ECE 2300
Digital Logic & Computer Organization

Fall 2016

More Verilog
Finite State Machines

Cornell University

Verilog Programming Styles

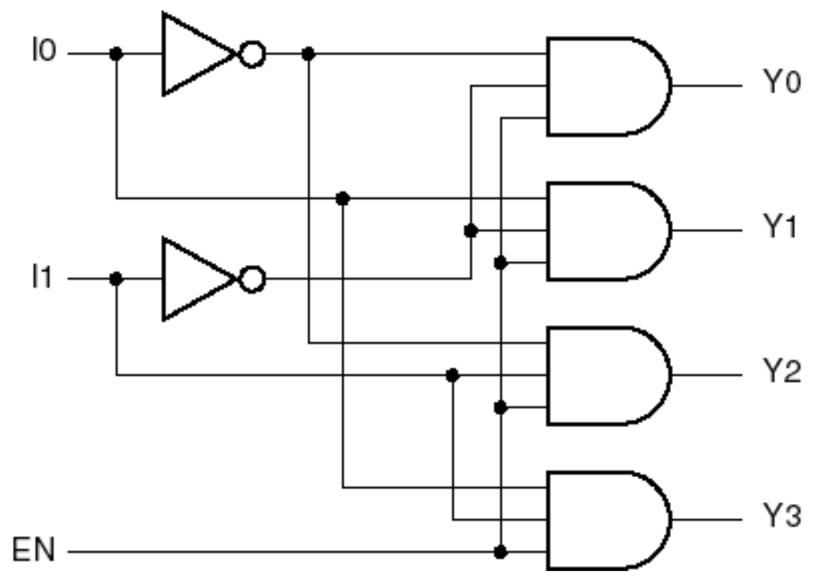
- Structural
  - Textual equivalent of drawing a schematic
  - Uses *instance* statements
- Dataflow
  - Describes circuit in terms of flow of data and operations on that data
  - Uses *continuous-assignment* statements
  - Called “structural” in textbook
- Behavioral
  - High-level algorithmic description
  - Uses *procedural code*

Structural Style

```
module V2to4dec_struct( i0,i1,en,y0,y1,y2,y3 );
   input i0,i1,en;
   output y0,y1,y2,y3;
   wire noti0,noti1;

   not U1(noti0,i0);
   not U2(noti1,i1);
   and U3(y0,noti0,noti1,en);
   and U4(y1,    i0,noti1,en);
   and U5(y2,noti0,    i1,en);
   and U6(y3,    i0,    i1,en);
endmodule
```

Correspondence with schematic



Dataflow Style

```
module V2to4dec_df( i0,i1,en,y0,y1,y2,y3 );
  input i0,i1,en;
  output y0,y1,y2,y3;

  assign y0 = en & ~i0 & ~i1;
  assign y1 = en &  i0 & ~i1;
  assign y2 = en & ~i0 &  i1;
  assign y3 = en &  i0 &  i1;
endmodule
```

Correspondence with Boolean logic

Only used for combination logic

New value is assigned to the lhs whenever any value on the rhs changes

Behavioral Style

```
module V2to4dec_beh( i0,i1,en,y0,y1,y2,y3 );
   input i0,i1,en;
   output reg y0,y1,y2,y3;

   always @ (en, i0, i1)
   begin
     y0 = en & ~i0 & ~i1;
     y1 = en &  i0 & ~i1;
     y2 = en & ~i0 &  i1;
     y3 = en &  i0 &  i1;
   end
endmodule
```

- **Procedural code**
- **Key element is the *always block***
- **Can be used for both combinational and sequential logic**
- reg **needed for variables on lhs of procedure statements**

Always Blocks

- *Always* block starts execution when the value of any signal in the *sensitivity list* changes
- Execution continues until there are no more changes in sensitivity list
- *Always* blocks execute concurrently with other *always* blocks, instance statements, and continuous assignment statements in a module

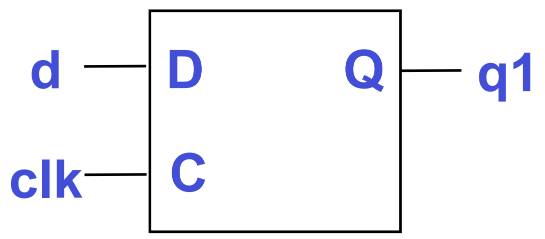
Sequential Logic Using Always Blocks

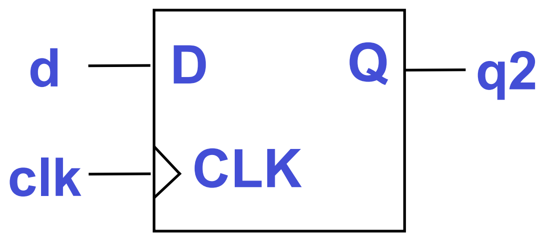
- Sequential logic can only be modeled using always blocks

```
reg q1, q2;

always @( clk, d )
begin
  if ( clk )
     q1 <= d;
end

always @( posedge clk )
begin
  q2 <= d;
end
```





Blocking and Nonblocking Assignments

- **Blocking statement**
  - **A = B & C;**
  - **Assignment is immediate**

- **Nonblocking statement**
  - **A <= B & C;**
  - **Assignment is delayed until end of *always* block**

```
always
begin
  A = B & C;
  D = ~A;
end
```

A = B & C
D = ~(B & C)

```
always
begin
  A <= B & C;
  D <= ~A;
end
```

A = B & C   D = ~A

General Guidelines

- Use blocking (=) assignments in *always* blocks intended as combinational logic

- Use nonblocking (<=) assignments in *always* blocks intended as sequential logic

- Do not mix blocking and nonblocking assignments in the same *always* block

- Do not make assignments to the same variable in two different *always* blocks

Inferred Latches In Comb Logic

- **Each variable within an *always* block should get assigned a value under all possible conditions**
  - **Otherwise, the compiler assumes that the last value should be used, and will create a latch**

```
reg dout;
always @(din, c)
begin
 /* dout not always assigned
     a value; latch inferred */
  if (c == 1'b1)
    dout = din;
end
```

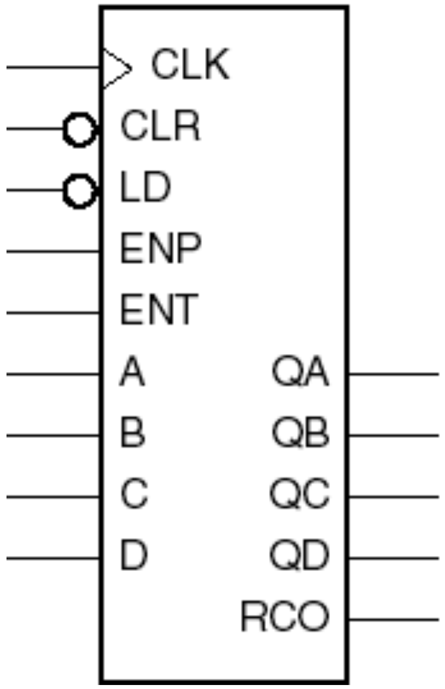
```
reg dout;
always @(din, c)
begin
   /* dout assigned a value in both
      conditions, latch not inferred */
  if (c == 1'b1)
    dout = din;
  else
    dout = ~din;
end
```

Verilog Code for ’163 Counter

```
module counter (CLK, CLR_L, LD_L, ENP, ENT, D, Q, RC0);
  input CLK, CLR_L, LD_L, ENP, ENT;
  input [3:0] D;
  output reg [3:0] Q;
  output reg RC0;

  always @ (posedge CLK)
    if (CLR_L ==0)                          Q <= 4'b0;
    else if (LD_L == 0)                     Q <= D;
    else if ((ENT ==1) && (ENP ==1))        Q <= Q+1;
    else                                    Q <= Q;

  always @ (Q, ENT)
    if ((ENT == 1) && (Q == 4'd15))         RC0 = 1;
    else                                    RC0 = 0;
endmodule
```



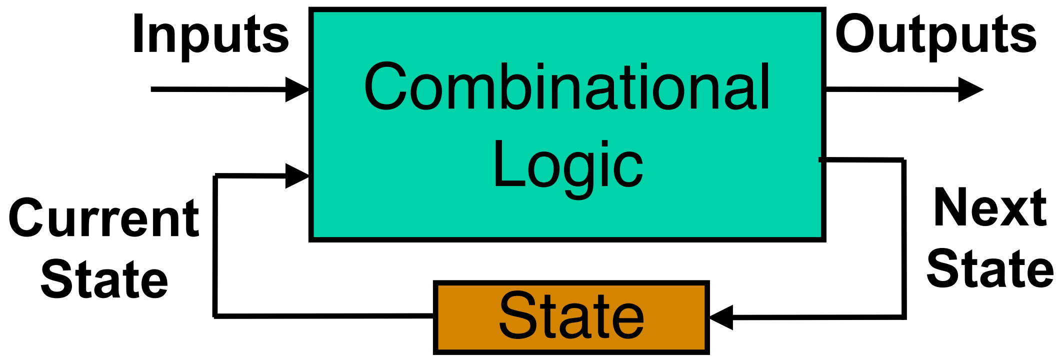
Sequential Logic

- Many logic functions require information about the past in addition to the current inputs
  - Vending machine controller
  - Traffic light system
  - Microprocessor control unit

- Such *sequential logic circuits* are implemented using combinational logic and storage

- One important sequential logic circuit is a *Finite State Machine (FSM)*

Finite State Machine



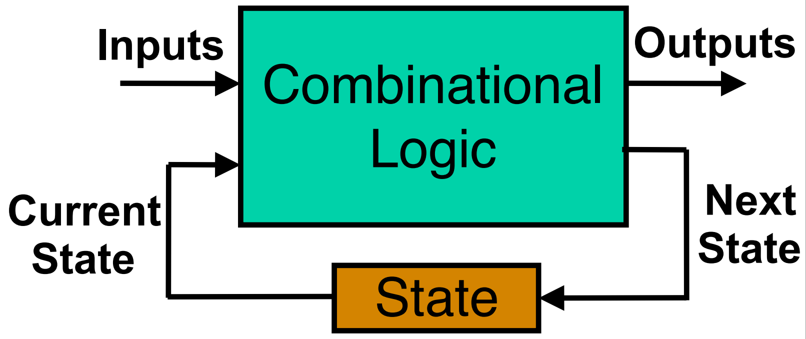
- The <u>state</u> embodies the condition of the system at this particular time

- The combinational logic determines the output and next state values

- The output values may depend only on the current state value, or on the current state and input values

Elements of a FSM

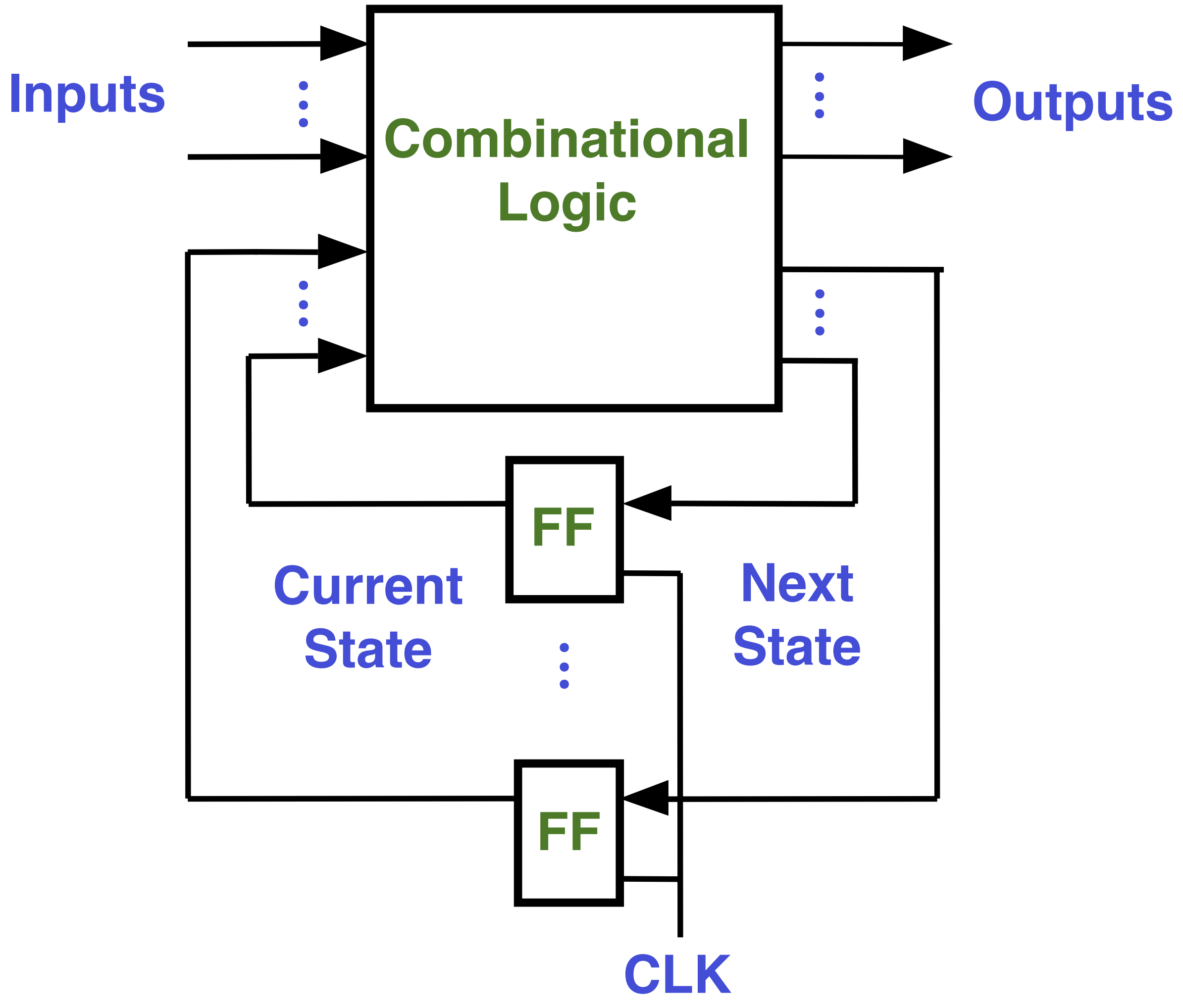
1. A finite number of states
2. A finite number of inputs
3. A finite number of outputs
4. A specification of all state transitions
5. A specification of the output values



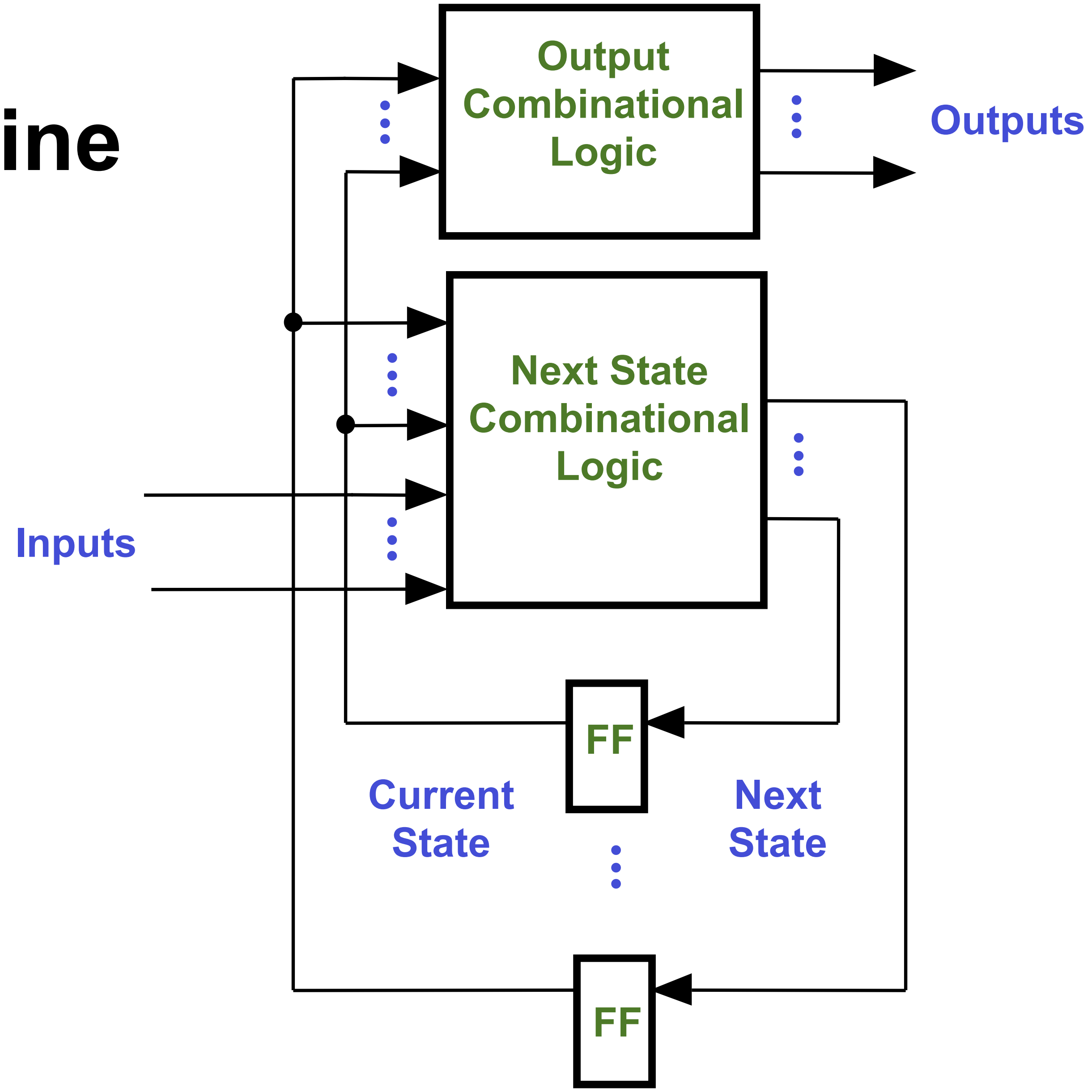
Described by a state diagram

- Inputs and current state trigger state transitions
- Output changes triggered by changes in
  - Current state, or
  - Current state + inputs

FSM: General Form

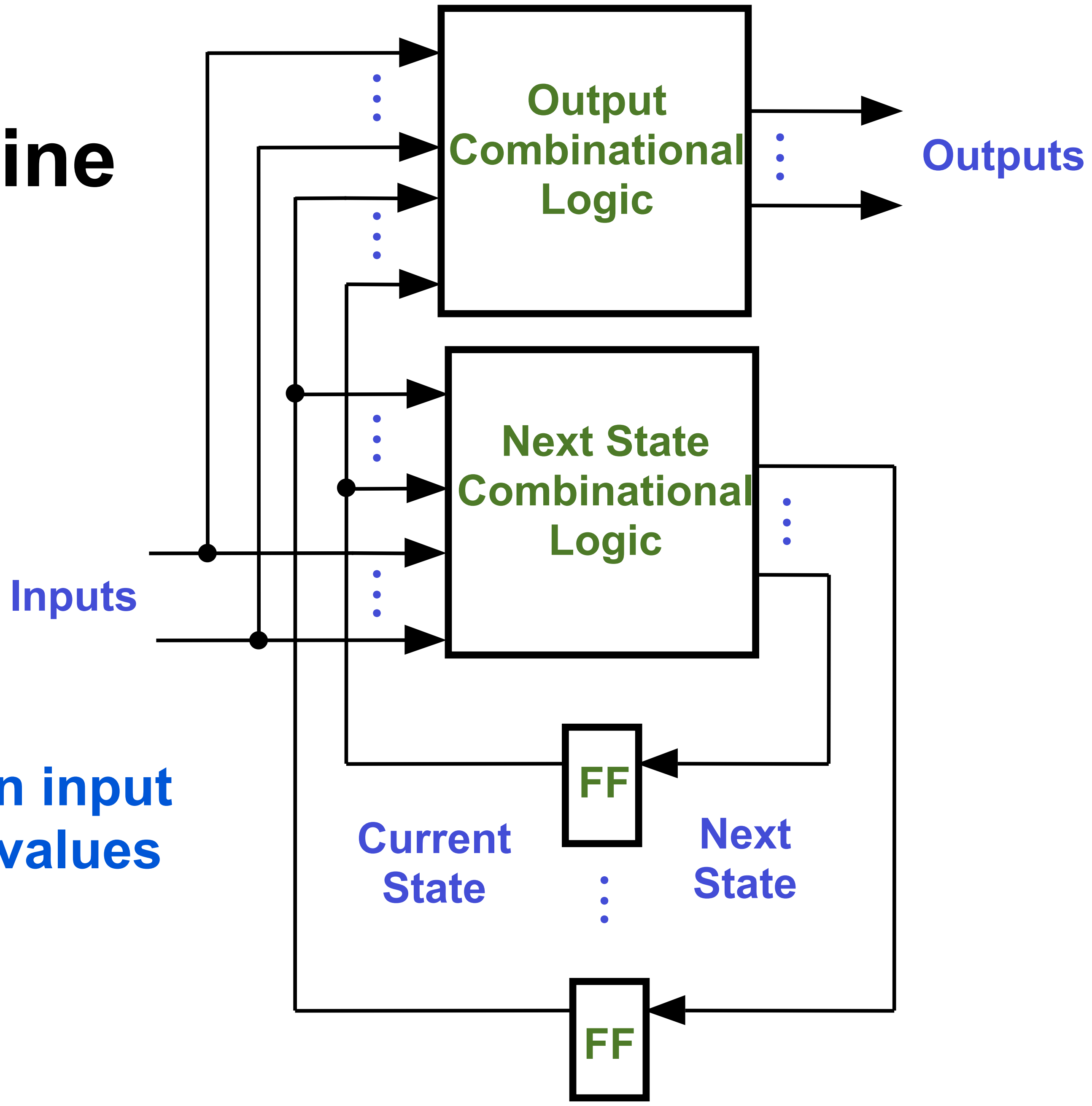


Moore Machine



Outputs depend on current state value

Mealy Machine



Outputs depend on input and current state values

FSM Design Procedure

(1) Understand the problem statement and determine inputs and outputs

(2) Identify states and create a *state diagram*

(3) Determine the number of required D FFs

(4) Implement combinational logic for outputs and next state

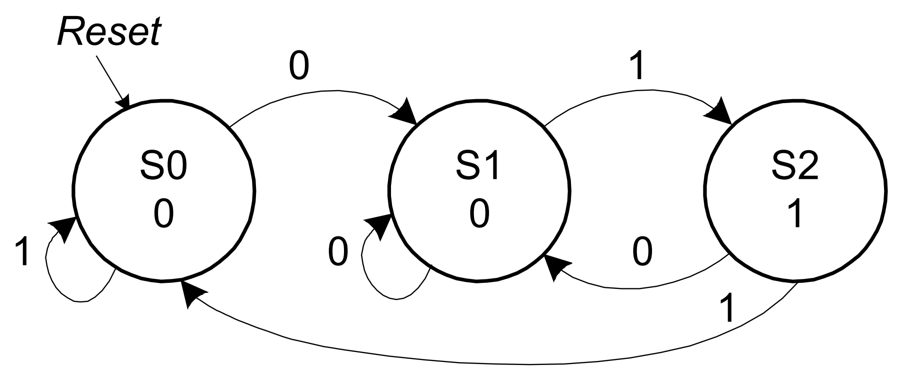
(5) Simulate the circuit to test its operation

State Diagram

- Visual specification of a FSM
- Bubble for every state
- Arcs showing state transitions
- Input values shown on the arcs
- Output values shown within the bubbles (Moore) or on the arcs (Mealy)
- Clock input not shown (always present)

Moore State Diagram

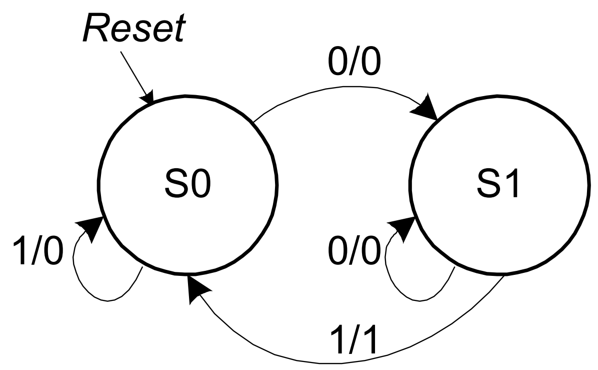
Moore FSM



- **1 input, 1 output, 3 states**
- **Bubble for each state**
- **State transitions (arcs) for each input value**
- **Input values on the arcs**
- **Output values within the bubbles**
- **Starts at S0 when Reset asserted**

Mealy State Diagram

Mealy FSM



- **1 input, 1 output, 2 states**
- **Bubble for each state**
- **State transitions (arcs) for each input value**
- **Input values on the arcs (first number)**
- **Output values on the arcs (second number)**
- **Starts at S0 when Reset asserted**

Example FSM: Pattern Detector

- Monitors the input, and outputs a 1 whenever a specified input pattern is detected

- Example: Output a 1 whenever 111 is detected on the input for 3 consecutive clock cycles
  - Overlapping patterns also detected (1111...)

- Input *In*
- Output *Out*
- *Reset* causes FSM to start in initial state
- *Clock* input not shown (always present)

111 Pattern Detector: Moore State Diagram

111 Pattern Detector: Mealy State Diagram

Example FSM: Pushbutton Lock

- Two pushbutton inputs, X1 and X2
- One output, UL (“Unlock”)
- UL = 1 when X1 is pushed, followed by X2 being pushed twice (X1, X2, X2)
- Represent X1 and X2 as two bit input
  - 00: neither button pushed
  - 01: X2 pushed
  - 10: X1 pushed
  - 11: both pushed simultaneously

Pushbutton Lock: Moore State Diagram

Pushbutton Lock: Mealy State Diagram

Next Time

More Finite State Machines