inst.eecs.berkeley.edu/~eecs151

EECS151 : Introduction to Digital Design and ICs

Lecture 6 – Finite State Machines

Bora Nikolić and Sophia Shao

Tera Floating-point Operations/second (TFLOPS)

Review

- Combinational Logic
  - Memoryless: outputs are only dependent on the current inputs.
  - Three representations:
    - Truth table
    - Boolean expression
    - Gate representation
  - Boolean algebra
    - Commonly-used laws
    - Canonical forms:
      - Sum of Products (SOP)
      - Products of Sum (POS)

Canonical Forms

- Two types:
  - Sum of Products (SOP)
  - Product of Sums (POS)
- Sum of Products
  - a.k.a Disjunctive normal form, minterm expansion
  - Minterm: a product (AND) involving all inputs
  - SOP: Summing minterms for which the output is True

Minterms	a	b	c	f	f'
a'b'c'	0	0	0	0	1
a'b'c'	0	0	1	0	1
a'bc'	0	1	0	0	1
a'bc	0	1	1	1	0
ab'c'	1	0	0	1	0
ab'c	1	0	1	1	0
abc'	1	1	0	1	0
abc	1	1	1	1	0

One product (and) term for each 1 in f:

```
f = a'bc + ab'c' + ab'c + abc' + abc
f' = a'b'c' + a'b'c + a'bc'
```

Canonical Forms

- Two types:
  - Sum of Products (SOP)
  - Product of Sums (POS)
- Product of Sums:
  - a.k.a. conjunctive normal form, maxterm expansion
  - Maxterm: a sum (OR) involving all inputs
  - POS: Product (AND) maxterms for which the output is FALSE
  - Can obtain POSs from applying DeMorgan's law to the SOPs of F (and vice versa)

Maxterms	a	b	c	f	f'
a+b+c	0	0	0	0	1
a+b+c′	0	0	1	0	1
a+b′+c	0	1	0	0	1
a+b′+c′	0	1	1	1	0
a′+b+c	1	0	0	1	0
a′+b+c′	1	0	1	1	0
a′+b′+c	1	1	0	1	0
a′+b′+c′	1	1	1	1	0

One sum (**or**) term for each **0** in f:

```
f = (a+b+c)(a+b+c')(a+b'+c)
f' = (a+b'+c')(a'+b+c)(a'+b+c')
     (a'+b'+c)(a+b+c')
```

Quiz

- Derive the sum of products form of $\bar{Y}$ based on the truth table.

a) $\bar{Y} = (A + B)(A + \bar{B})$

b) $\bar{Y} = A\bar{B} + AB$

c) $\bar{Y} = \bar{A}\bar{B} + \bar{A}B$

A	B	Y	$\bar{Y}$
0	0	0	1
0	1	0	1
1	0	1	0
1	1	1	0

http://www.yellkey.com/method

Combinational Logic

Introduction

Boolean Algebra

Boolean Simplification

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

Example: Full Adder (FA) Carry out

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

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

Why do Boolean 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

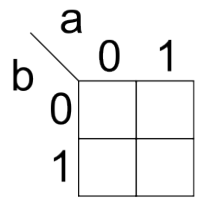
a values don't change

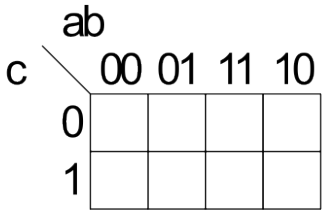
b is eliminated, a remains

Karnaugh Map Method

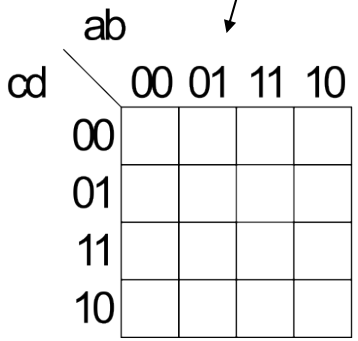
- K-map is an alternative method of representing the truth table and to help visual the adjacencies.

ab	f
00	0
01	0
10	1
11	1





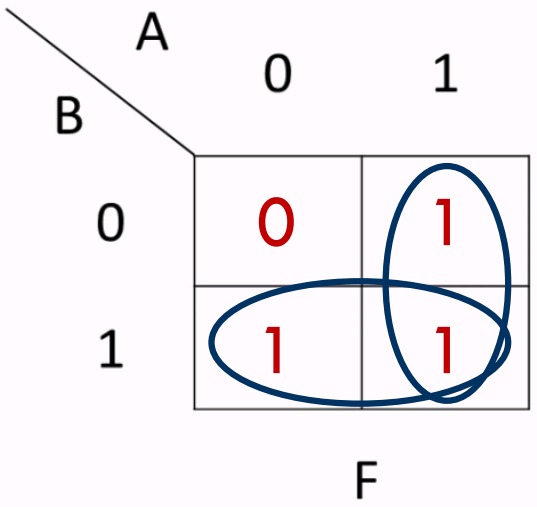
Note: "gray code" labeling.



Karnaugh Map Method

- Adjacent groups of 1's represent product terms

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



$F = A + B$

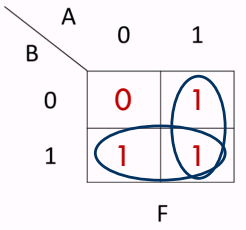
Karnaugh Map Method

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, …)
   - ✓ Form as large as possible groups and as few groups as possible.
   - ✓ 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.

OR

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

Karnaugh Map

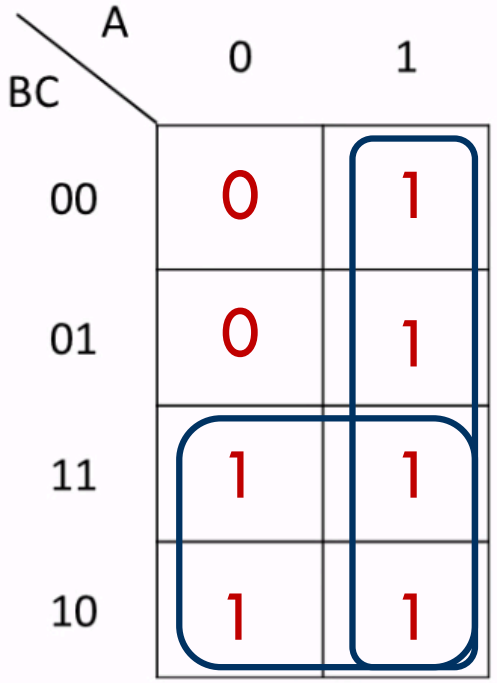


F = A + B

Karnaugh Map Method

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

$F = A + B$



Summary

- Combinational logic:
  - 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

- Midterm1
  - Wednesday, 10/2/2019
  - 2:30pm – 4pm
- Midterm2
  - Early November
- EECS Colloquium: Moore's Law is Not Dead
  - Jim Keller, Senior Vice President, Intel
  - Today, 4-5pm, Soda Hall, 306

Sequential Logic

Introduction

Finite State Machines
Moore vs Mealy FSM
FSM in Verilog

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
  - Common sequential circuits:
    - Finite State Machines (FSMs)

Sequential Logic

Introduction

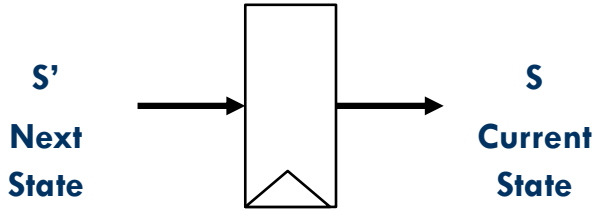
Finite State Machines

Moore vs Mealy FSM

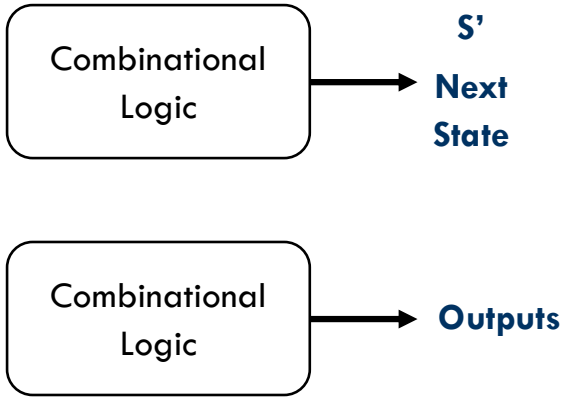
FSM in Verilog

Finite State Machines

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

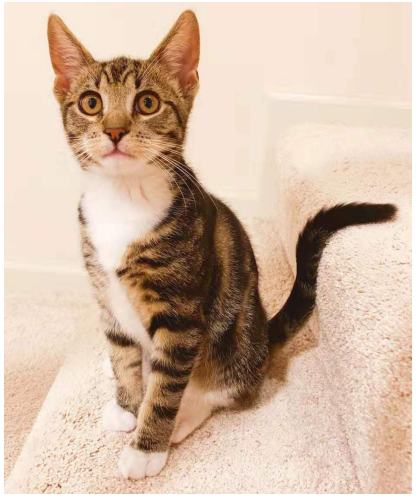


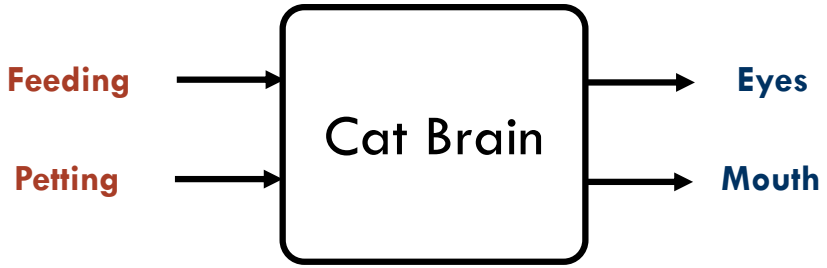
- Consists of:
  - State register
    - Stores current state
    - Loads previously calculated next state
  - Combinational logic
    - Computes the next state
    - Computes the outputs



FSM Example

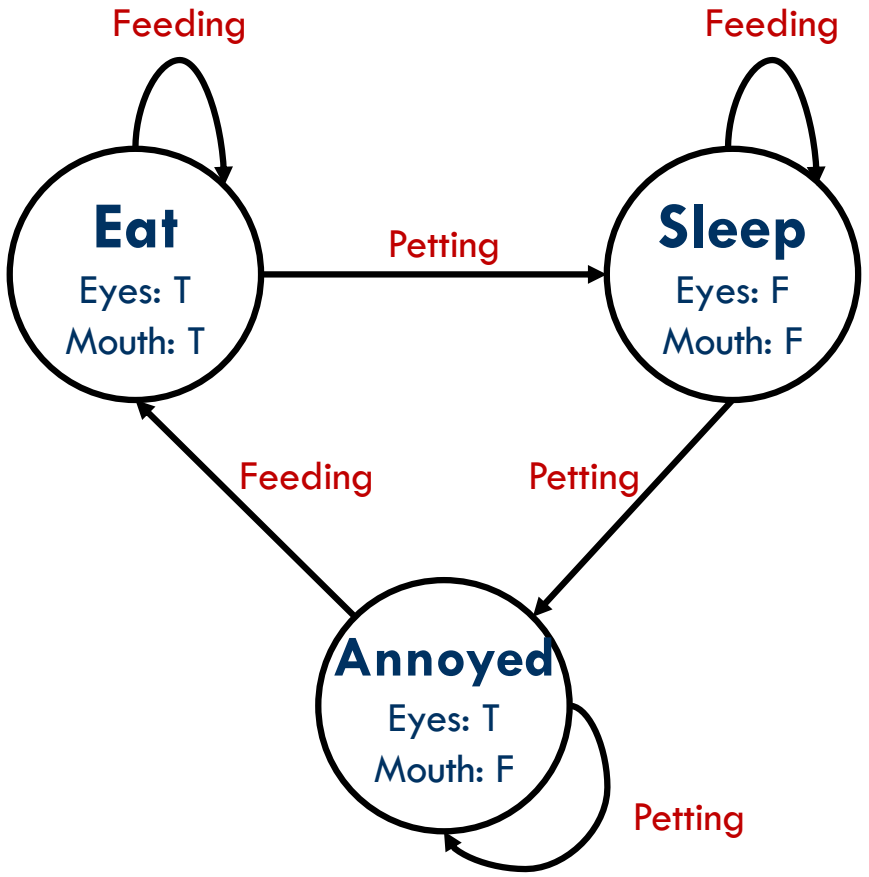
- Cat Brain (Simplified…)
  - Inputs:
    - Feeding
    - Petting
  - Outputs:
    - Eyes: open or close
    - Mouth: open or close
  - States:
    - Eat
    - Sleep
    - Annoyed…





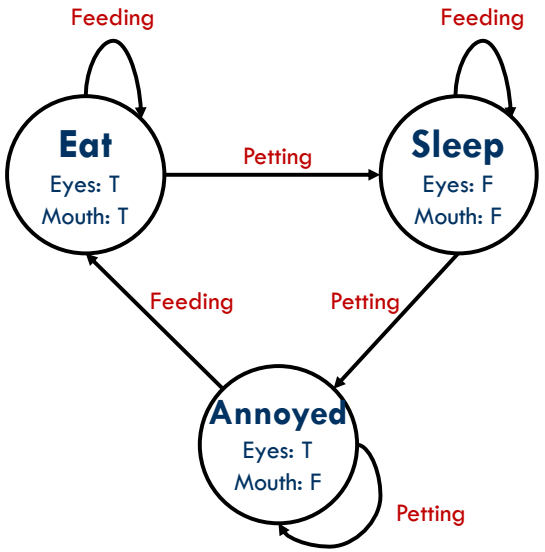
FSM State Transition Diagram

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



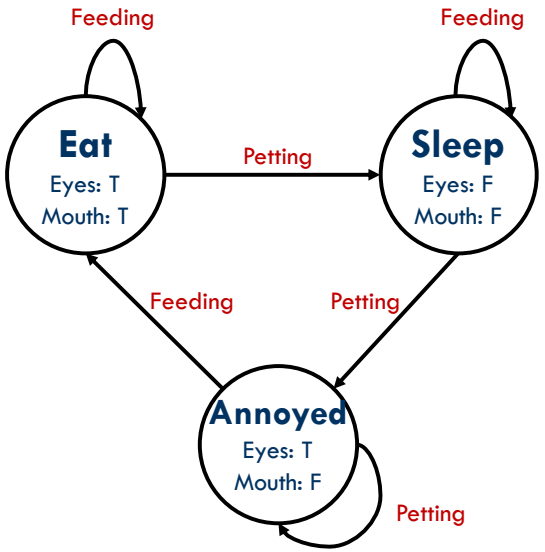
FSM Symbolic State Transition Table

Current State	Inputs	Next State
Eat	Feeding	Eat



FSM Symbolic State Transition Table

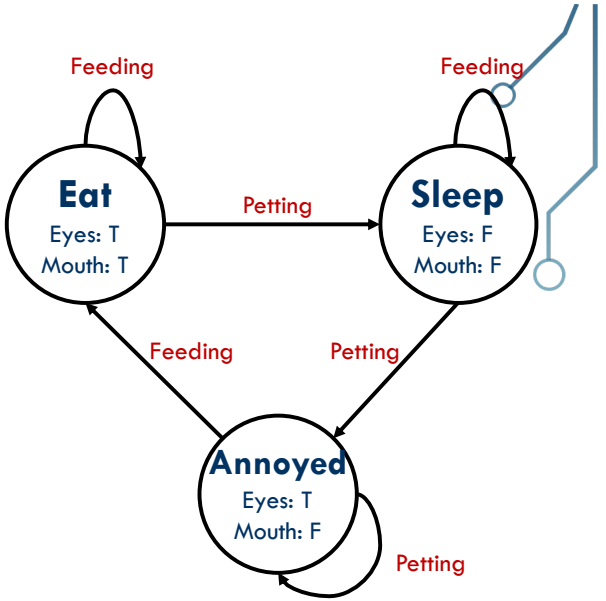
Current State	Inputs	Next State
Eat	Feeding	Eat
Eat	Petting	Sleep
Sleep	Feeding	Sleep
Sleep	Petting	Annoyed
Annoyed	Feeding	Eat
Annoyed	Petting	Annoyed



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

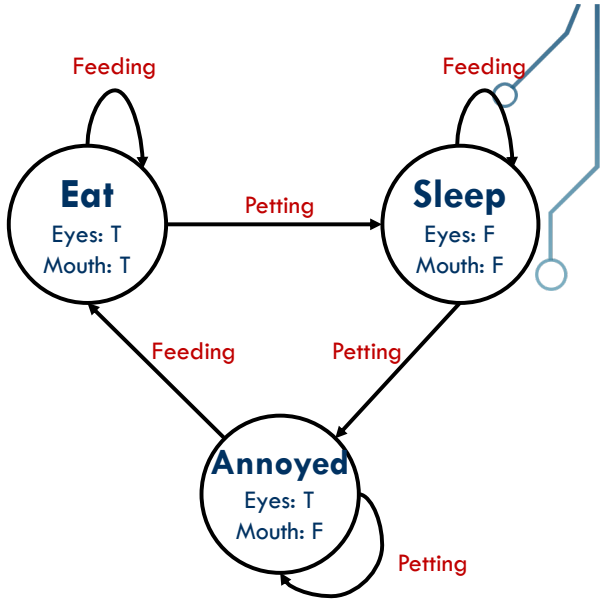


Current State	Inputs	Next State
Eat	Feeding	Eat
Eat	Petting	Sleep
Sleep	Feeding	Sleep
Sleep	Petting	Annoyed
Annoyed	Feeding	Eat
Annoyed	Petting	Annoyed

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

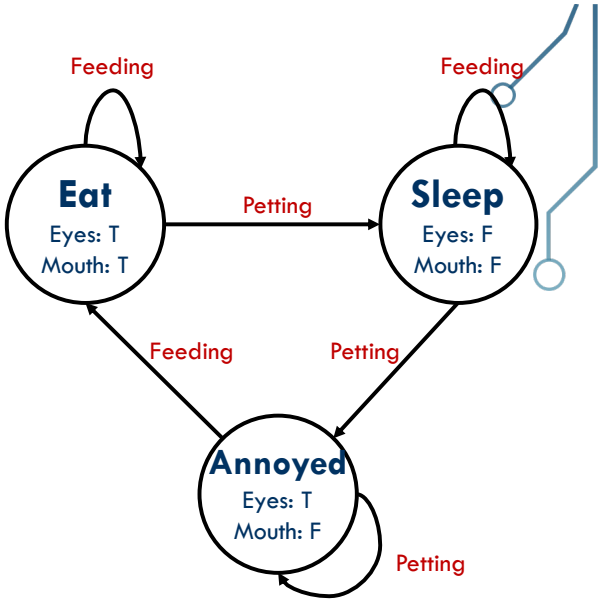
$S0' =$

$S1' =$

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

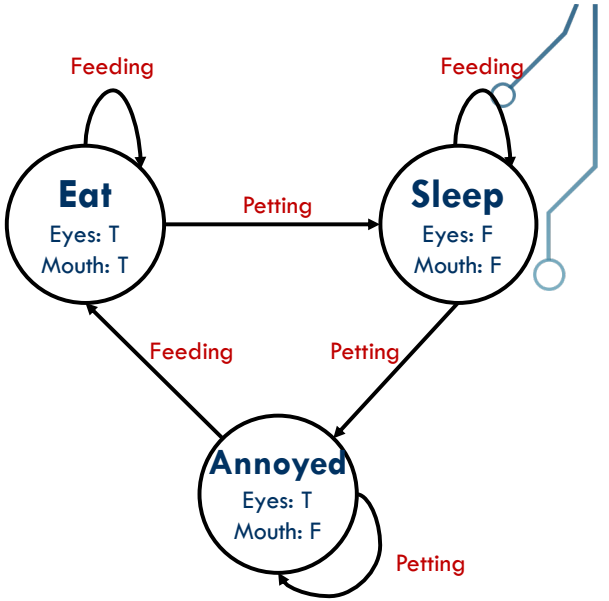
$$S0' = \overline{S1}\,\overline{S0}X + \overline{S1}S0\bar{X} = \overline{S1}(\overline{S0}X + S0\bar{X}) = \overline{S1}(S0 \oplus X)$$
$$S1' = \overline{S1}S0X + S1\overline{S0}X = (S1 \oplus S0)X$$

FSM Output Table

State	Encoding
Eat	00
Sleep	01
Annoyed	10

Current State		Outputs	
S1	S0	E	M
0	0	1	1

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

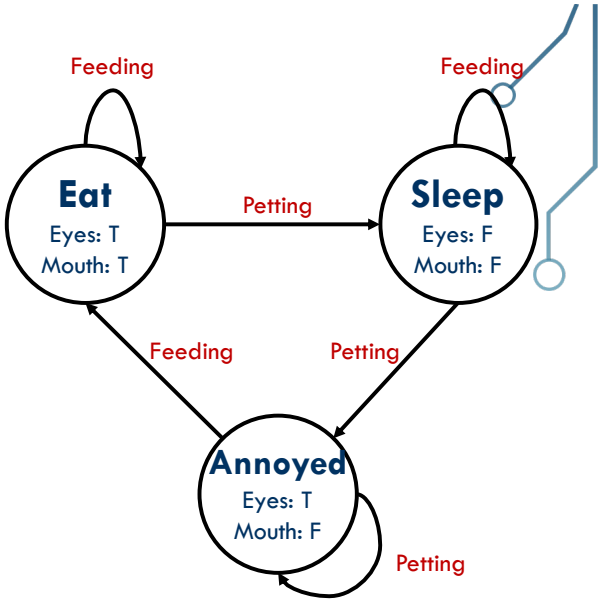


FSM Output Table

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



$E =$

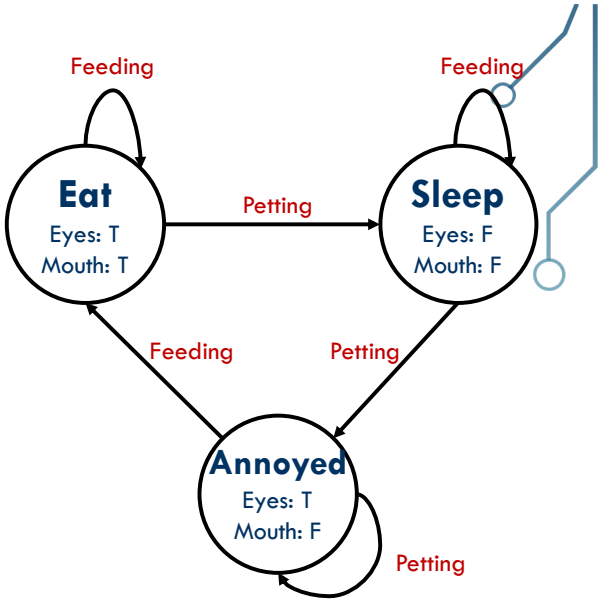
$M =$

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



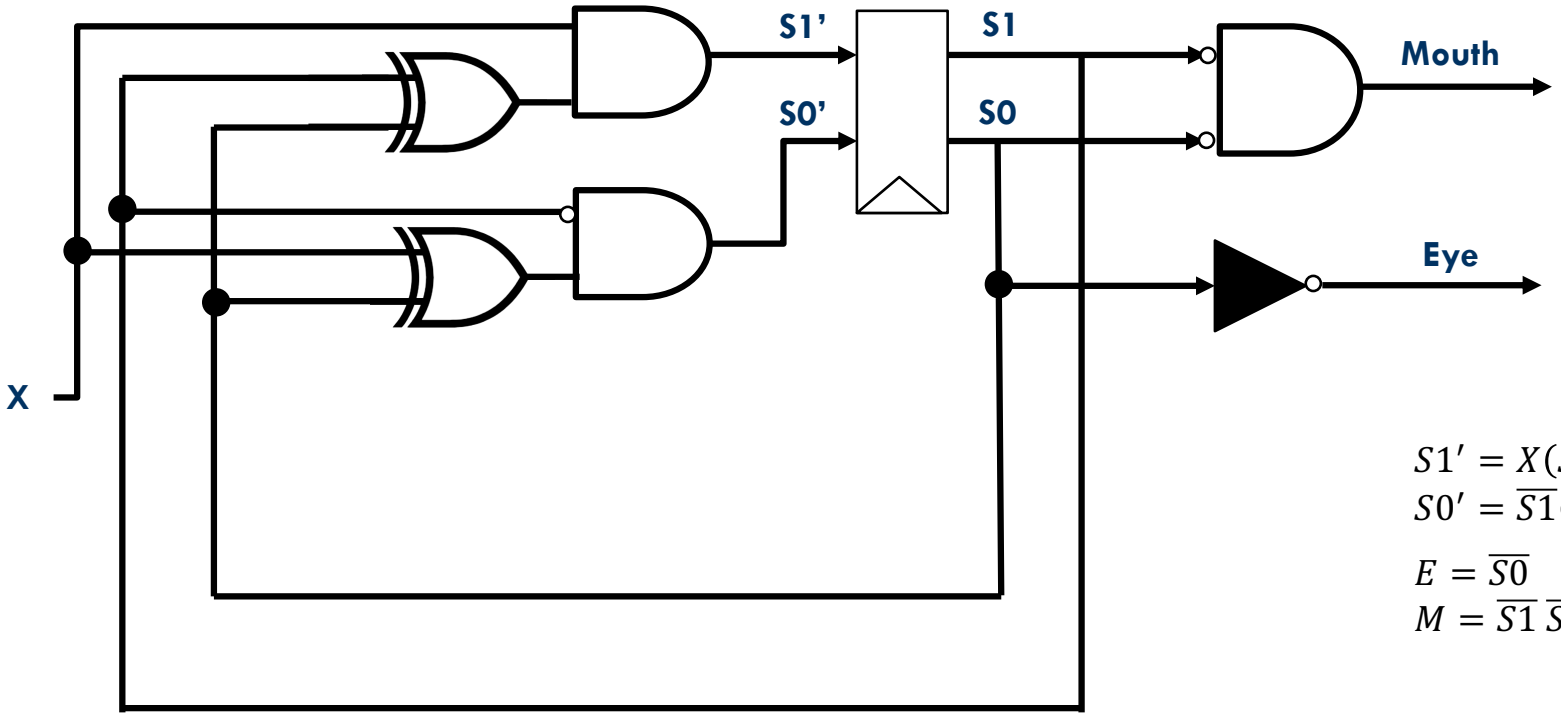
$$E = \overline{S1S0} + S1\overline{S0} = \overline{S0}$$
$$M = \overline{S1}\,\overline{S0}$$

FSM Gate Representation

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

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

FSM 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

FSM State Encoding

- 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

Sequential Logic

Introduction

Finite State Machines

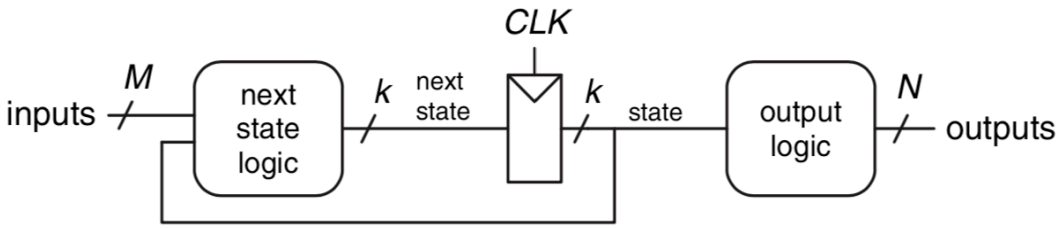
Moore vs Mealy FSM

FSM in Verilog

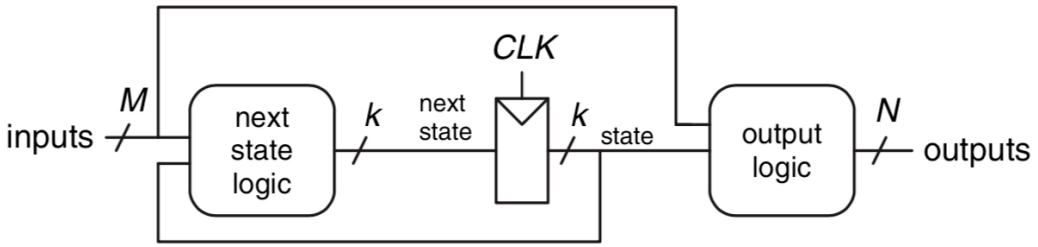
Moore vs Mealy FSMs

- 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

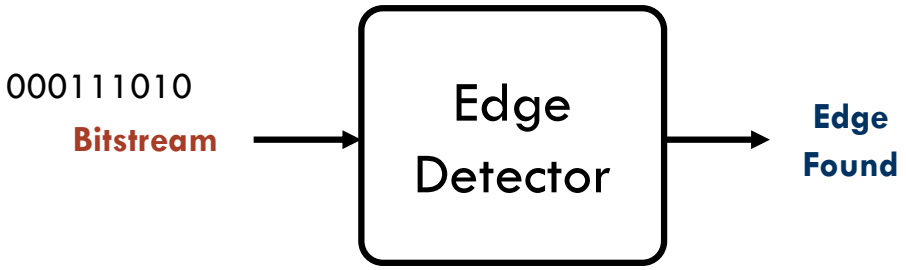


Mealy FSM

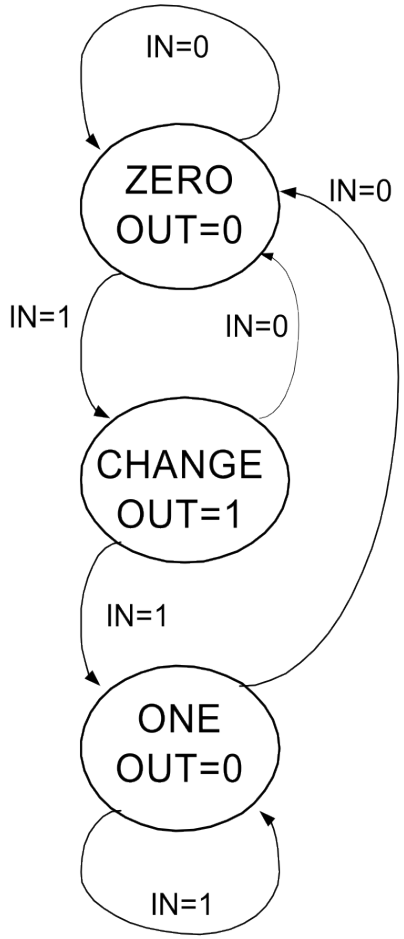


Example: Edge Detector

- 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.

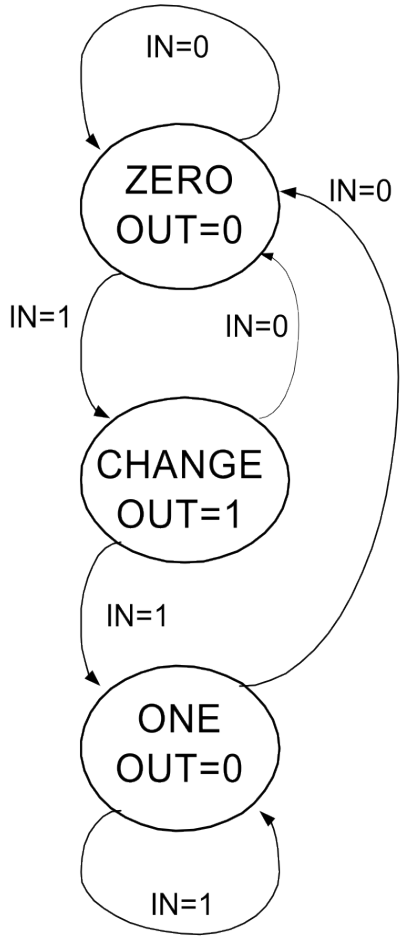


State Transition Diagram Solution A



Input	Current State	Next State	Output
0	Zero (00)	Zero	0

State Transition Diagram Solution A



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

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

NS_1 = IN *AND* CS0

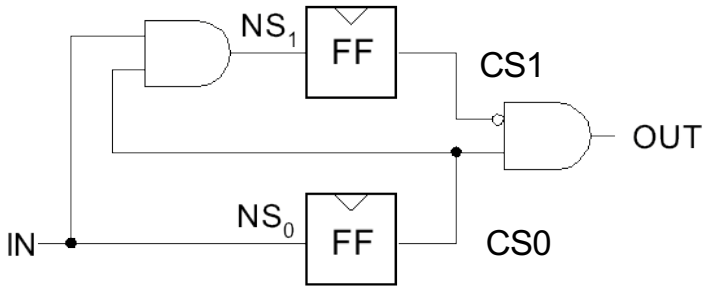
CS / IN	00	01	11	10
0	0	0	0	-
1	1	1	1	-

NS_0 = IN

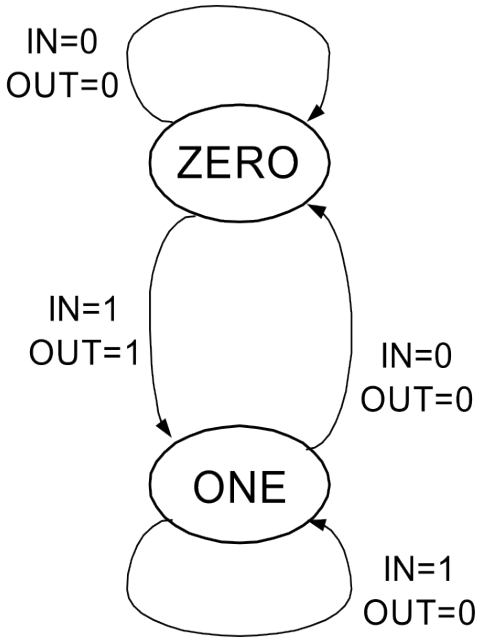
CS / IN	00	01	11	10
0	0	1	0	-
1	0	1	0	-

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

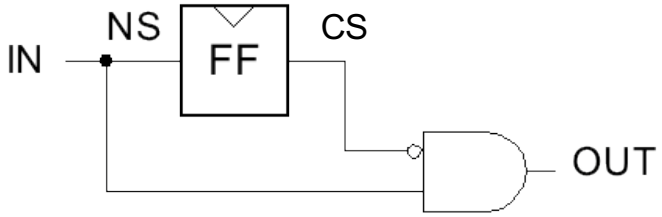
Input	Current State	Next State	Output
0	Zero (00)	Zero	0
1	Zero (00)	Change	0
0	Change (01)	Zero	1
1	Change (01)	One	1
0	One (11)	Zero	0
1	One (11)	One	0



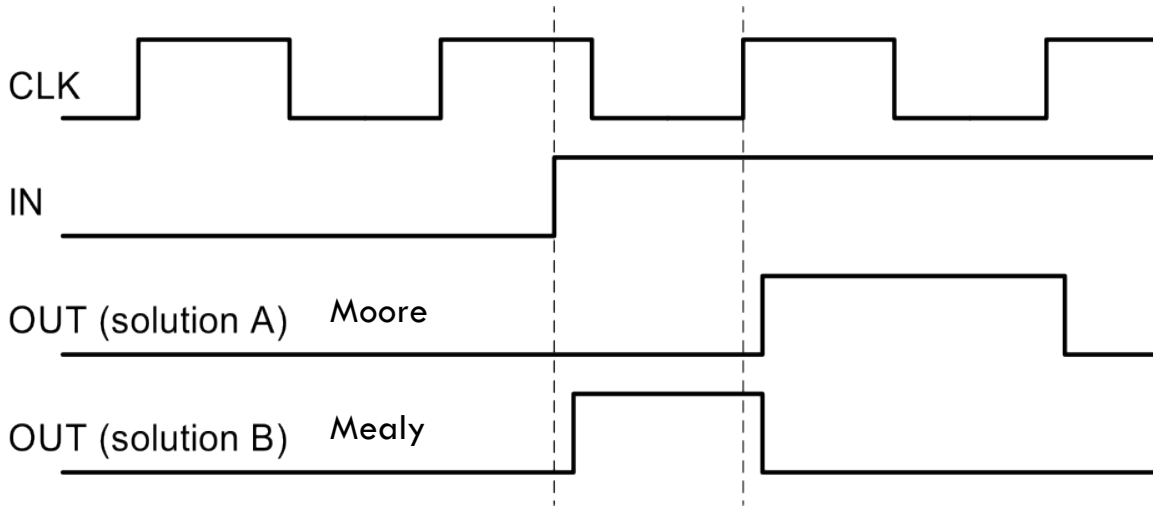
State Transition Diagram Solution B



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 fails synchronous with next clock edge

FSM Comparison

Solution A

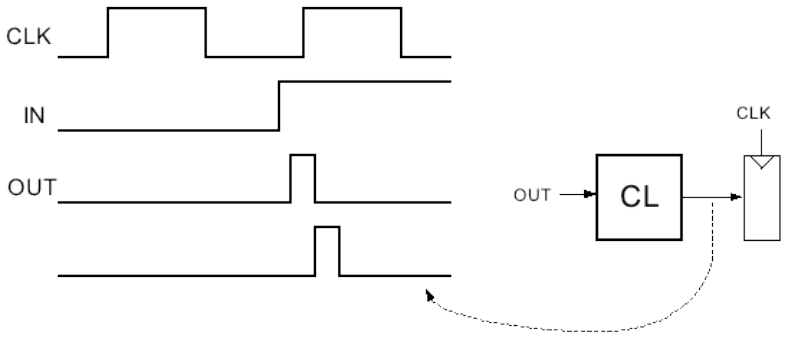
Moore Machine

- 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

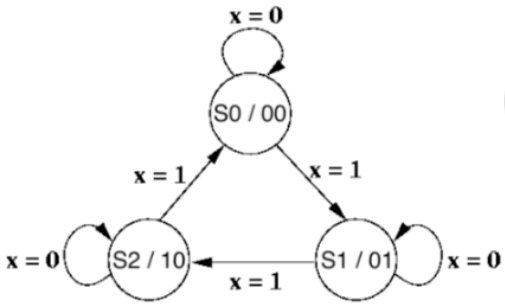
Mealy Machine

- 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):

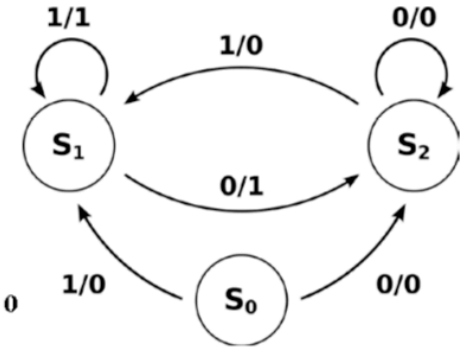


If output of Mealy FSM goes through combinational logic before being registered, the CL might delay the signal and it could be missed by the clock edge (or violate set-up time requirement)

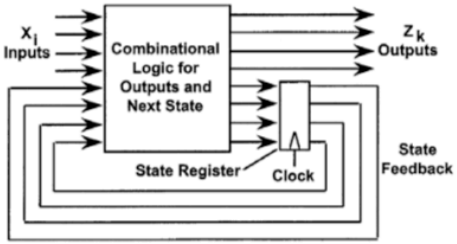
Quiz: Which of the diagrams are **Moore** machines?



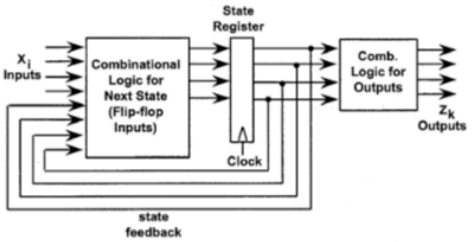
A.



B.



C.



D.

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

www.yellkey.com/century

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

Sequential Logic

Introduction

Finite State Machines

Moore vs Mealy FSM

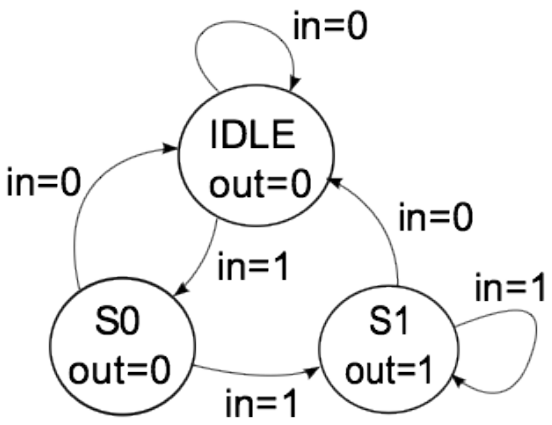
FSM in Verilog

Implement FSM with Verilog

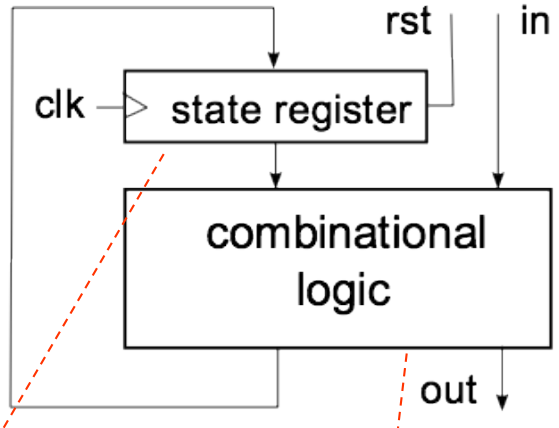
- Specify circuit function
- Draw state transition diagram
- Write down symbolic state transition table
- Assign encodings (bit patterns) to symbolic states
- Code as Verilog behavioral description
  - Use parameters to represent encoded states
  - Use separate always blocks for register assignment and combinational logic block
  - Use case statement for combinational logic.
    - Within each case section (state), assign outputs and next state based on inputs
    - Moore: outputs only dependent on states not on inputs

Finite State Machine in Verilog

State Transition Diagram



Circuit Diagram



Holds a symbol to keep track of which bubble the FSM is in.

CL functions to determine output value and next state based on input and current state.

out = f(in, current state)

next state = f(in, current state)

Finite State Machine in Verilog

```
module FSM1(clk, rst, in, out);
input clk, rst;
input in;
output out;

// Defined state encoding:
parameter IDLE = 2'b00;
parameter S0 = 2'b01;
parameter S1 = 2'b10;
reg out;
reg [1:0] current_state, next_state;

// always block for state register
always @(posedge clk)
   if (rst) current_state <= IDLE;
   else current_state <= next_state;
```

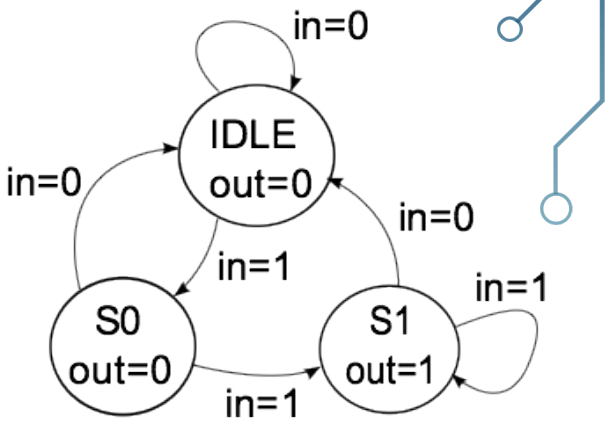
Must use reset to force to initial state.

reset not always shown in STD

Constants local to this module.

Combinational logic signals for transition.

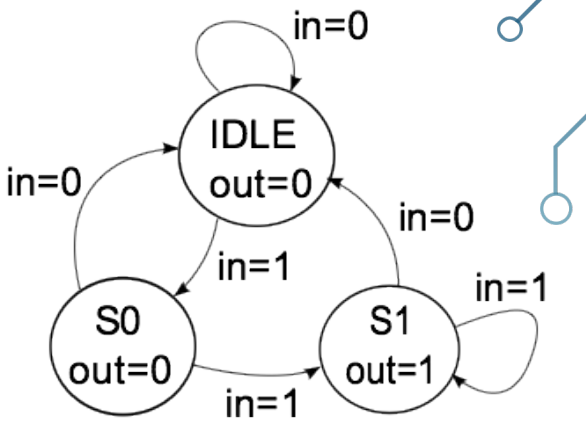
THE register to hold the "state" of the FSM.



A separate always block should be used for combination logic part of FSM. Next state and output generation. (Always blocks in a design work in parallel.)

Finite State Machine in Verilog (cont.)

```
// always block for combinational logic portion
always @(current_state or in)
case (current_state)
// For each state def output and next
  IDLE    : begin
              out = 1'b0;
              if (in == 1'b1) next_state = S0;
              else next_state = IDLE;
            end
  S0      : begin
              out = 1'b0;
              if (in == 1'b1) next_state = S1;
              else next_state = IDLE;
            end
  S1      : begin
              out = 1'b1;
              if (in == 1'b1) next_state = S1;
              else next_state = IDLE;
            end
  default: begin
             next_state = IDLE;
             out = 1'b0;
           end
endcase
endmodule
```



Each state becomes a case clause.

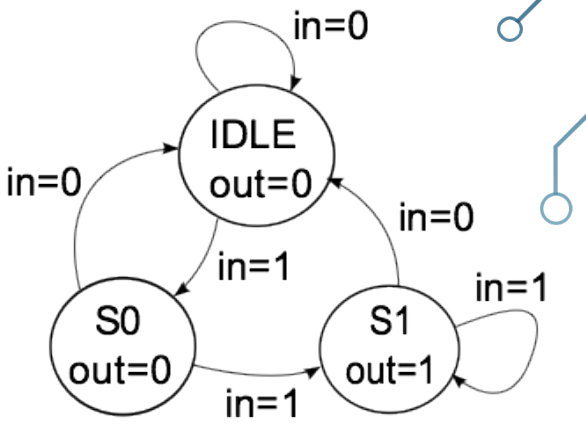
For each state define:

Output value(s)

State transition

Use "default" to cover unassigned state. Usually unconditionally transition to reset state.

Finite State Machine in Verilog (cont.)




```
always @*
  begin
    next_state = IDLE;
    out = 1'b0;
    case (state)
      IDLE    : if (in == 1'b1) next_state = S0;
      S0      : if (in == 1'b1) next_state = S1;
      S1      : begin
                  out = 1'b1;
                  if (in == 1'b1) next_state = S1;
                end
      default: ;
    endcase
  end
endmodule
```

* for sensitivity list

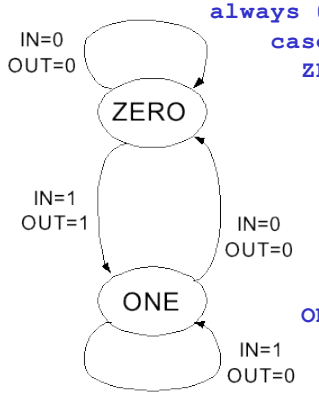
Normal values: used unless specified below.

Within case only need to specify exceptions to the normal values.

Note: The use of "blocking assignments" allow signal values to be "rewritten", simplifying the specification.

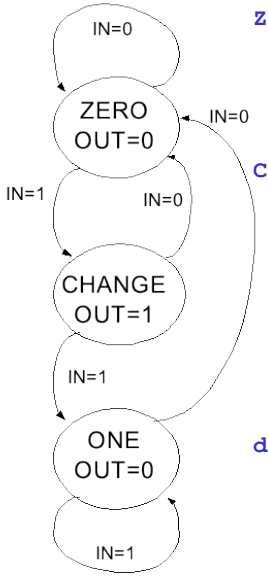
Edge Detector Example

Mealy Machine




```
always @(posedge clk)
    if (rst) ps <= ZERO;
    else ps <= ns;
always @(ps in)
    case (ps)
      ZERO: if (in) begin
              out = 1'b1;
              ns = ONE;
            end
          else begin
            out = 1'b0;
            ns = ZERO;
          end
      ONE: if (in) begin
            out = 1'b0;
            ns = ONE;
           end
           else begin
             out = 1'b0;
             ns = ZERO;
           end
      default: begin
                 out = 1'bx;
                 ns = default;
               end
```

Moore Machine




```
always @(posedge clk)
    if (rst) ps <= ZERO;
    else ps <= ns;
always @(ps in)
    case (ps)
      ZERO: begin
          out = 1'b0;
          if (in) ns = CHANGE;
                  else ns = ZERO;
        end
      CHANGE: begin
          out = 1'b1;
          if (in) ns = ONE;
          else ns = ZERO;
        end
      ONE: begin
          out = 1'b0;
          if (in) ns = ONE;
          else ns = ZERO;
      default: begin
          out = 1'bx;
          ns = default;
        end
```