Digital System Design

Hajar Falahati

hfalahati@ipm.ir
hfalahati@ce.sharif.edu

Combinational Logic & Verilog

- Combinational Logic
  - Use **always block** + **“blocking” assignments**
    - Normally for high-complexity Comb. Logic
    - When output depends on several conditions, which requires if-else

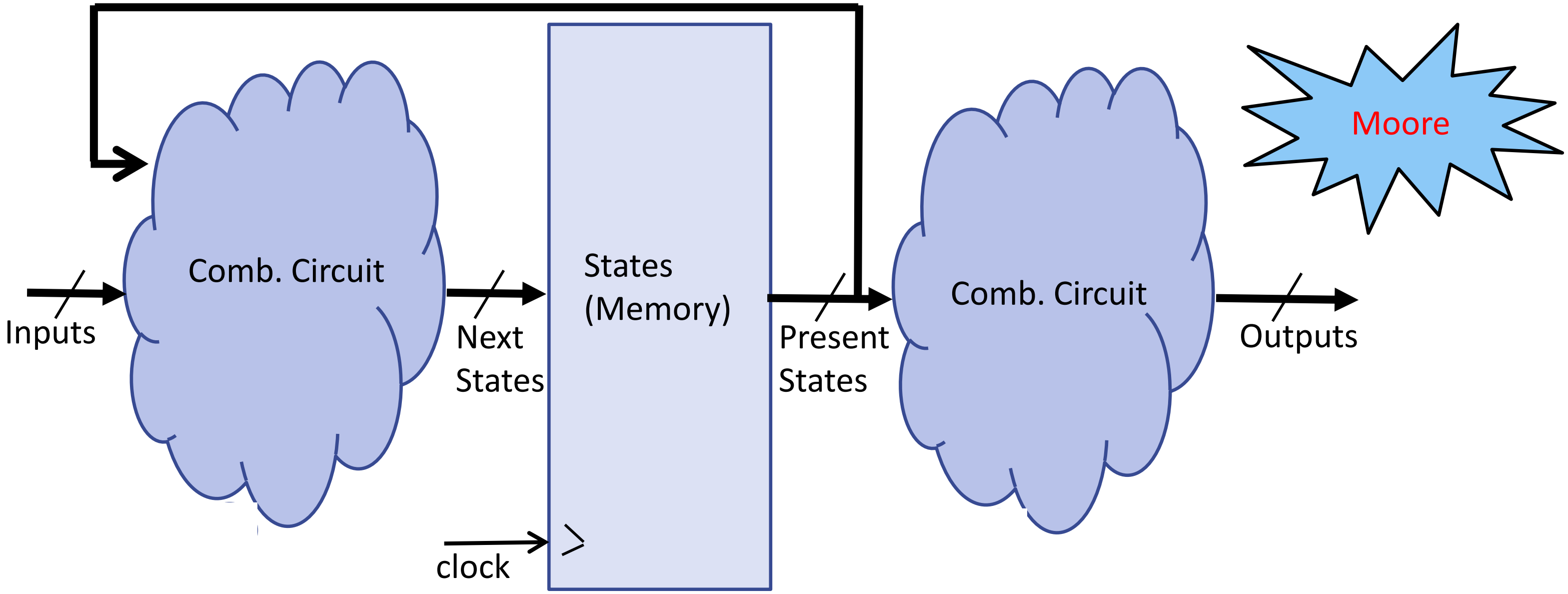
- Sequential Logic
  - Can **only** be realized using an **always block**
  - When using the **always block** for the sequential Logic, **“Non-blocking” assignments** are used

Outline

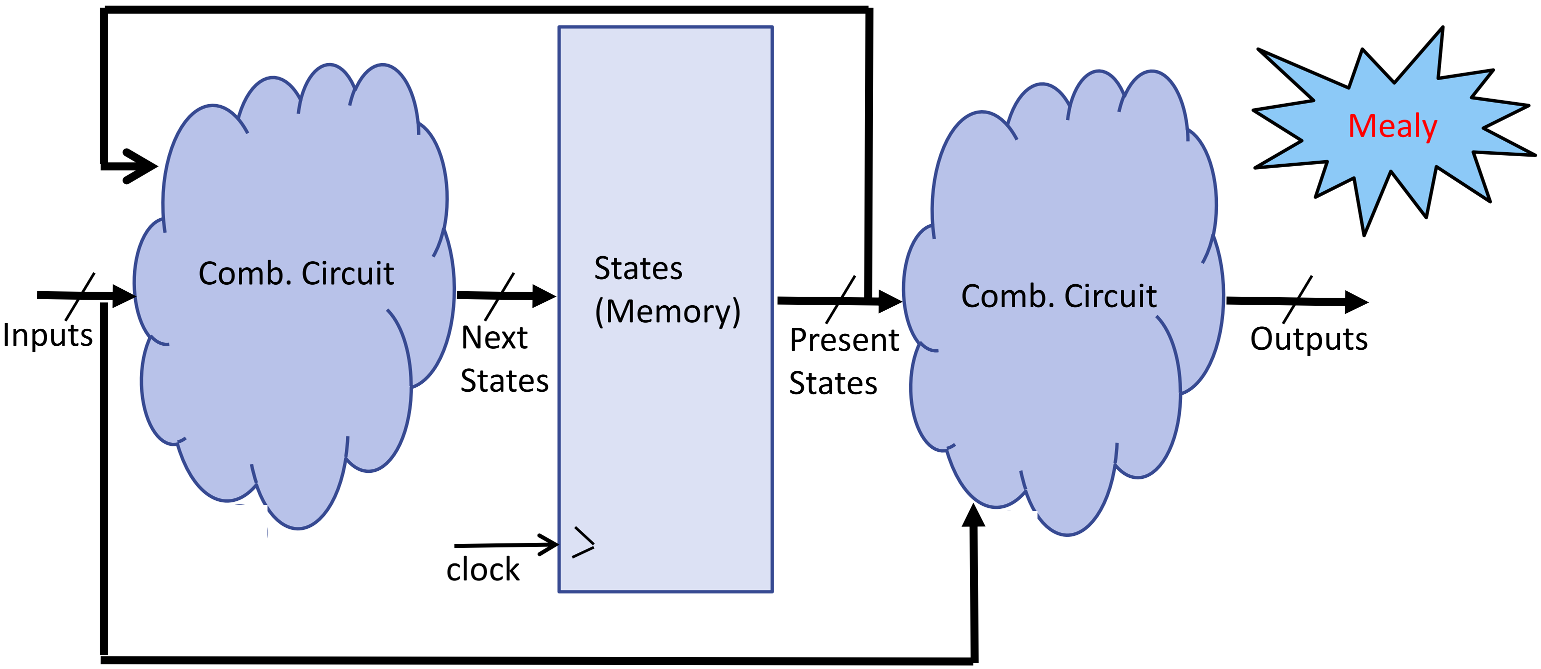
- How to implement FSMs

FSM

Moore and Mealy Machines



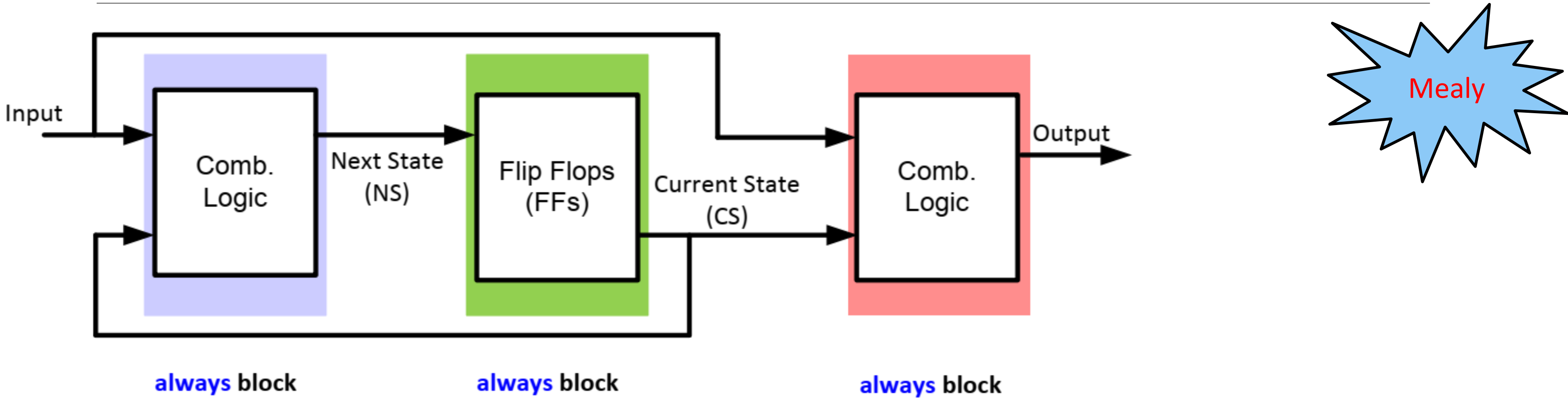
Moore and Mealy Machines



How to Describe FSMs in Verilog?

- Determine how to derive
  - Next State (NS)
  - Current State (CS)
  - Output

Verilog Statements and Verilog



Mealy

Input

Comb. Logic

Next State (NS)

Flip Flops (FFs)

Current State (CS)

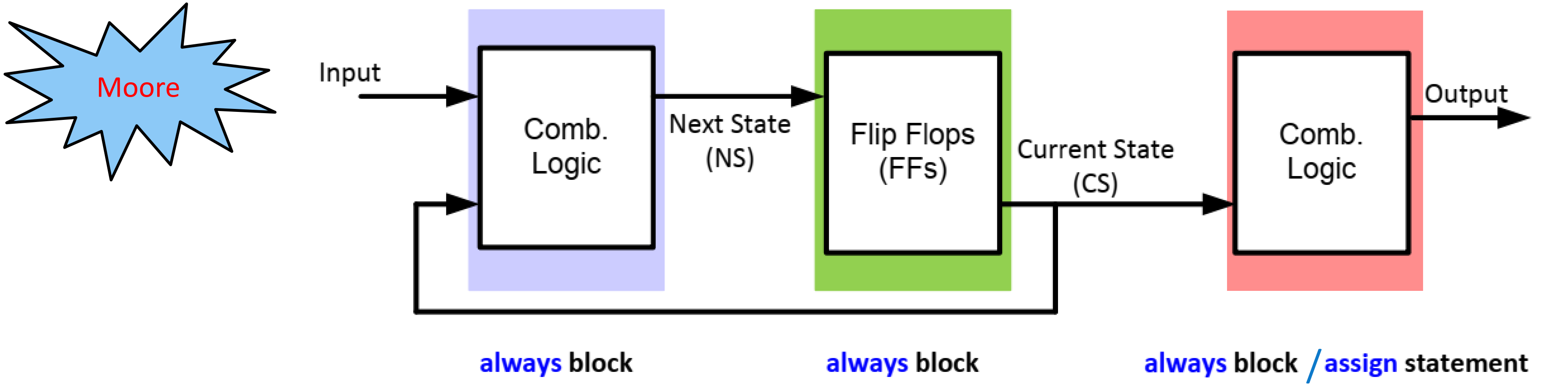
Comb. Logic

Output

always block

always block

always block



Moore

Input

Comb. Logic

Next State (NS)

Flip Flops (FFs)

Current State (CS)

Comb. Logic

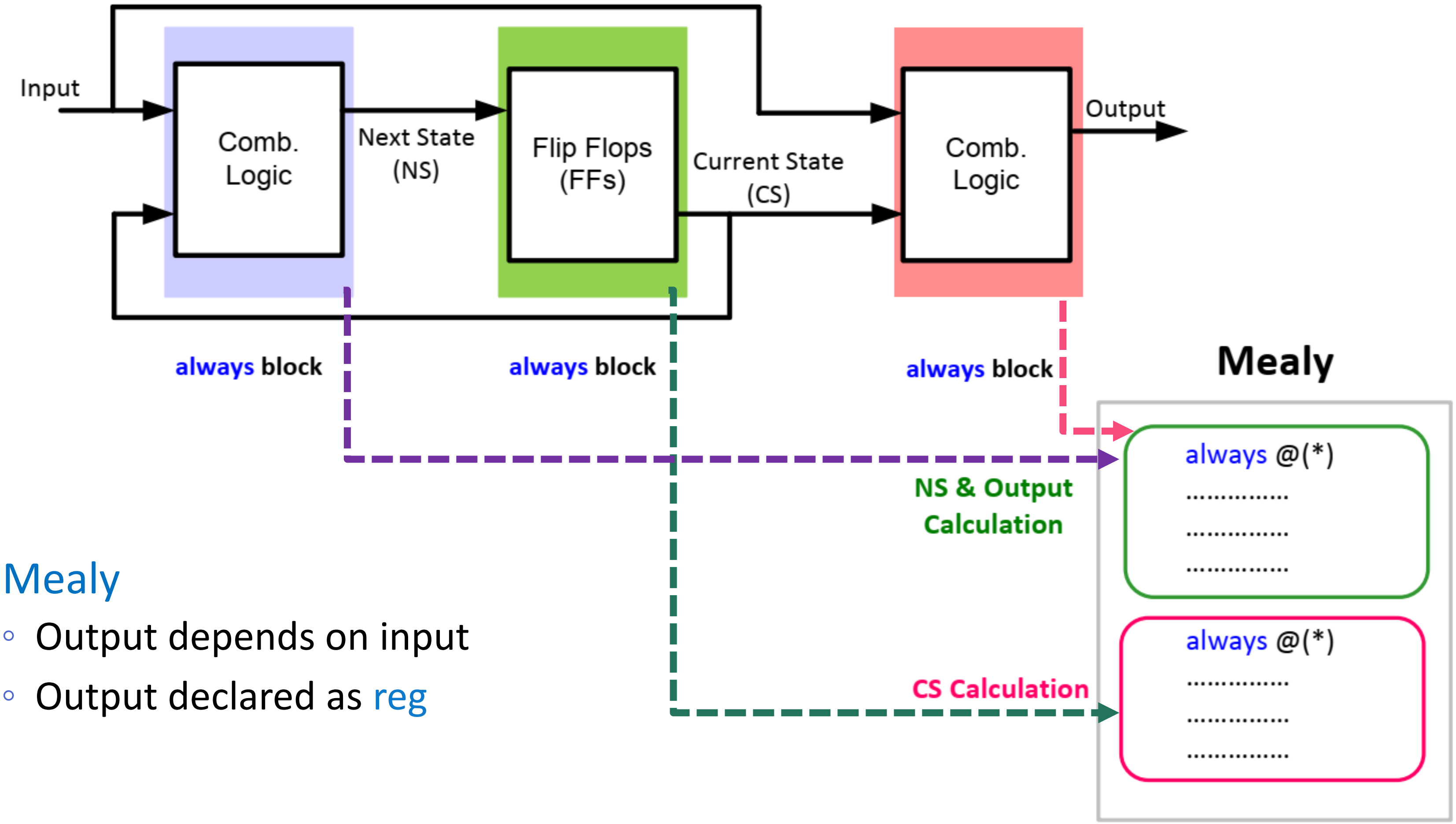
Output

always block

always block

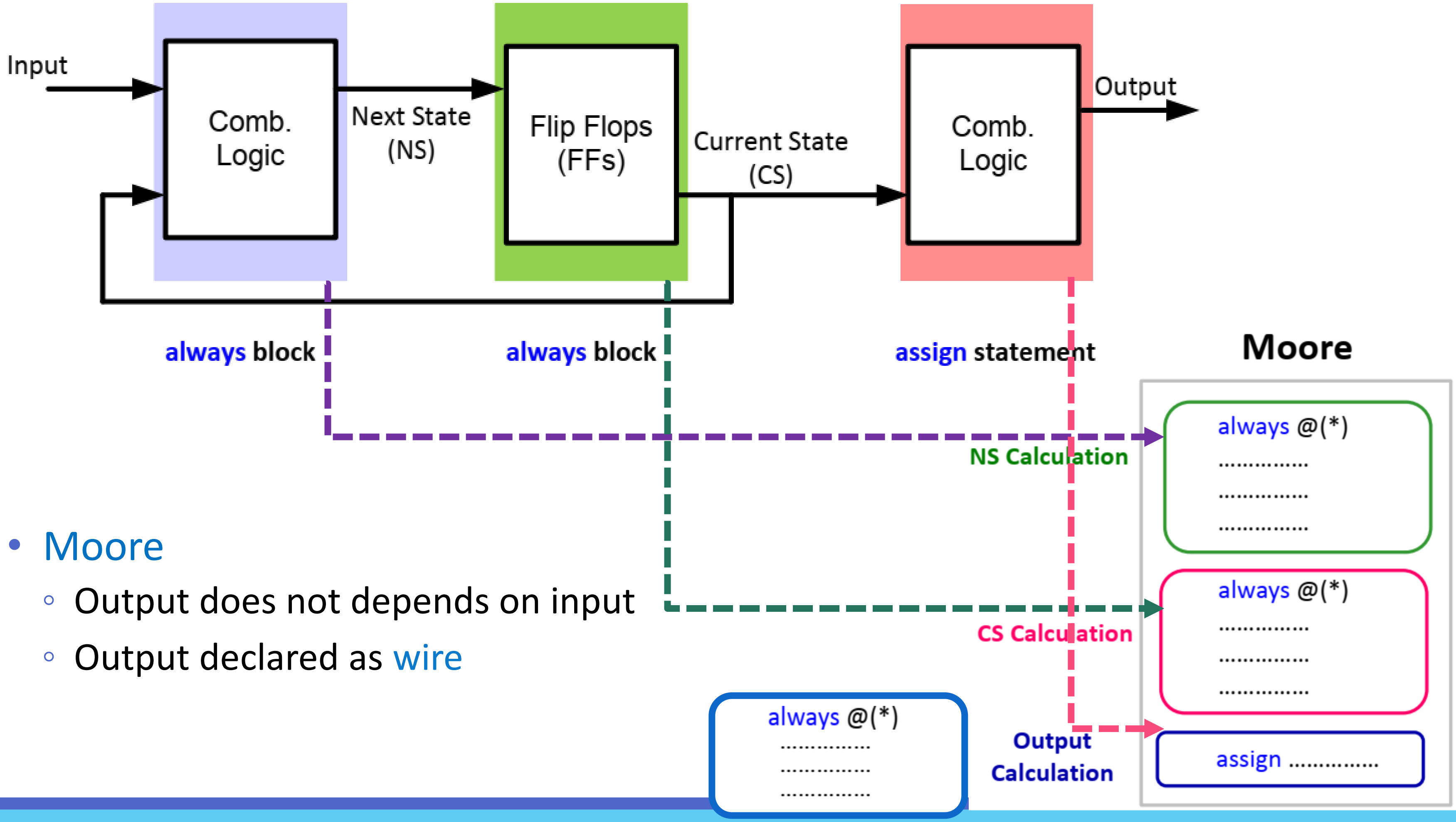
always block / assign statement

Mealy FSM Code Structure



- Mealy
  - Output depends on input
  - Output declared as reg

Moore FSM Code Structure



Input

Comb.
Logic

Next State
(NS)

Flip Flops
(FFs)

Current State
(CS)

Comb.
Logic

Output

always block

always block

assign statement

Moore

always @(*)
....
....
....

NS Calculation

always @(*)
....
....
....

CS Calculation

always @(*)
....
....
....

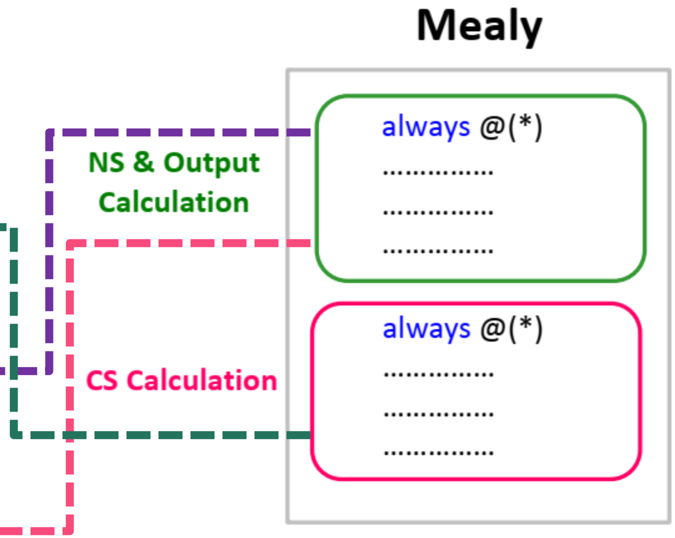
Output
Calculation

assign ....

- Moore
  - Output does not depends on input
  - Output declared as wire

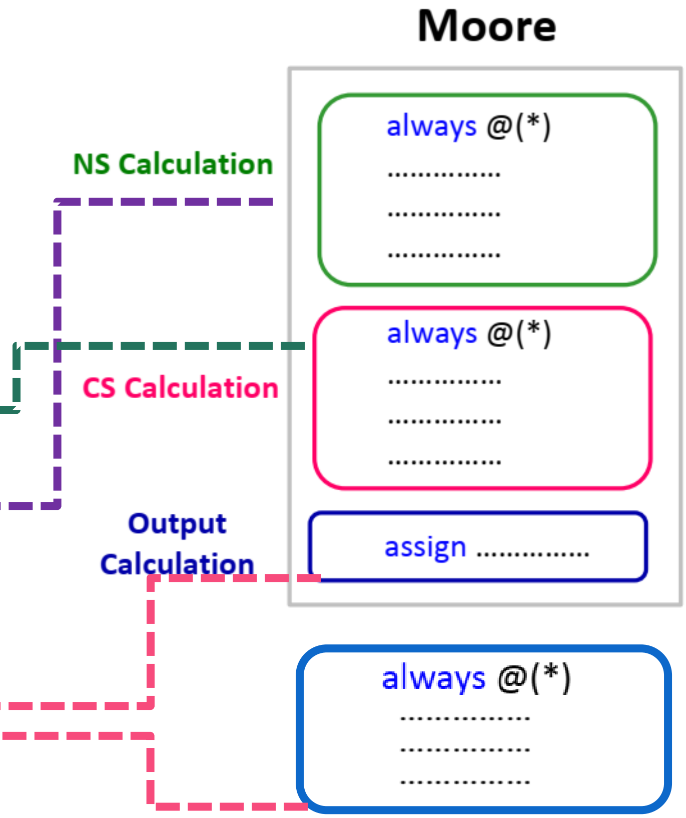
General Mealy FSM

```
module mealy(outs, inputs, clk);
            input        inputs, clk,;
            output reg  outs;

          reg  [log-num-states-1:0] state, next_state;

            always @(posedge/negedge clk)
              //  Change the state

           always @(state or inputs)
              // Evaluate next state


           always @(state or inputs)
               // Evaluate output
endmodule
```



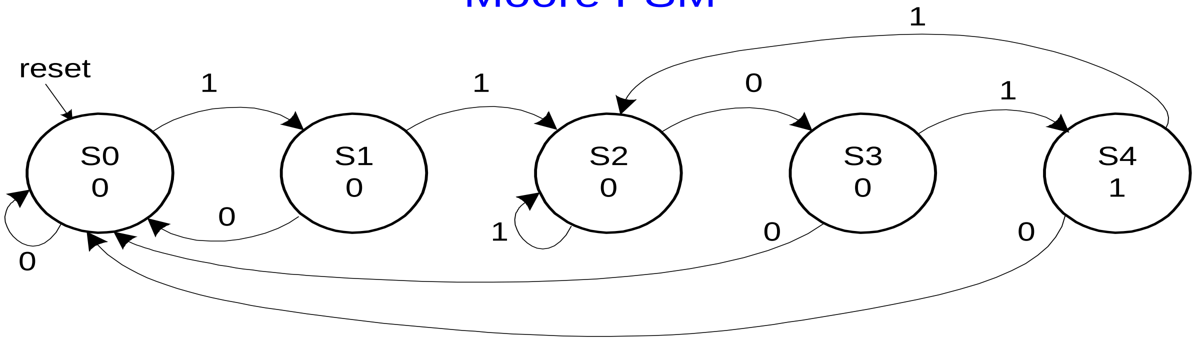
General Moore FSM

```
module moore(outs, inputs, clk);
            input        inputs, clk,;
            output reg  outs;

         reg  [log-num-states-1:0] state, next_state;

           always @(posedge/negedge clk)
             //  Change the state

           always @(state or inputs)
             // Evaluate next state

           always @(state)
              // Evaluate output
endmodule
```



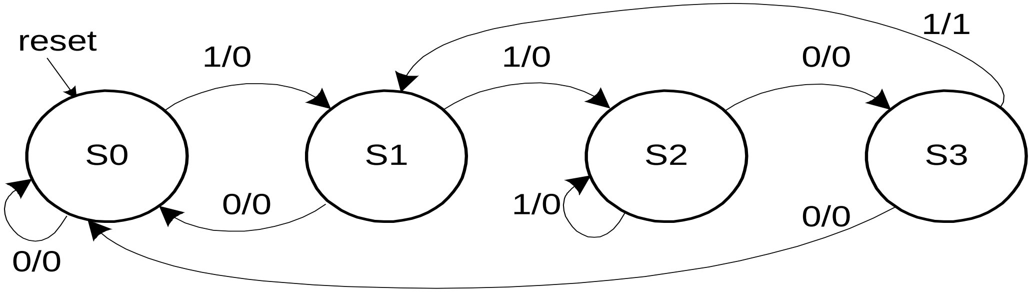
Finite State Machine (FSM)

Moore FSM



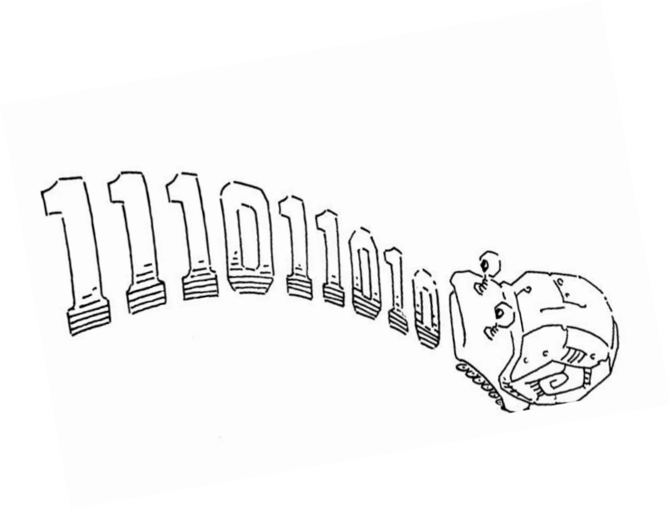
What are the tradeoffs?

Mealy FSM



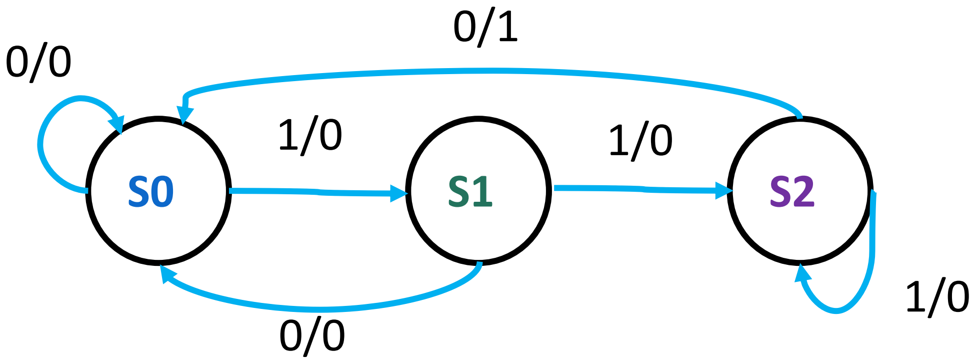
FSM Sample 1

- Recognize a specific bit pattern (*110*) in a bitstream



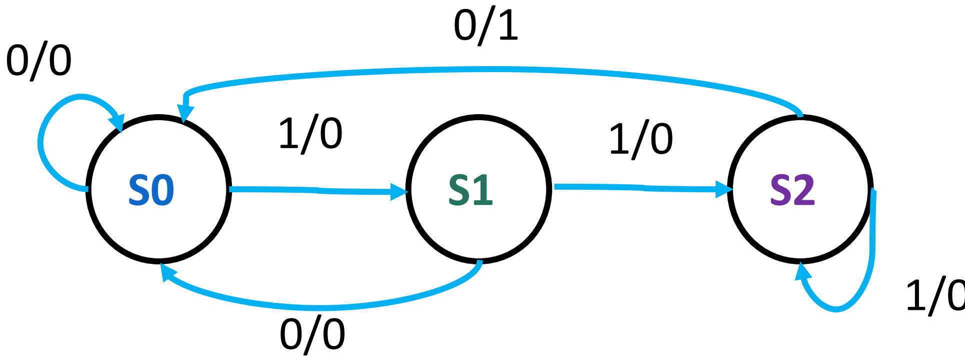
110 Detector

- State **S0**
  - We have not recognized any useful pattern
- State **S1**
  - We have recognized the pattern ’1’
- State **S2**:
  - We have recognized the pattern ’11’
  - **Output**: recognizing an input bit ’0’ in state S2



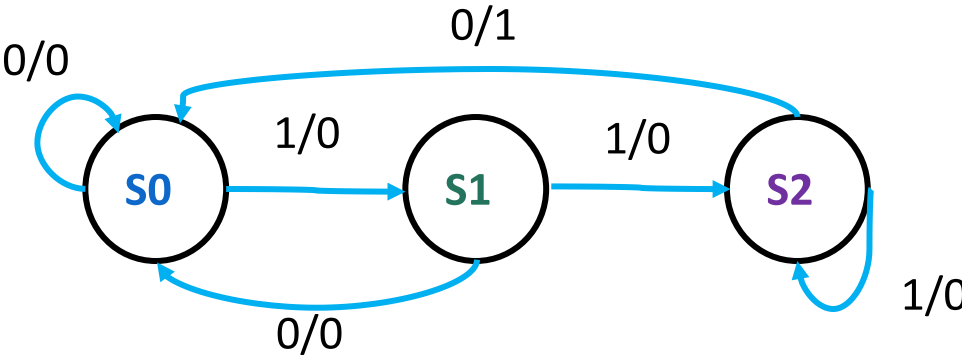
110 Detector : Module

```
module mealy (out, i, clk, reset);

endmodule
```



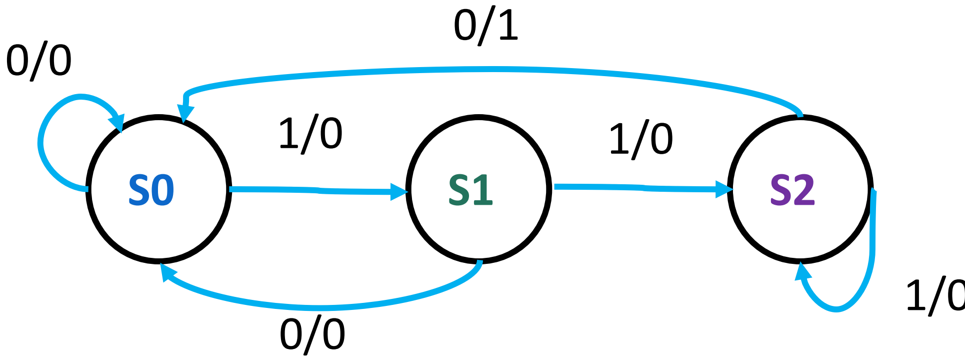
110 Detector : Ports

```
module mealy (out, i, clk, reset);
          input    i, clk, reset;
          output  out;
           reg      out;





endmodule
```



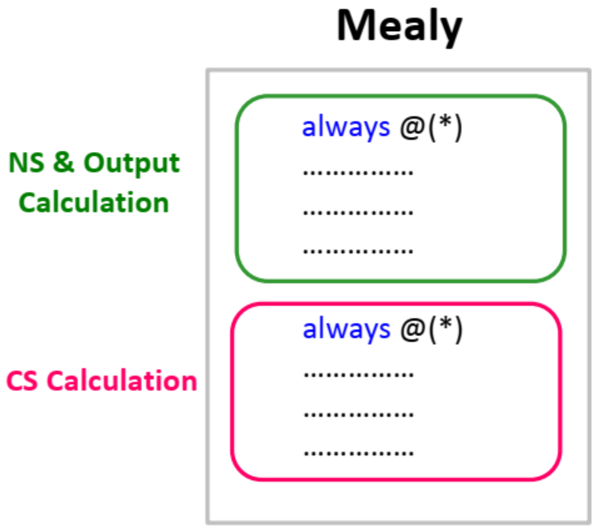
110 Detector : Signals

```
module mealy (out, i, clk, reset);
        input    i, clk, reset;
        output  out;
         reg      out;
         reg  [1:0] state, next_state;



endmodule
```



110 Detector :
Next State & Output

```
module mealy (out, i, clk, reset);
        input    i, clk, reset;
        output  out;
         reg      out;
         reg  [1:0] state, next_state;




        always @(state or i)
          case (state)
             2'b00 : next_state = i ? 1 : 0;
             2'b01 : next_state = i ? 2 : 0;
             2'b10 : next_state = i ? 2 : 0;
             default: next_state = 0;
          endcase

         always @(state or  i)
           if (state == 2'b10 &&  i==0) out = 1;
          else                          out = 0;

endmodule
```



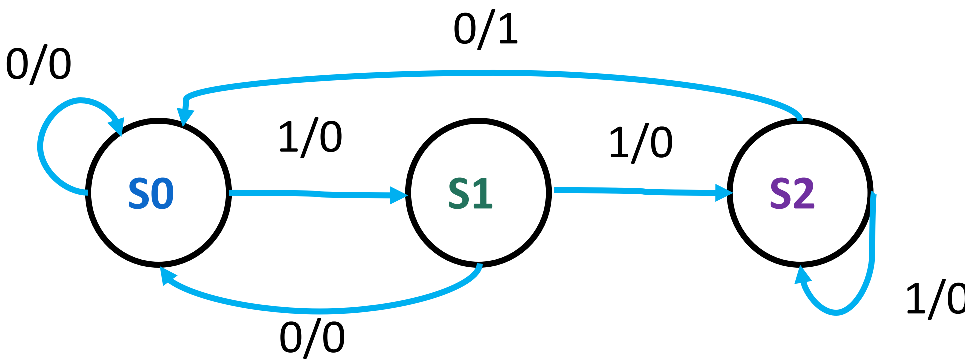
Mealy

NS & Output Calculation

always @(*)
……………
……………
……………

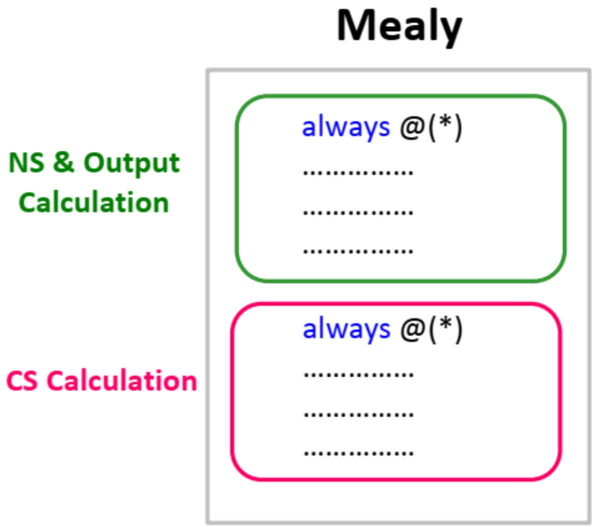
CS Calculation

always @(*)
……………
……………
……………



110 Detector : Current State

```
module mealy (out, i, clk, reset);
        input    i, clk, reset;
        output  out;
         reg      out;
         reg  [1:0] state, next_state;

         always @(posedge clk)
           if( reset) state <= 0;
           else state <= next_state;

         always @(state or i)
           case (state)
              2'b00 : next_state = i ? 1 : 0;
              2'b01 : next_state = i ? 2 : 0;
              2'b10 : next_state = i ? 2 : 0;
              default: next_state = 0;
           endcase

          always @(state or  i)
            if (state == 2’b10 &&  i==0) out = 1;
           else                                        out = 0;

endmodule
```

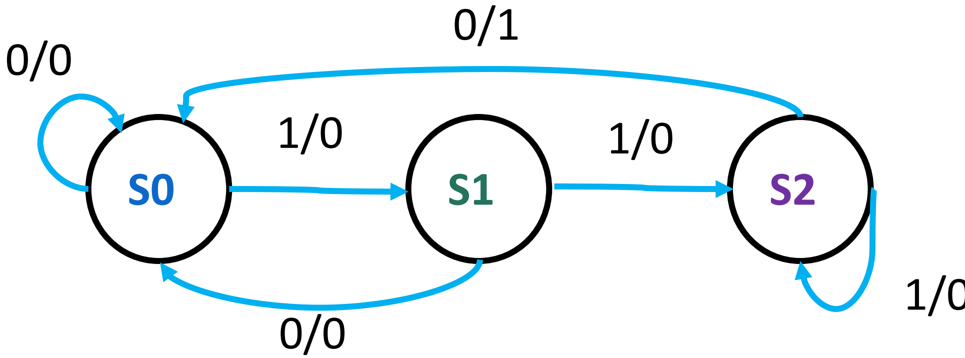

Mealy

NS & Output Calculation

always @(*)
....
....
....

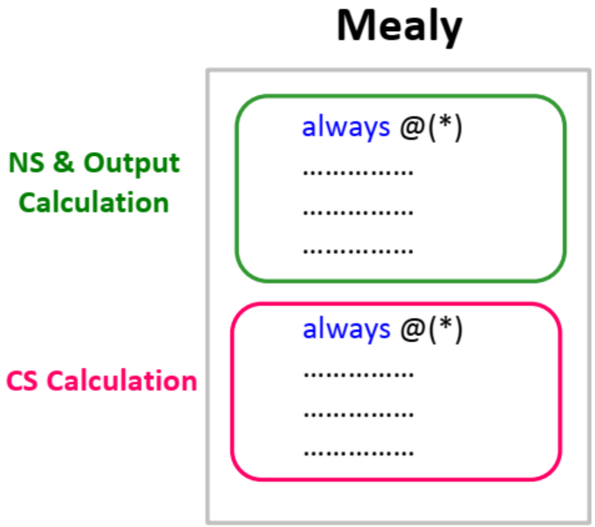
CS Calculation

always @(*)
....
....
....



110 Detector : Merged Next State + Output

```
module mealy(out, i, clk, reset);
        input    i, clk, reset;
        output  out;
         reg      out;
         reg  [1:0] state, next_state;

         always @(posedge clk)
           if( reset) state <= 0;
           else state <= next_state;

         always @(state or i)
           case (state)
              2'b00 : next_state <= i ? 1 : 0;
              2'b01 : next_state <= i ? 2 : 0;
              2'b10 :
                   begin
                     next_state <= i ? 2 : 0;
                     out <= i? 0:1;
                   end
              default: next_state = 0;
           endcase

endmodule
```



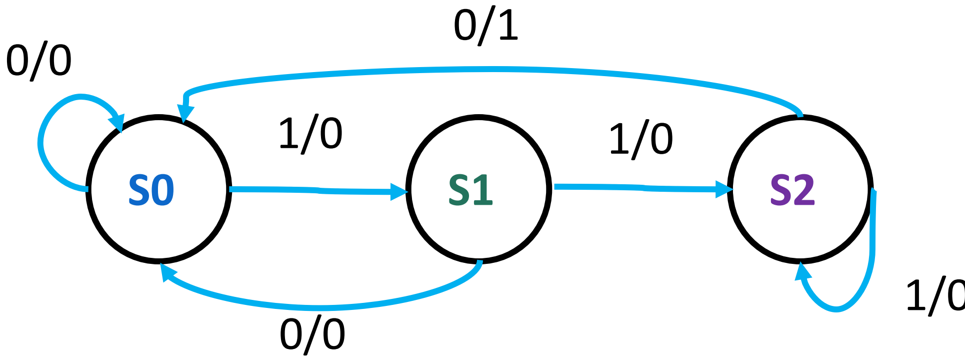
Mealy

NS & Output Calculation

always @(*)
....
....
....

CS Calculation

always @(*)
....
....
....

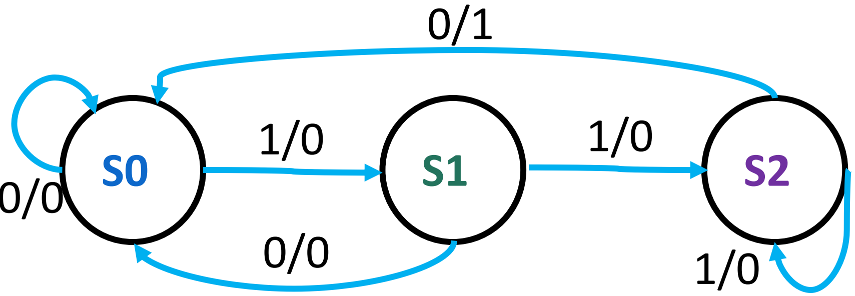


110 Detector: Modular

```
module mealy(out, i, clk, reset);
         input    i, clk, reset;
         output  out;
          reg       out;
          reg  [1:0] state, next_state;

          always @(posedge clk)
            if( reset) state <= 0;
            else state <= next_state;

           cmp_next_state (next_state, state, i);
            cmp_output (next_state, state, i);
endmodule
```

```
module cmp_next_state (output reg [1:0] next_state, input [1:0] state,
                                          input i);
       always @(state or  i)
           begin
              if (state == 0)       next_state = i ? 1 : 0;
             else if (state == 1) next_state = i ? 2 : 0;
             else if (state == 2) next_state = i ? 2 : 0;
             else next_state = 0;
           end
endmodule
```

```
module cmp_output (output reg out, input [1:0] state,
                                 input i);
       always @(state or  i)
           begin
              if (state == 2'b10 &&  i==0) out = 1;
             else                                   out = 0;
           end
endmodule
```

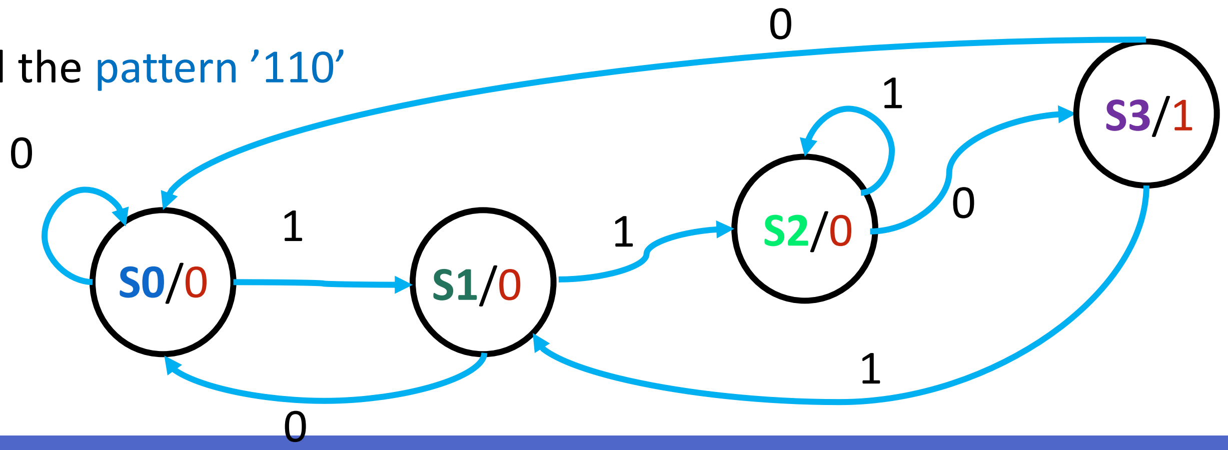


FSM Sample 2

- Recognize a specific bit pattern (*110*) in a bitstream using Moore FSM

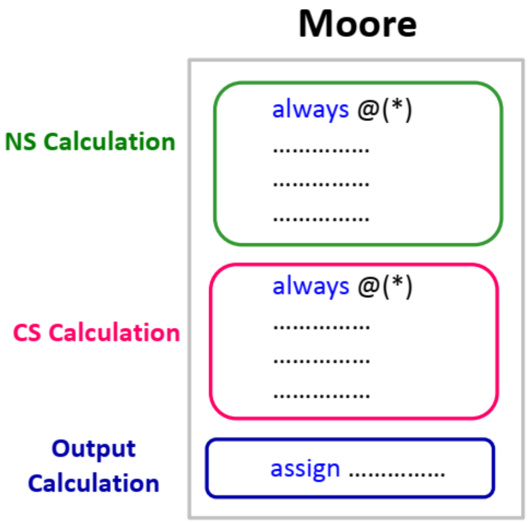
Moore FSM Sample : 110

- State **S0**
  - We have not recognized any useful pattern
- State **S1**
  - We have recognized the pattern ’1’
- State **S2**:
  - We have recognized the pattern ’11’
- State **S3**:
  - We have recognized the pattern ’110’
  - Output becomes 1



Moore 110 Detector : Module

```
module moore (out, i, clk, reset);











endmodule
```



Moore

NS Calculation

```
always @(*)
...............
...............
...............
```

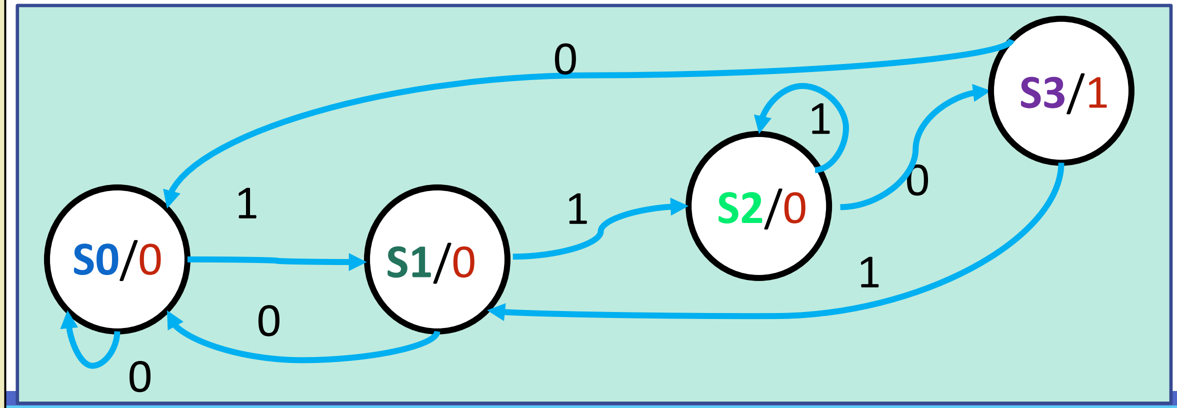
CS Calculation

```
always @(*)
...............
...............
...............
```

Output Calculation

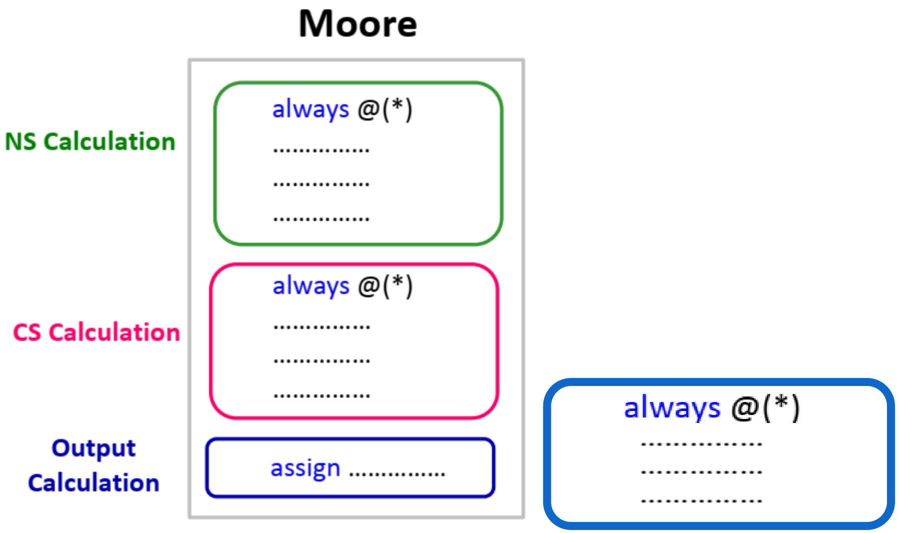
```
assign ...............
```

```
always @(*)
...............
...............
...............
```



Moore 110 Detector : Ports

```
module moore (out, i, clk, reset);
        input    i, clk, reset;
        output  out;
         reg       out;
```



Moore

NS Calculation

```
always @(*)
……………
……………
……………
```

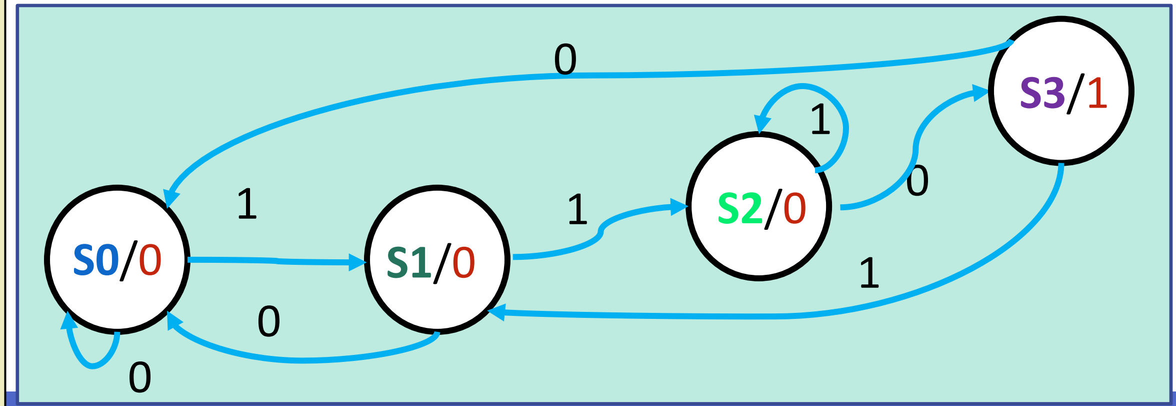
CS Calculation

```
always @(*)
……………
……………
……………
```

Output Calculation

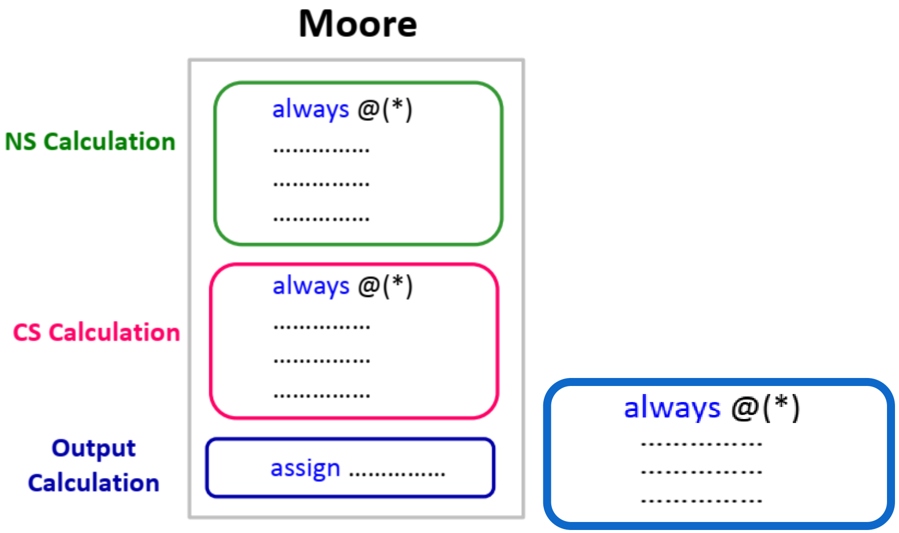
```
assign ……………
```

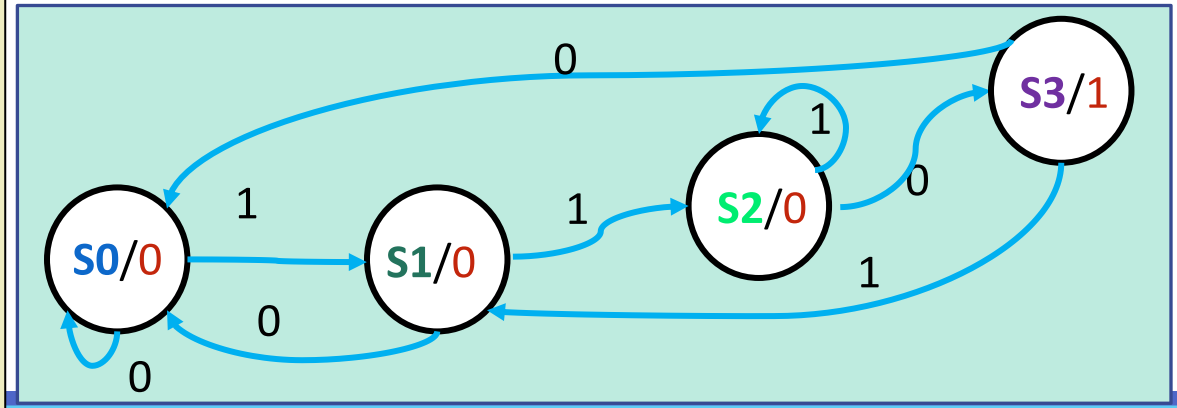
```
always @(*)
……………
……………
……………
```



Moore 110 Detector : Signals

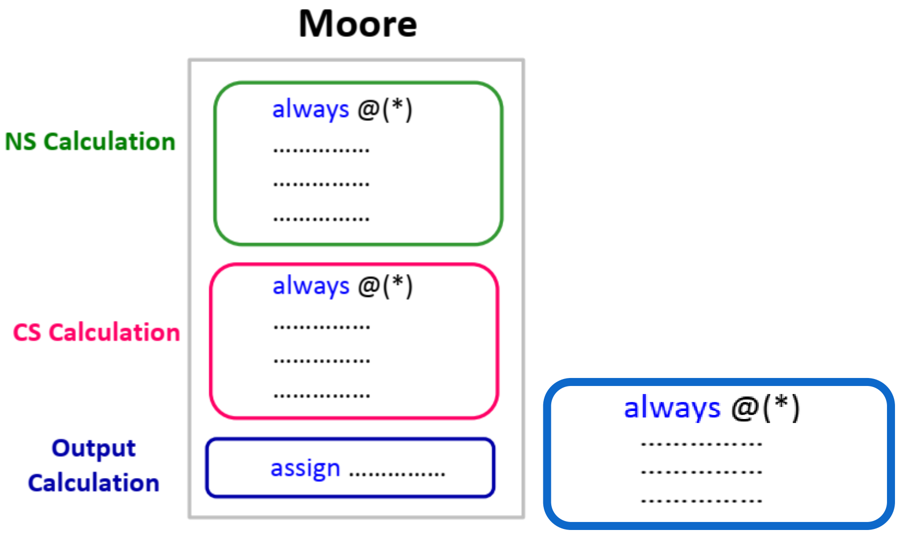
```
module moore (out, i, clk, reset);
        input    i, clk, reset;
        output  out;
         reg       out;
         reg  [1:0] state, next_state;




endmodule
```





Moore 110 Detector : Next State & Output

```
module moore (out, i, clk, reset);
        input    i, clk, reset;
        output  out;
         reg      out;
         reg  [1:0] state, next_state;




         always @(state or i)
           case (state)
              2'b00 : next_state = i ? 1 : 0;
              2'b01 : next_state = i ? 2 : 0;
              2'b10 : next_state = i ? 2 : 3;
              2’b11 : next_state = i ? 1 : 0;
              default: next_state = 0;
           endcase

          always @(state)
           if (state == 2’b11 ) out = 1;
           else                  out = 0;

endmodule
```



Moore

NS Calculation

```
always @(*)
...............
...............
...............
```

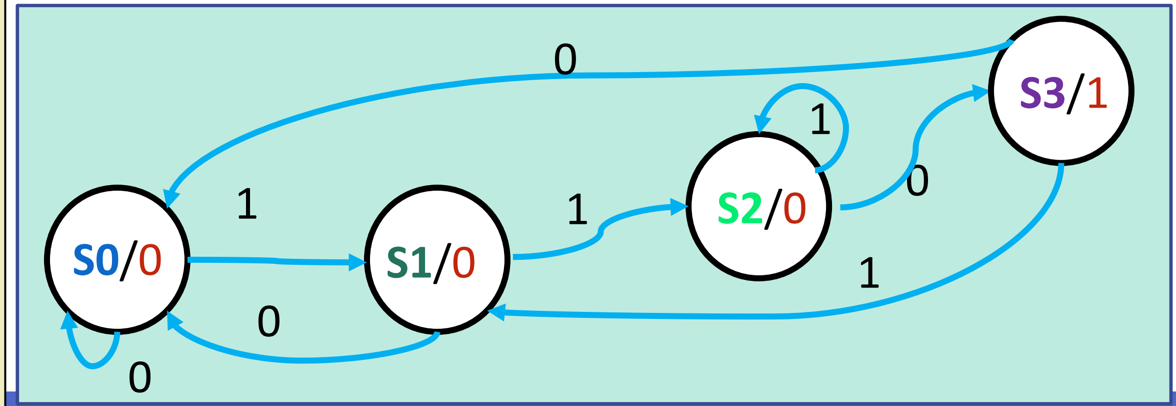
CS Calculation

```
always @(*)
...............
...............
...............
```

Output Calculation

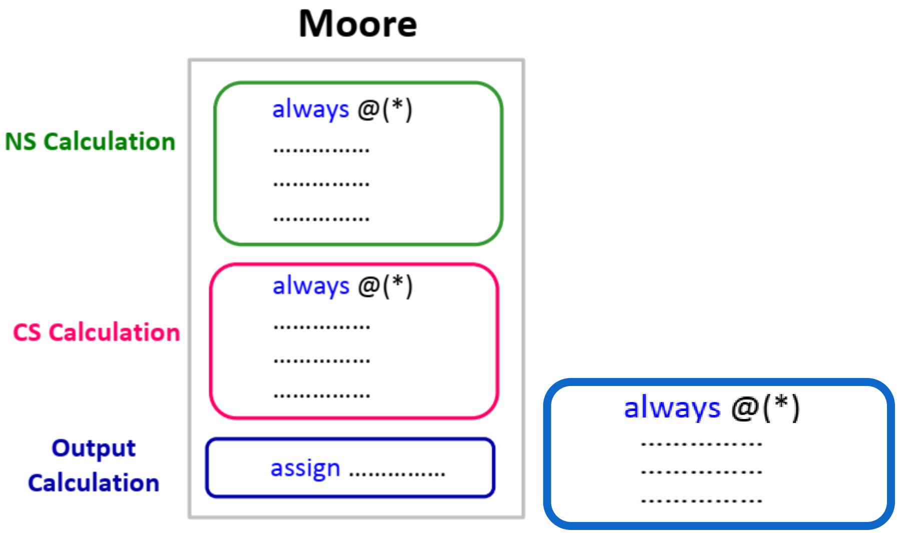
```
assign ...............
```

```
always @(*)
...............
...............
...............
```



Moore 110 Detector : Current State

```
module moore(out, i, clk, reset);
        input    i, clk, reset;
        output  out;
         reg      out;
         reg  [1:0] state, next_state;

         always @(posedge clk)
           if( reset) state <= 0;
           else state <= next_state;

         always @(state or i)
           case (state)
              2'b00 : next_state = i ? 1 : 0;
              2'b01 : next_state = i ? 2 : 0;
              2'b10 : next_state = i ? 2 : 3;
              2’b11 : next_state = i ? 1 : 0;
              default: next_state = 0;
           endcase

          always @(state )
           if (state == 2’b11 ) out = 1;
           else                          out = 0;

endmodule
```



Moore

NS Calculation

```
always @(*)
...............
...............
...............
```

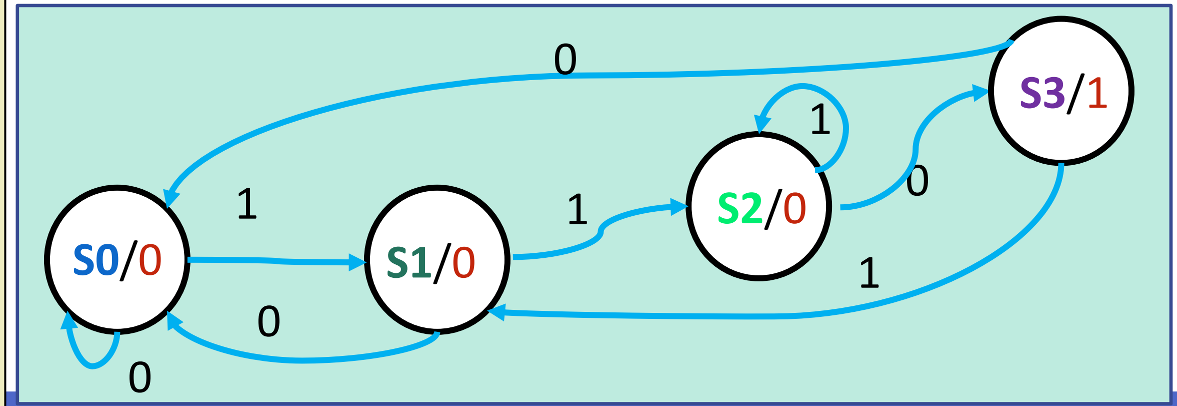
CS Calculation

```
always @(*)
...............
...............
...............
```

Output Calculation

```
assign ...............
```

```
always @(*)
...............
...............
...............
```

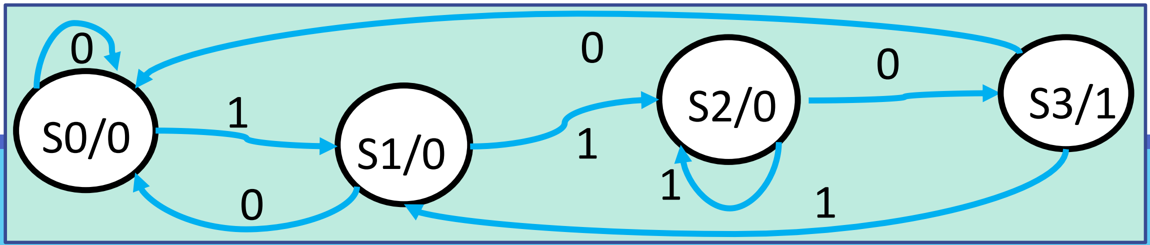


Moore 110 Detector : Current State

```
module moore(out, i, clk, reset);
        input    i, clk, reset;
        output  out;
         reg      out;
         reg  [1:0] state, next_state;

         always @(posedge clk)
           if( reset) state <= 0;
           else state <= next_state;

          cmp_next_state (next_state, state, i)

          cmp_output (next_state, state);

endmodule
```

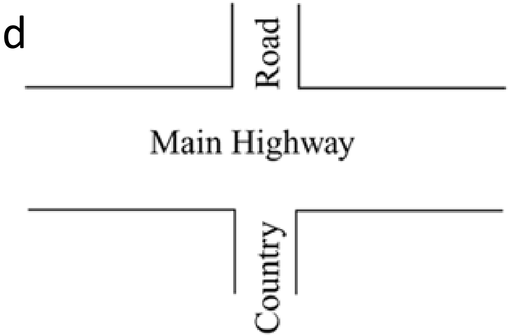
```
module cmp_next_state(output reg [1:0] next_state, input [1:0] state,
                                          input i);
      always @(state or  i)
          begin
             if (state == 0)       next_state = i ? 1 : 0;
             else if (state == 1) next_state = i ? 2: 0;
             else if (state == 2) next_state = i ? 2 : 3;
             else if (state == 3) next_state = i ? 1 : 0;
             else next_state = 0;
           end
endmodule
```

```
module cmp_output(output reg out,
                                  input  [1:0] state);
         always @(state)
            begin
               if (state == 2'b11) out = 1;
                else                    out = 0;
             end
endmodule
```

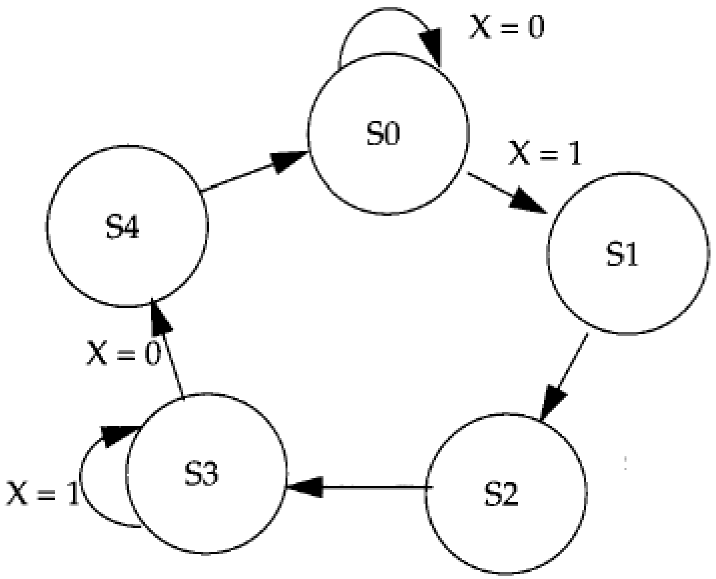


Traffic Signal Controller

- Traffic Signal for main highway gets highest priority
  - Cars are continuously present on the main highway
  - Main highway signal remains green by default
- Traffic signal for the country road
  - Must turn green only long enough to let the cars on the country road go
  - As soon as there are no cars on the country road
    - Country road traffic signal turns yellow and then red
    - Traffic signal on the main highway turns green again
  - There is a sensor to detect cars waiting on the country road
    - Sends a signal X as input to the controller.
    - X = 1 if there are cars on the country road; otherwise, X= 0.



Traffic Signal Controller (cont'd)



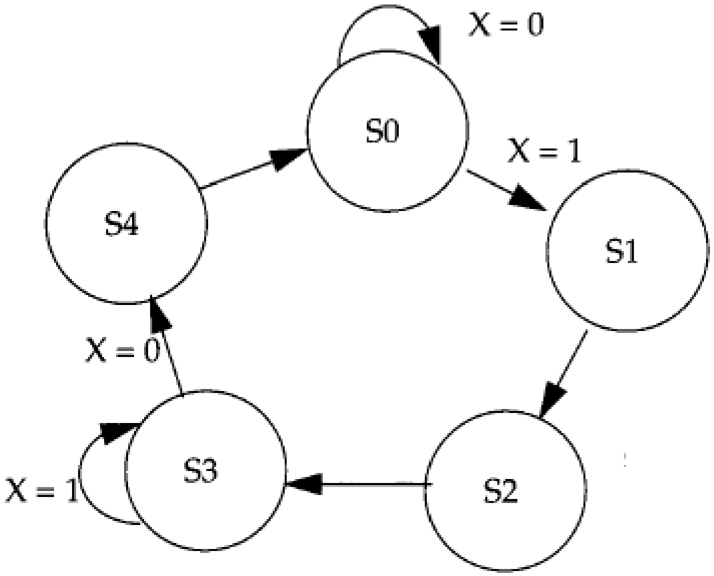
State	Signals
S0	*Hwy = G Cntry = R*
S1	*Hwy = Y Cntry = R*
S2	*Hwy = R Cntry = R*
S3	*Hwy = R Cntry = G*
S4	*Hwy = R Cntry = Y*

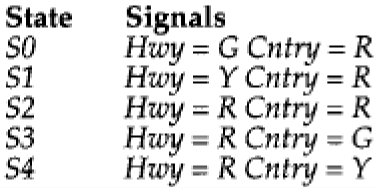
Traffic Signal Controller (cont’d)

```
`define TRUE  1'b1
`define FALSE 1'b0

//Delays
`define Y2RDELAY  3 //Yellow to red delay
`define R2GDELAY  2 //Red to green delay

module sig_control
    (hwy, cntry, X, clock, clear);

//I/O ports
output [1:0] hwy, cntry;
      //2-bit output for 3 states of signal
      //GREEN, YELLOW, RED;
reg [1:0] hwy, cntry;
      //declared output signals are registers

input X;
      //if TRUE, indicates that there is car on
      //the country road, otherwise FALSE
input clock, clear;

parameter RED = 2'd0,
          YELLOW = 2'd1,
          GREEN = 2'd2;

//State definition     HWY          CNTRY
parameter S0 = 3'd0, //GREEN         RED
          S1 = 3'd1, //YELLOW        RED
          S2 = 3'd2, //RED           RED
          S3 = 3'd3, //RED           GREEN
          S4 = 3'd4; //RED           YELLOW
```

Traffic Signal Controller (cont’d)

```
//Internal state variables
reg [2:0] state;
reg [2:0] next_state;


//state changes only at positive edge of clock
always @(posedge clock)
  if (clear)
      state <= S0; //Controller starts in S0 state
  else
      state <= next_state; //State change


//Compute values of main signal and country signal
always @(state)
begin
  hwy = GREEN; //Default Light Assignment for Highway light
  cntry = RED; //Default Light Assignment for Country light
  case(state)
     S0: ; // No change, use default
     S1: hwy = YELLOW;
     S2: hwy = RED;
     S3:  begin
            hwy = RED;
            cntry = GREEN;
          end
     S4:  begin
            hwy = RED;
            cntry = `YELLOW;
          end
  endcase
end
```

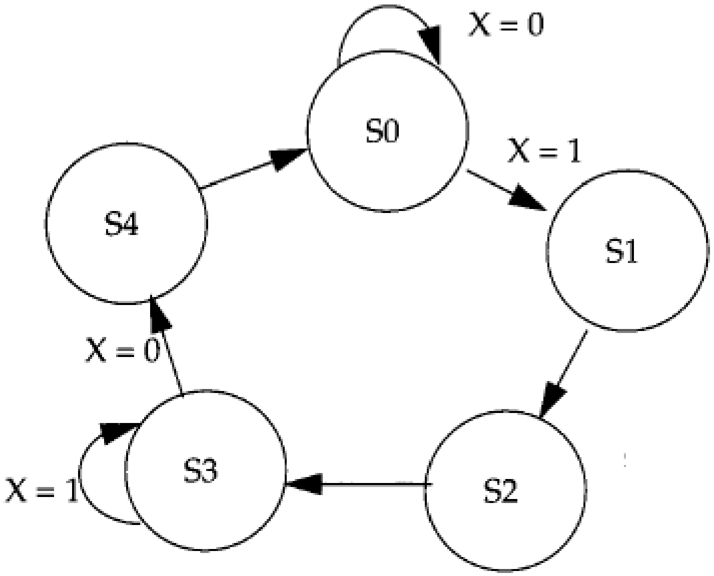




State	Signals
S0	*Hwy* = *G Cntry* = *R*
S1	*Hwy* = *Y Cntry* = *R*
S2	*Hwy* = *R Cntry* = *R*
S3	*Hwy* = *R Cntry* = *G*
S4	*Hwy* = *R Cntry* = *Y*

Traffic Signal Controller (cont'd)

```
//State machine using case statements
always @(state or  X)
begin
    case (state)
       S0: if(X)
            next_state = S1;
          else
            next_state = S0;
       S1: begin //delay some positive edges of clock
            repeat(`Y2RDELAY) @(posedge clock) ;
            next_state = S2;
          end
       S2: begin //delay some positive edges of clock
            repeat(`R2GDELAY) @(posedge clock);
            next_state = S3;
          end
       S3: if(X)
            next_state = S3;
          else
            next_state = S4;
       S4: begin //delay some positive edges of clock
            repeat(`Y2RDELAY) @(posedge clock) ;
            next_state = S0;
          end
      default: next_state = S0;
    endcase
end

endmodule
```



State	Signals
S0	*Hwy = G Cntry = R*
S1	*Hwy = Y Cntry = R*
S2	*Hwy = R Cntry = R*
S3	*Hwy = R Cntry = G*
S4	*Hwy = R Cntry = Y*

Traffic Signal Controller: Test

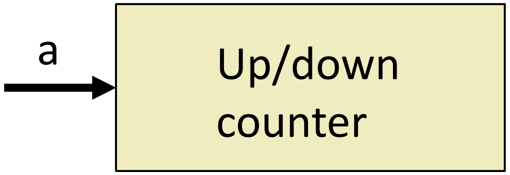
```
//Stimulus Module
module stimulus;

wire [1:0] MAIN_SIG, CNTRY_SIG;
reg CAR_ON_CNTRY_RD;
      //if TRUE, indicates that there is car on
      //the country road
reg CLOCK, CLEAR;

//Instantiate signal controller
sig_control SC(MAIN_SIG, CNTRY_SIG, CAR_ON_CNTRY_RD, CLOCK, CLEAR);

//Set up monitor
initial
  $monitor($time, " Main Sig = %b Country Sig = %b Car_on_cntry = %b",
                       MAIN_SIG, CNTRY_SIG, CAR_ON_CNTRY_RD);

//Set up clock
initial
begin
    CLOCK = `FALSE;
    forever #5 CLOCK = ~CLOCK;
end

//control clear signal
initial
```

Traffic Signal Controller: Test (cont’d)

```
//control clear signal
initial
begin
    CLEAR = `TRUE;
    repeat (5) @(negedge CLOCK);
    CLEAR = `FALSE;
end

//apply stimulus
initial
begin
    CAR_ON_CNTRY_RD = `FALSE;

    repeat(20)@(negedge CLOCK); CAR_ON_CNTRY_RD = `TRUE;
    repeat(10)@(negedge CLOCK); CAR_ON_CNTRY_RD = `FALSE;

    repeat(20)@(negedge CLOCK); CAR_ON_CNTRY_RD = `TRUE;
    repeat(10)@(negedge CLOCK); CAR_ON_CNTRY_RD = `FALSE;

    repeat(20)@(negedge CLOCK); CAR_ON_CNTRY_RD = `TRUE;
    repeat(10)@(negedge CLOCK); CAR_ON_CNTRY_RD = `FALSE;

    repeat(10)@(negedge CLOCK); $stop;
end
endmodule
```

Sample Counter

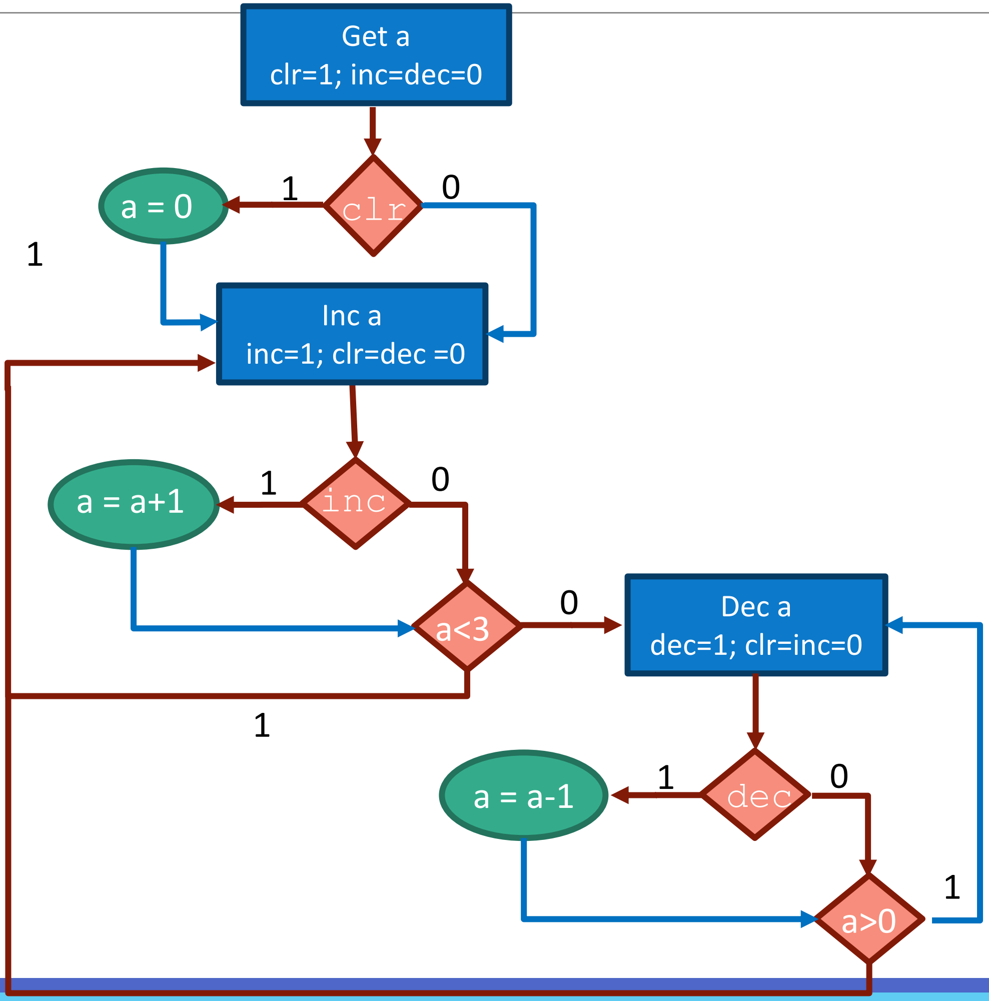
- Design an up/down counter



a

Up/down counter

Up/Down Counter ASM



Up/Down Counter: DataPath

```
module updown(c, inc, dec, clr)
  output reg [2:0] c;
  input  inc, dec, clr;
  reg    [2:0] a;

  always @(a)
      c = a;
  always@ (inc, dec, clr)
       begin
           if(inc) a = a + 1;
           if(dec) a = a - 1;
           if(clr) a = 0;
       end
endmoduel
```

Up/Down Counter: Control

```
module ctl_updown(input a, output inc, output dec, output clr)
  reg  [1:0] state, next_state;

  always @(posedge clk)
       if( reset) state <= 0;
       else state <= next_state;

  always@ (state)
   case (state)
     2’b00:
       begin clr =1; inc = dec =0; end
     2’b01:
       begin inc =1; clr = dec =0; end
     2’b10:
       begin dec =1: clr = inc =0; end
     default: begin clr =1; inc = dec =0; end

always@ (state)
   case (state)
     2’b00: next_state =  1;
     2’b01: next_state = (a<3)? 1: 2;
     2’b10: next_state = (a>0)? 2: 1;

endmoduel
```

Thank You