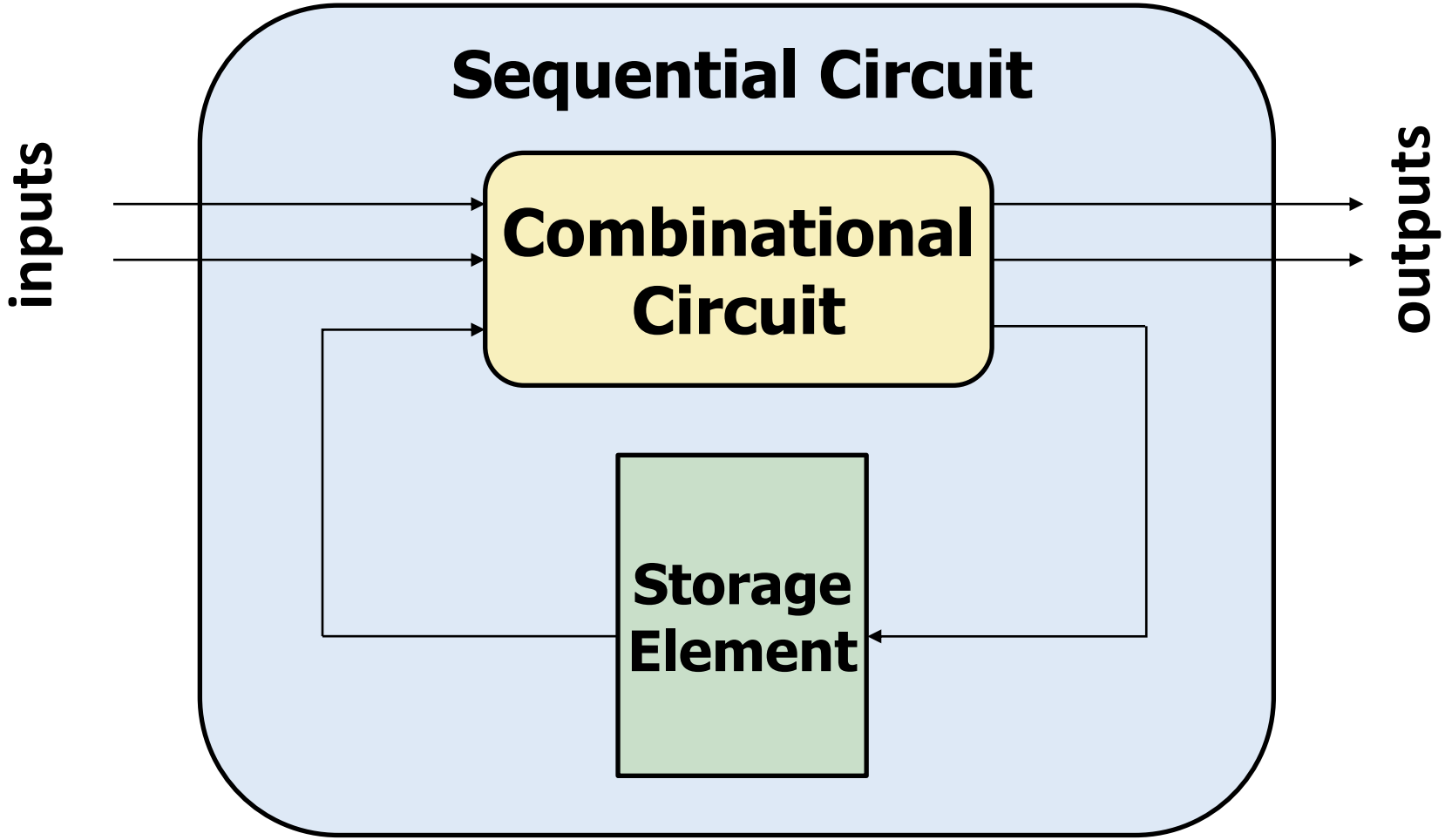
Hardware Description Languages and Verilog

陈刚
副教授
人工智能与无人系统研究所
计算机学院
中山大学

Constructing State Machines in Verilog

Combinational + Memory = Sequential



Sequential Logic Circuits

- We have examined designs of circuit elements that can **store information**
- Now, we will use these elements to build circuits that **remember** past inputs

Combinational
Only depends on current inputs

Sequential
Opens depending on past inputs

https://www.easykeys.com/228_ESP_Combination_Lock.aspx
https://www.fosmon.com/product/tsa-approved-lock-4-dial-combo

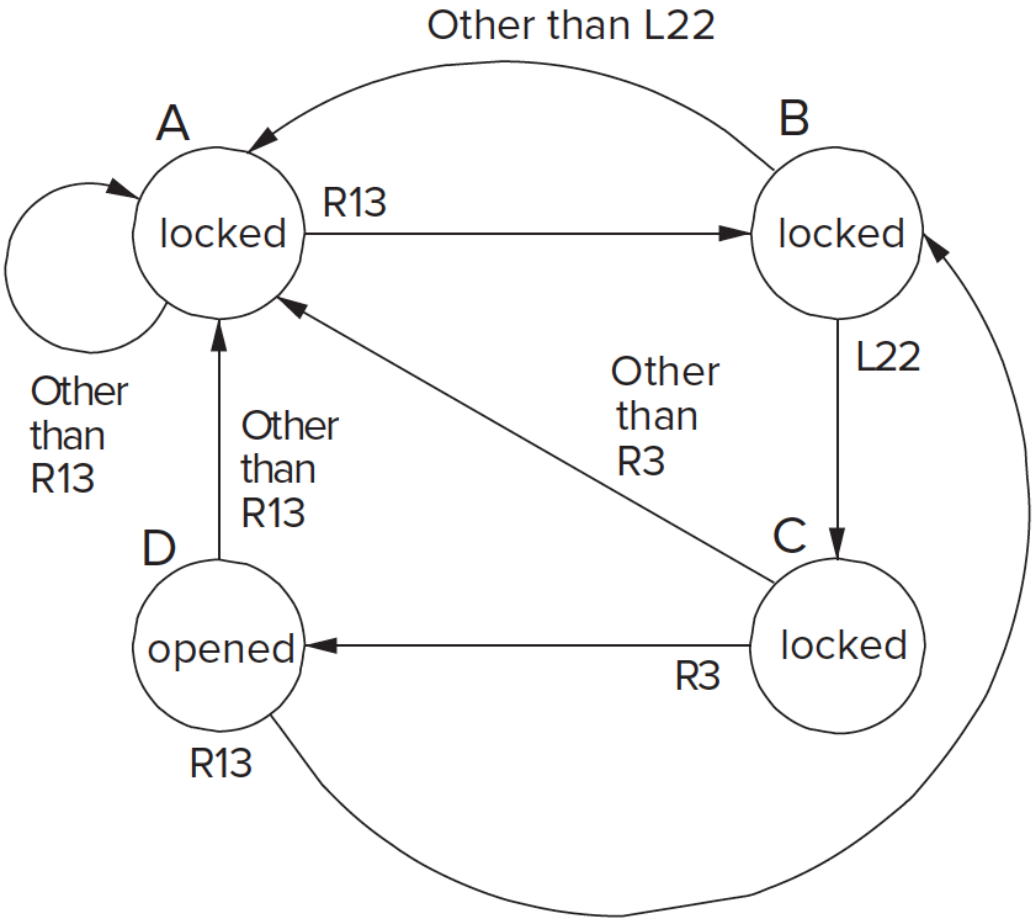
State

- In order for this lock to work, it has to keep track (**remember**) of the past events!
- If passcode is **R13-L22-R3**, sequence of **states** to unlock:
  A. The lock is not open (locked), and no relevant operations have been performed
  B. Locked but user has completed R13
  C. Locked but user has completed R13-L22
  D. Unlocked: user has completed R13-L22-R3

- The **state** of a system is a snapshot of all relevant elements of the system at the moment of the snapshot
  - To open the lock, **states A-D must be completed in order**
  - If anything else happens (e.g., L5), lock **returns** to state A

State Diagram of Our Sequential Lock

- Completely describes the operation of the sequential lock

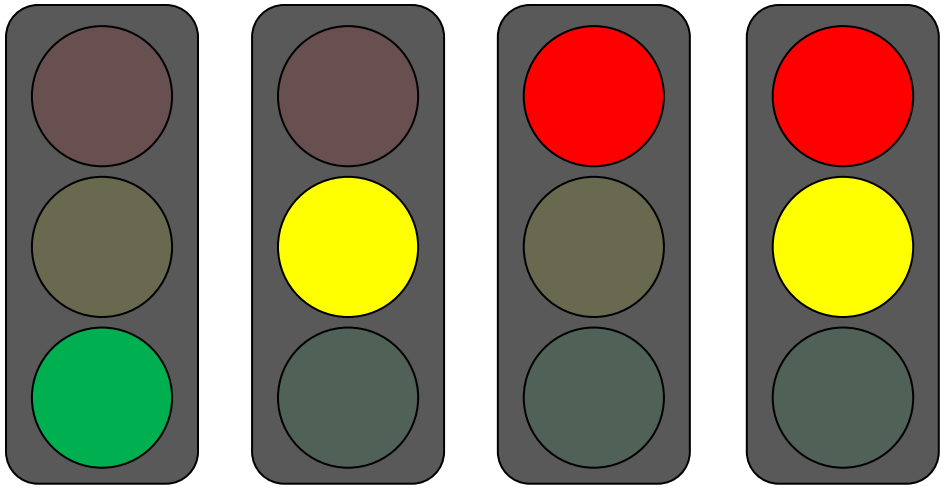


- We will understand "state diagrams" fully later today

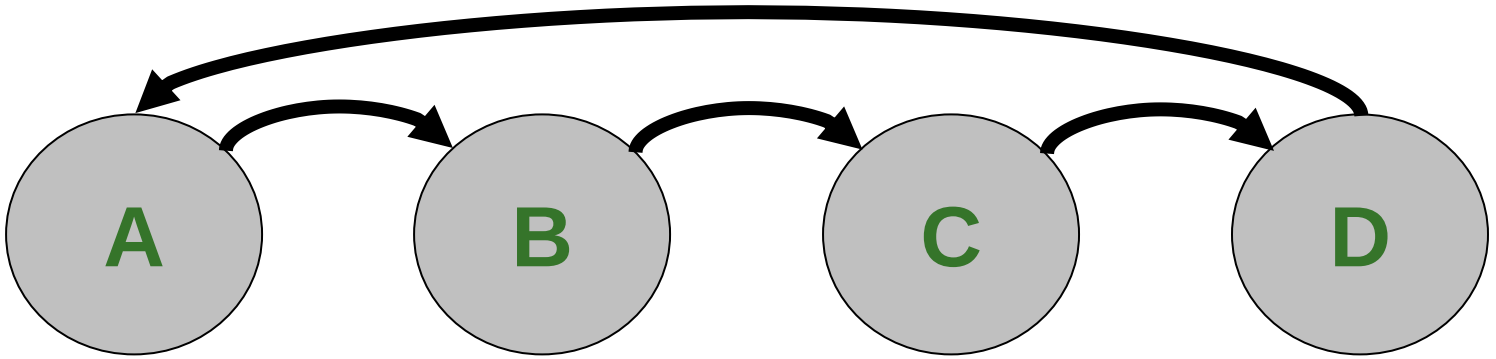
Image source: Patt and Patel, "Introduction to Computing Systems", 2nd ed., page 76.

Another Simple Example of State

- A standard Swiss traffic light has **4 states**
  A. Green
  B. Yellow
  C. Red
  D. Red and Yellow



- The sequence of these states are always as follows



Finite State Machines

- What is a **Finite State Machine** (FSM)?
  - **A discrete-time model** of a stateful system
  - Each state represents a snapshot of the system at a given time

- An FSM pictorially shows
  1. the set of all possible **states** that a system can be in
  2. how the system transitions from one state to another

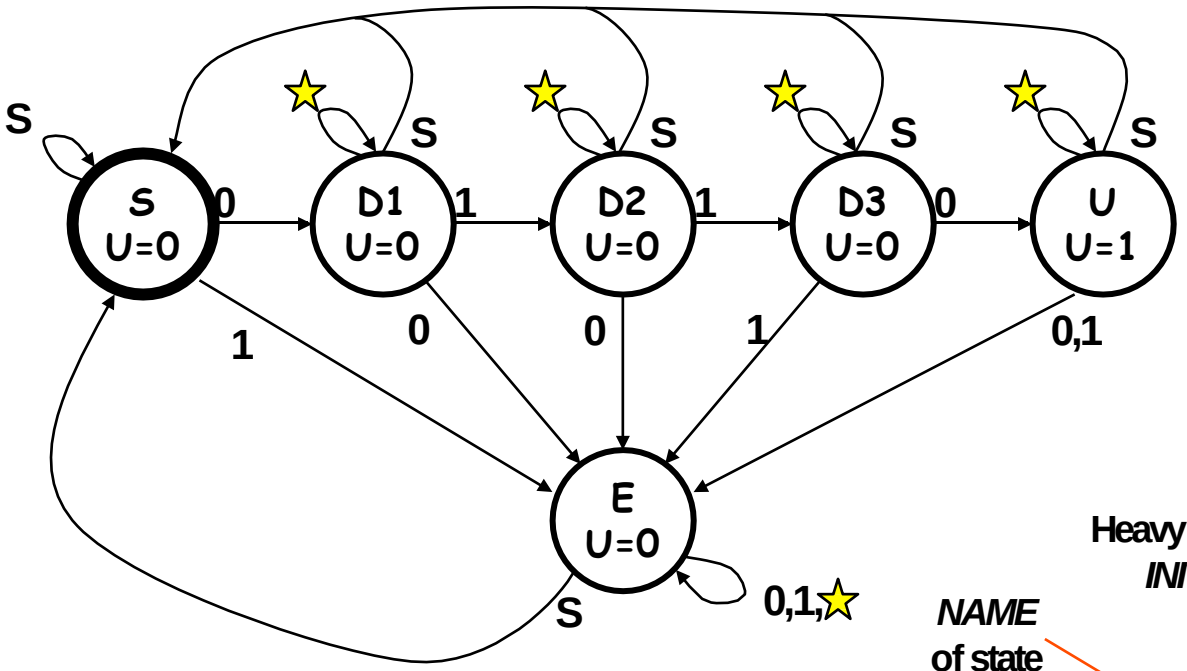
- An FSM can model
  - A traffic light, an elevator, fan speed, a microprocessor, etc.

- **An FSM enables us to pictorially think of a stateful system using simple diagrams**

Finite State Machines (FSMs) Consist of:

- **Five elements:**
  1. A **finite** number of states
     - ***State***: snapshot of all relevant elements of the system at the time of the snapshot
  2. A **finite** number of external inputs
  3. A **finite** number of external outputs
  4. An explicit specification of all state transitions
     - How to get from one state to another
  5. An explicit specification of what determines each external output value

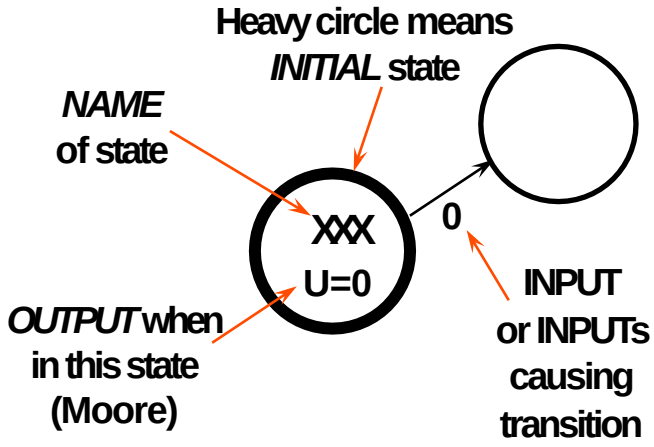
State Transition Diagrams



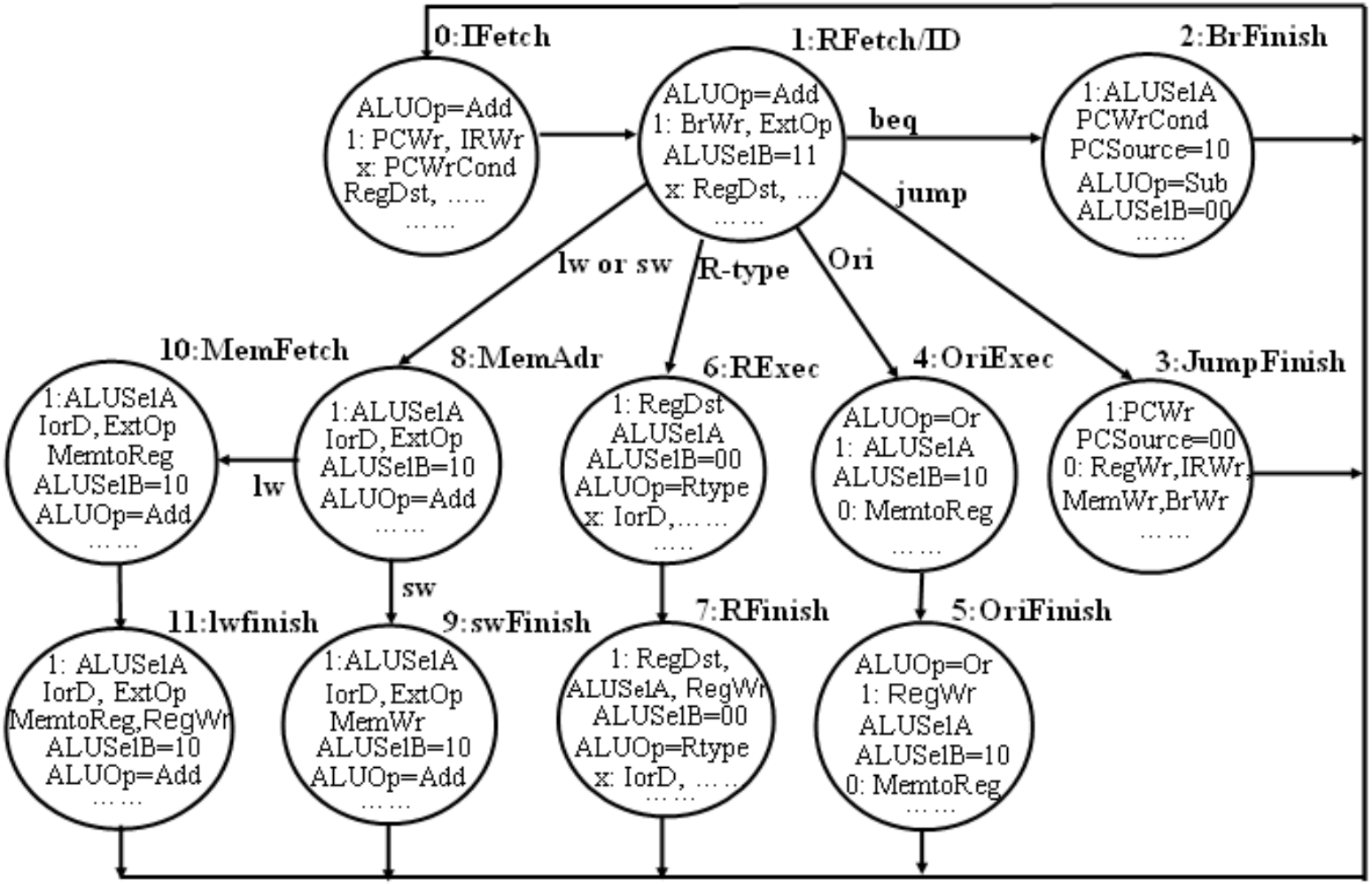
☆ = no buttons pressed

A state transition diagram is an abstract "graph" representation of a "state transition table", where each state is represented as a node and each transition is represented as a as an arc.

It represents the machine's behavior not its implementation!



Another Example : CPU controller



Constructing State Machines in Verilog

- We need register to hold
- the current state
  - always @(posedge clk) block

- We need next state function
  - Where do we go from each state given the inputs
  - state by state case analysis
    - next state determined by current state and inputs
- We need the output function
  - State by state analysis
  - Moore: output determined by current state only
  - Mealy: output determined by current state and inputs

State Register

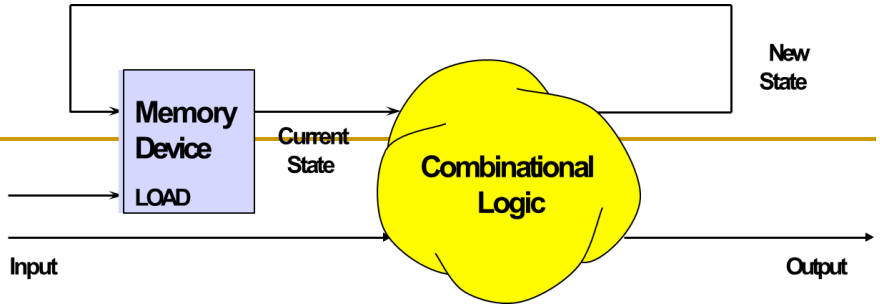
- Declare two values
  - state : current state – output of state register
  - nxtState : next state – input to state register
  - We rely on next state function to give us nxtState

- Declare symbols for states with state assignment

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

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

State Register



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

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

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

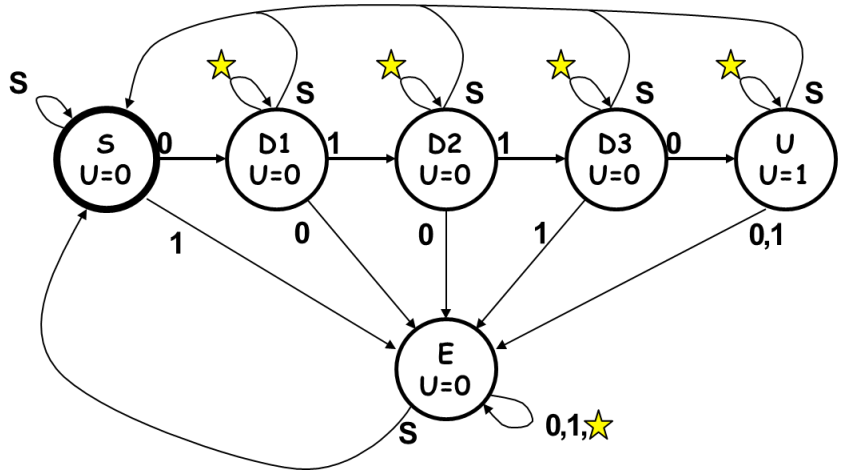
Next State Function

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

Next State Function

- Describe what happens in each state
- Case statement is natural for this

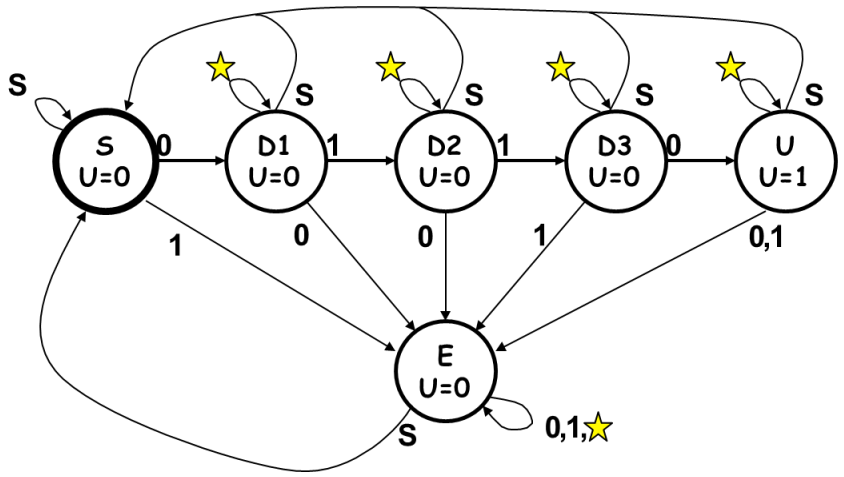
```
always @(*) begin
  nxtState = state;  // Default next state: don't move
  case (state)
    IDLE : begin
      if (B) nxtState = ERROR;
      else if (A) nxtState = WAITFORB;
    end
    WAITFORB : begin
      if (B) nxtState = DONE;
    end
    DONE : begin
    end
    ERROR : begin
    end
  endcase
end
```



Output Function

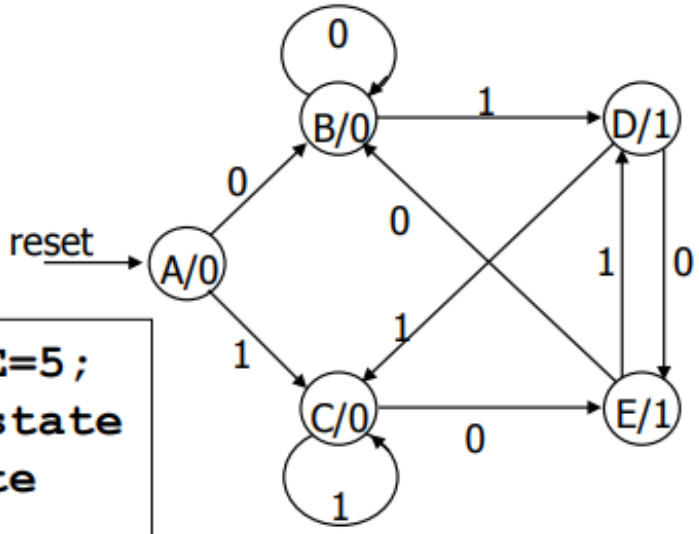
- Describe the output of each state

```
always @(*) begin
  nxtState = state;  // Default next state: stay where we are
  out = 0;           // Default output
  case (state)
    IDLE : begin
      if (B) nxtState = ERROR;
      else if (A) nxtState = WAITFORB;
    end
    WAITFORB : begin
      if (B) nxtState = DONE;
    end
    DONE : begin
      out = 1;
    end
    ERROR : begin
    end
  endcase
end
```



Edge Detector (Moore)

- Describe the output of each state

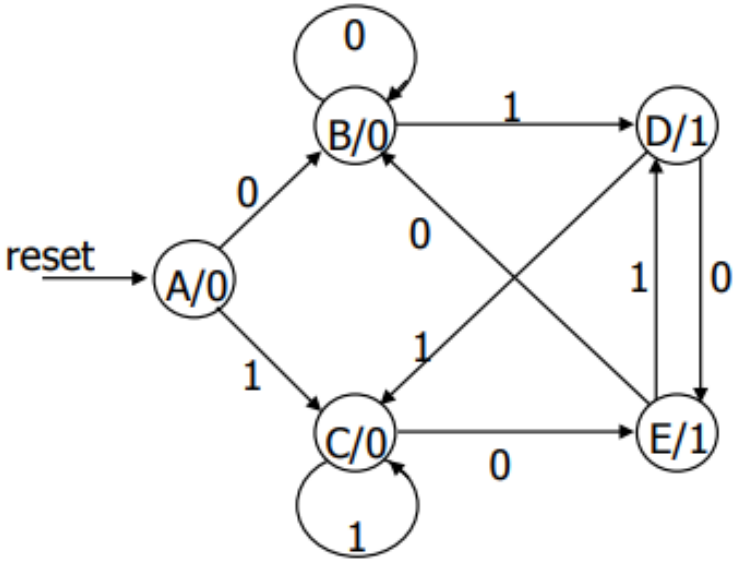



```
localparam A=0, B=1, C=2, D=4, E=5;
reg [2:0] state,    // Current state
          nxtState; // Next state

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

Edge Detector (Moore)

```
always @(*) begin
  nxtState = state;
  out = 0;
  case (state)
    A : if (in) nxtState = C;
        else    nxtState = B;
    B : if (in) nxtState = D;
    C : if (~in) nxtState = E;
    D : begin
      out = 1;
      if (in) nxtState = C;
      else    nxtState = E;
    end
    E : begin
      out = 1;
      if (in) nxtState = D;
      else    nxtState = B;
    end
    default : begin
      out = 1'bX;
      nxtState = 3'bX;
    end
  endcase
end
```



Summary

- **Please use “standard” FSM Verilog style shown here**
  - Do not be beguiled into thinking this is programming C!
- **always @(posedge clk) block**
  - Use only for registers with simple logic
  - e.g. shifter, counter, enabled register, etc.
- **Miscellaneous combinational logic**
  - assign statements (always safe)
  - always @(*) block (be very careful)
    - Think of this as a complex assign statement
    - Fine to have more than one