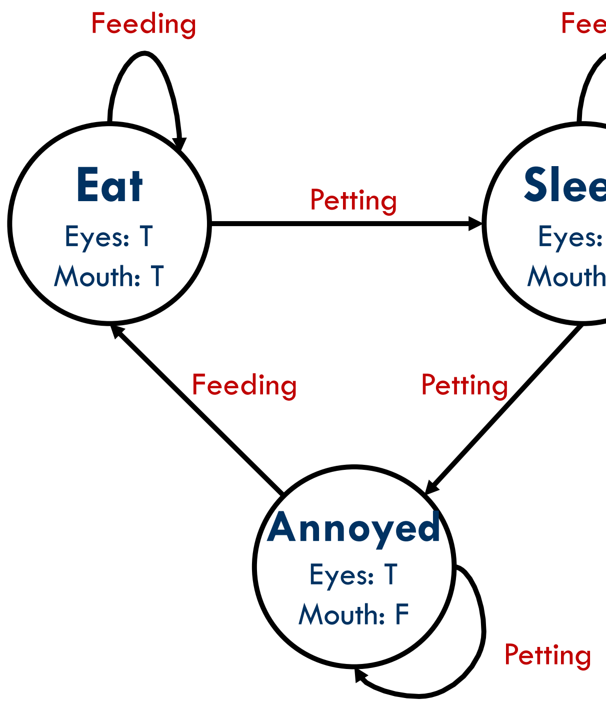
$$S0' = \overline{S1}\,\overline{S0}X + \overline{S1}S0\overline{X} = \overline{S1}(\overline{S0}X + S0\overline{X}) = \overline{S1}(S0 \oplus X)$$
$$S1' = \overline{S1}S0X + S1\overline{S0}X = (S1 \oplus S0)X$$

FSM Output Table

State	Encoding
Eat	00
Sleep	01
Annoyed	10

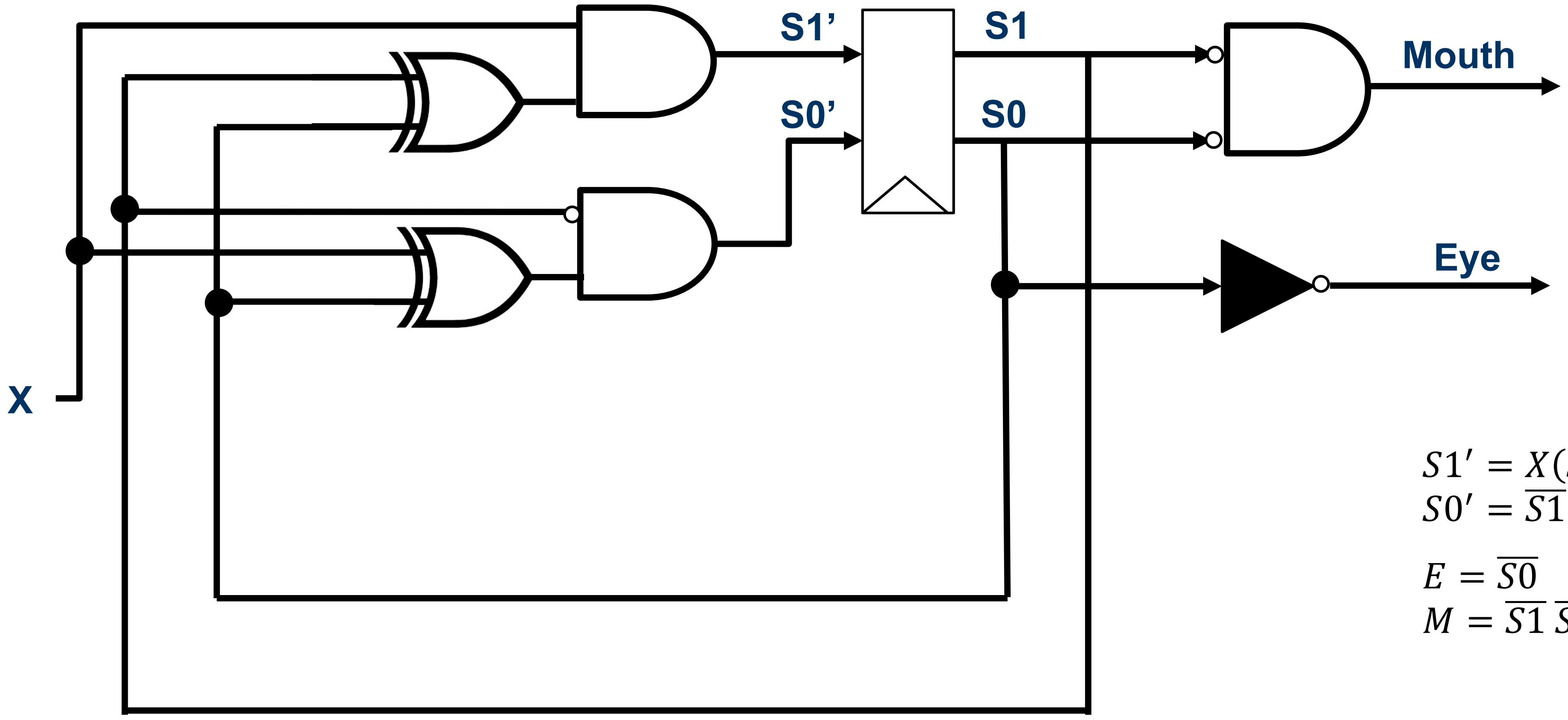
Current State		Outputs	
S1	S0	E	M
0	0	1	1
0	1	0	0
1	0	1	0

Outputs		Encoding
Eyes	Mouth	
Open	Open	11
Close	Close	00
Open	Close	10



$$E = \overline{S1}\,\overline{S0} + S1\overline{S0} = \overline{S0}$$
$$M = \overline{S1}\,\overline{S0}$$

FSM Gate Representation



$$S1' = X(S0 \oplus S1)$$
$$S0' = \overline{S1}(S0 \oplus X)$$

$$E = \overline{S0}$$
$$M = \overline{S1}\,\overline{S0}$$

FSM Design Process

- Specify circuit function
- Draw state transition diagram
- Write down symbolic state transition table
- Write down encoded state transition table →
- Derive logic equations
- Derive circuit diagram
  - Register to hold state
  - Combinational logic for next state and outputs

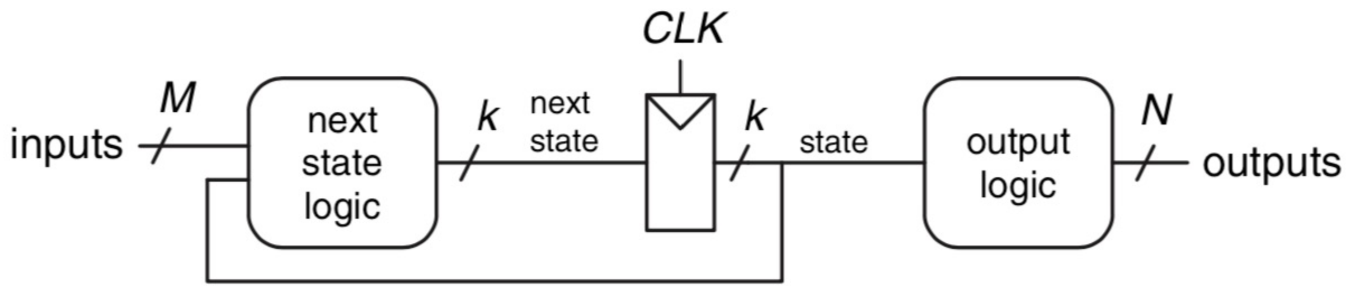
- Binary encoding:
  - i.e., for four states, 00, 01, 10, 11
- One-hot encoding
  - One state bit per state
  - Only one state bit TRUE at once
  - i.e., for four states, 0001, 0010, 0100, 1000
  - Requires more flip-flops
  - Often next state and output logic can be simpler

- Finite State Machine
  - Introduction
  - Moore vs Mealy FSM

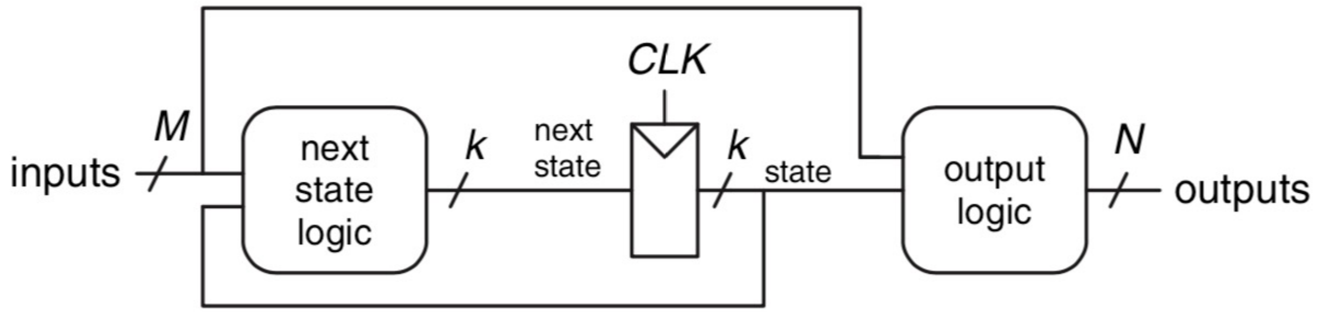
Moore vs Mealy FSMs

- Next state is always determined by current state and inputs
- Differ in output logic:
    - Moore FSM: outputs depend only on current state
    - Mealy FSM: outputs depend on current state and inputs

Moore FSM

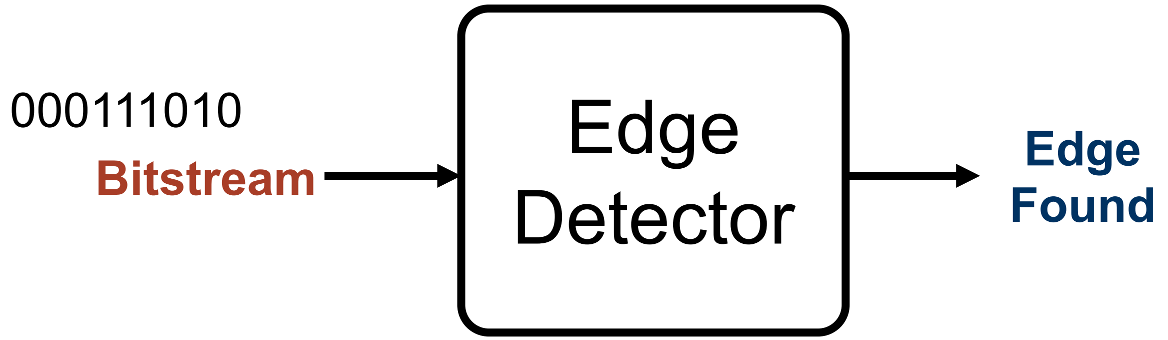


Mealy FSM

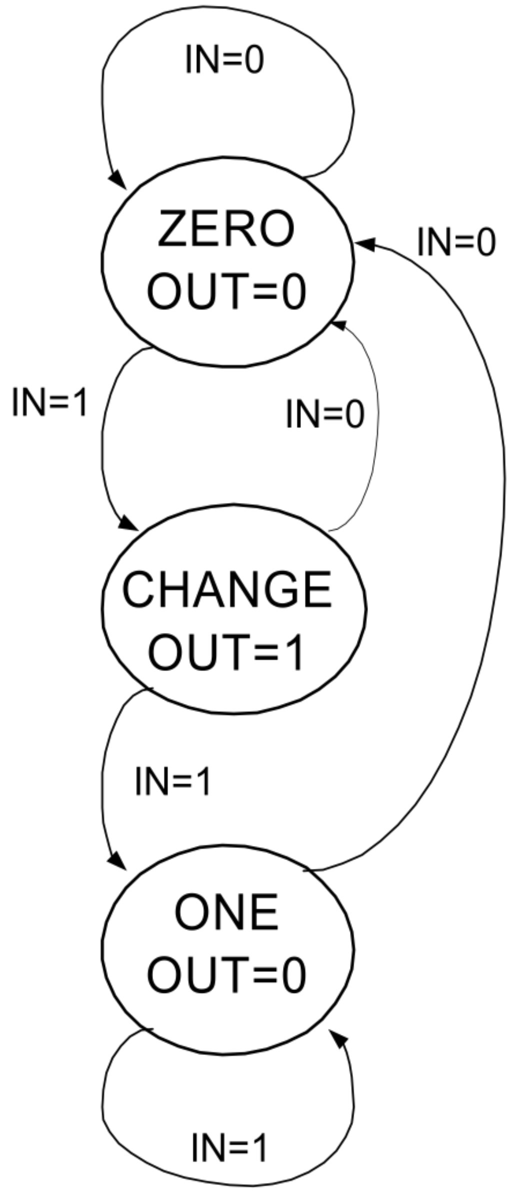


Example: Edge Detector

- Input:
  - A bit stream that is received one bit at a time.
- Output:
  - 0/1
- Circuit:
  - Asserts its output to be true when the input bit stream changes from 0 to 1.



State Transition Diagram Solution A (Moore)



Input	Current State	Next State	Output
0	Zero (00)	Zero	0
1	Zero (00)	Change	0
0	Change (01)	Zero	1
1	Change (01)	One	1
0	One (11)	Zero	0
1	One (11)	One	0

State Transition Diagram Solution A (Moore)

CS

IN	00	01	11	10
0	0	0	0	-
1	0	1	1	-

$NS_1 =$ IN *AND* CS0

CS

IN	00	01	11	10
0	0	0	0	-
1	1	1	1	-

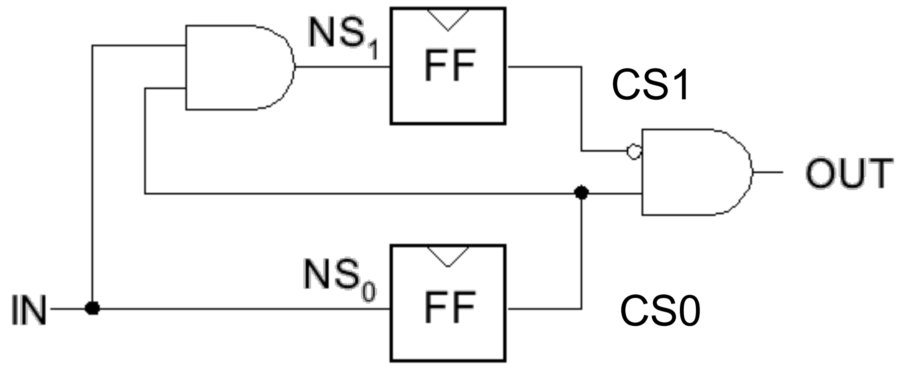
$NS_0 =$ IN

CS

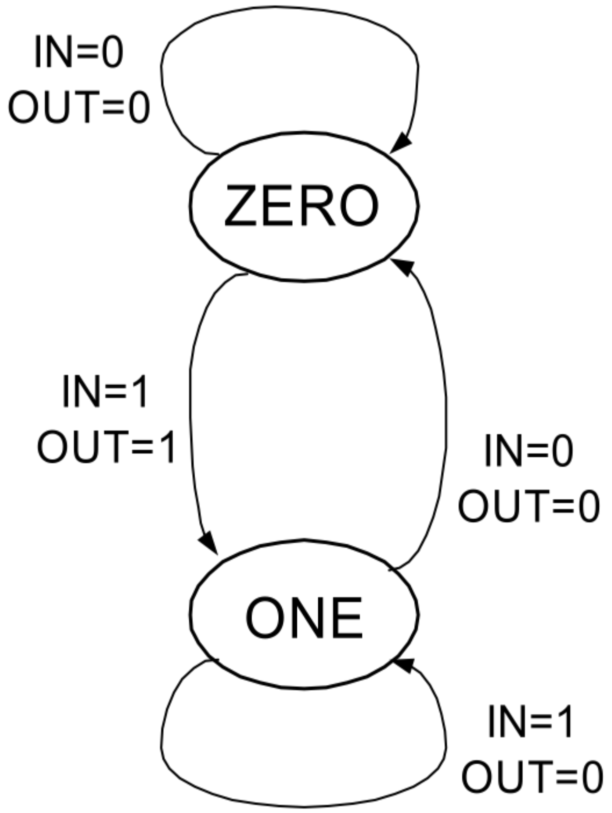
IN	00	01	11	10
0	0	1	0	-
1	0	1	0	-

OUT= *NOT* (CS1) *AND* CS0

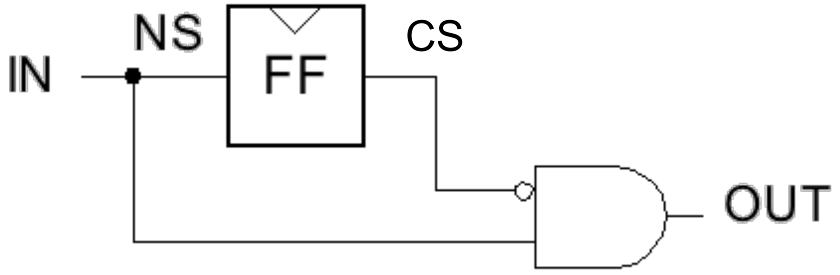
Input	Current State	Next State	Output
0	Zero (00)	Zero	0
1	Zero (00)	Change	0
0	Change (01)	Zero	1
1	Change (01)	One	1
0	One (11)	Zero	0
1	One (11)	One	0



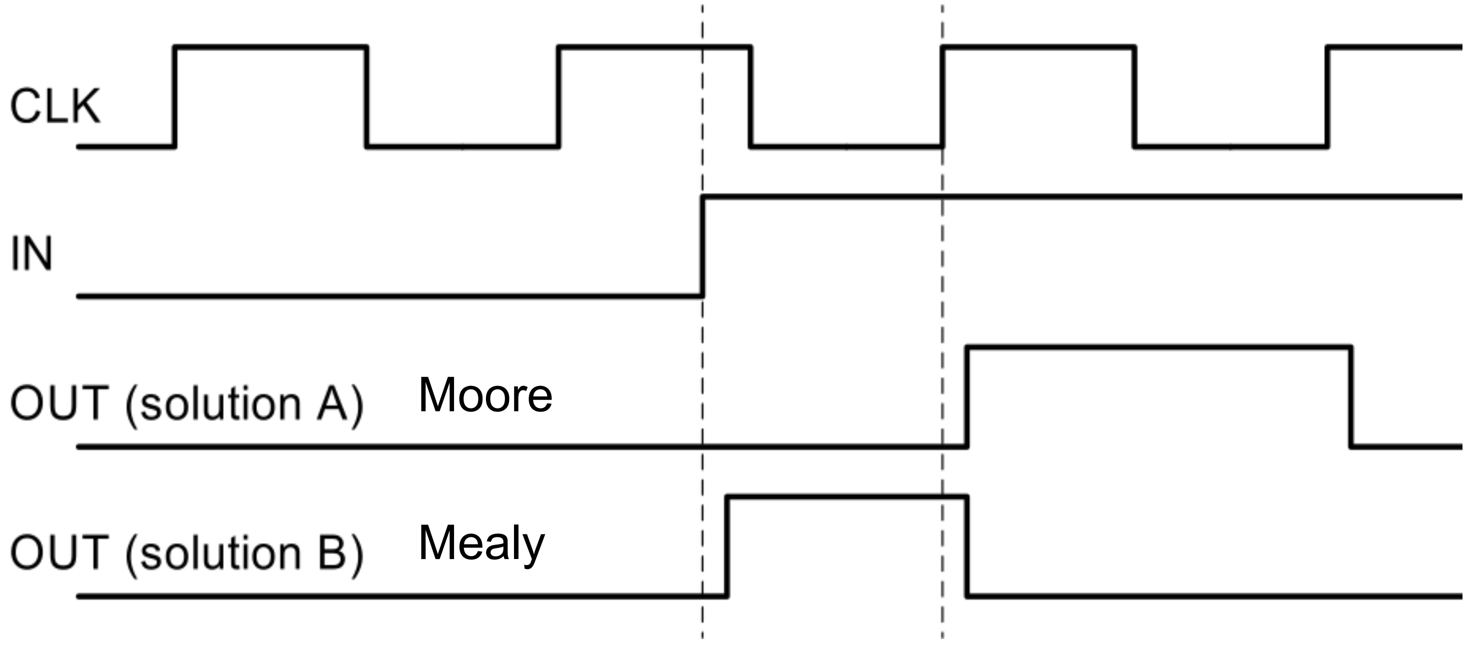
State Transition Diagram Solution B (Mealy)



Input	Current State	Next State	Output
0	Zero (0)	Zero	0
1	Zero (0)	One	1
0	One (1)	Zero	0
1	One (1)	One	0



Edge Detection Timing Diagrams



- Solution A (Moore) : both edges of output follow the clock
- Solution B (Mealy) : output rises with input rising edge and is asynchronous wrt the clock, output falls synchronous with next clock edge

FSM Comparison

Solution A

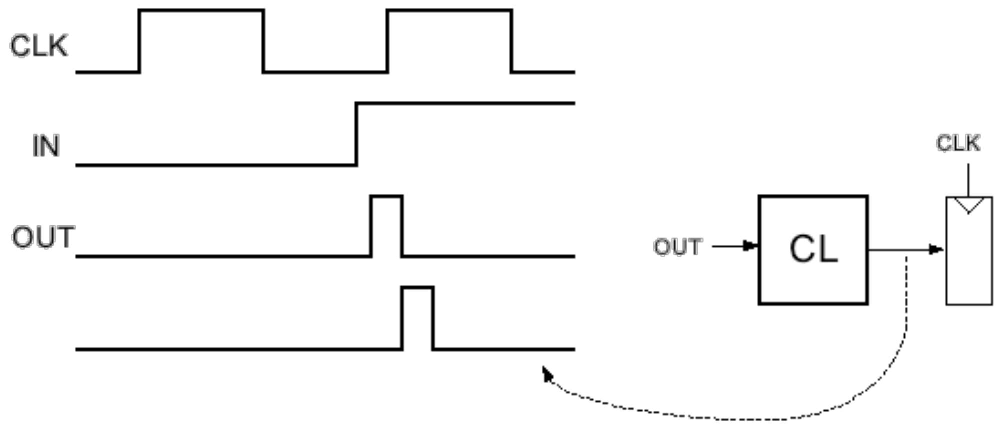
Moore Machine

- output function only of current state
- maybe more states (why?)
- synchronous outputs
  - Input glitches not send at output
  - one cycle "delay"
  - full cycle of stable output

Solution B

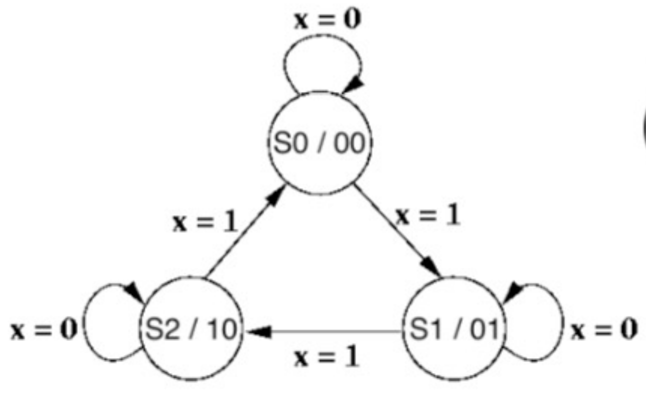
Mealy Machine

- output function of both current = & input
- maybe fewer states
- asynchronous outputs
  - if input glitches, so does output
  - output immediately available
  - output may not be stable long enough to be useful (below):

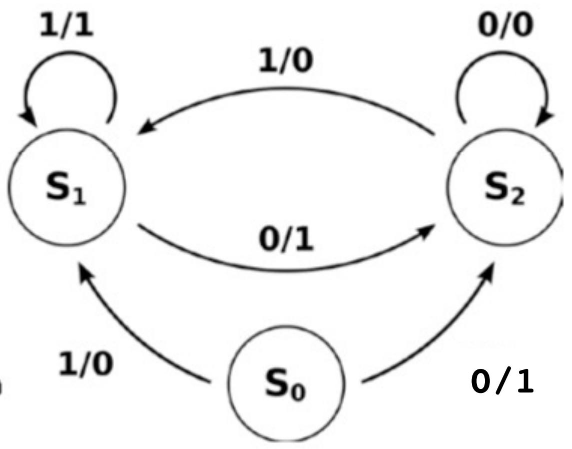


If output of Mealy FSM goes through combinational logic before being registered, the CL might delay the signal and it could be missed by the clock edge (or violate set-up time requirement)

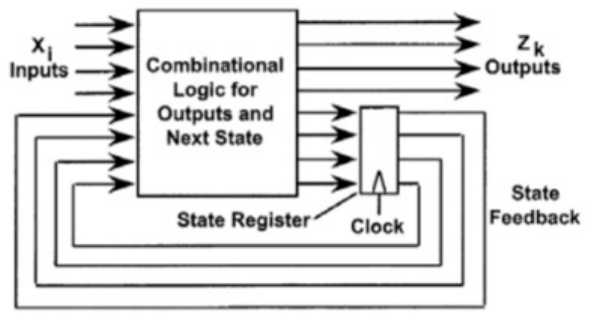
Quiz: Which of the diagrams are **Moore** machines?



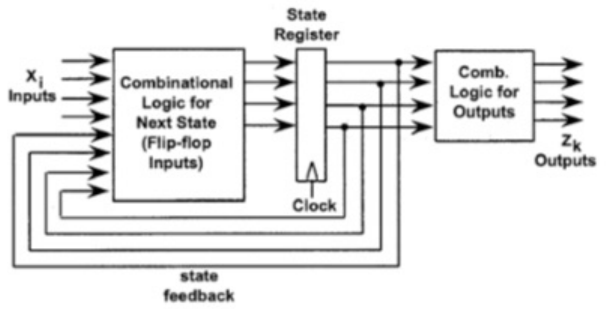
A.



B.



C.



D.

A. AC
B. BD
C. AD
D. BC

Summary

- Sequential logic:
  - Memory: the outputs depend on both current and previous values of the inputs.
- Finite State Machine:
  - Registers to store current states
  - Combinational logic:
    - Compute the next state
    - Compute the outputs
- Moore vs Mealy FSM:
  - Moore: Outputs depend only on current state
  - Mealy: Outputs depend on current state and inputs