Bridge of Life Education

SOC Design
Verilog FSM Design

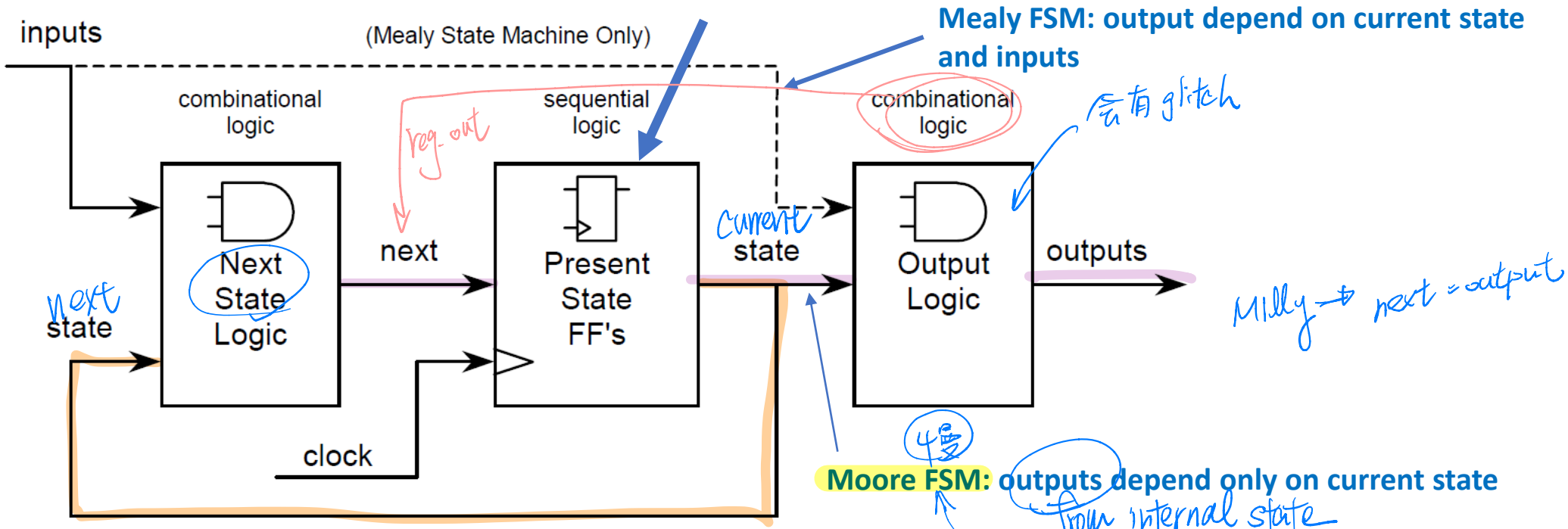
Jiin Lai

FSM Block Diagram

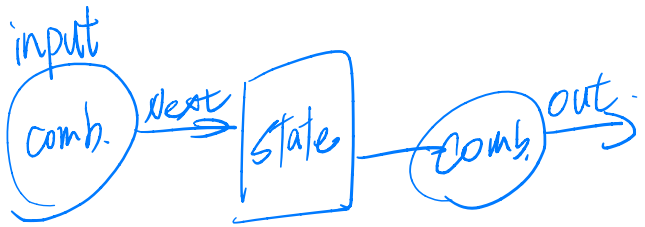
A sequential circuit which has

- External inputs
- Externally visible outputs
- Internal states

- **Store current state**
- **Load previously calculated next state**
- **# of states <= 2^(# of FFs)**



FSM Example



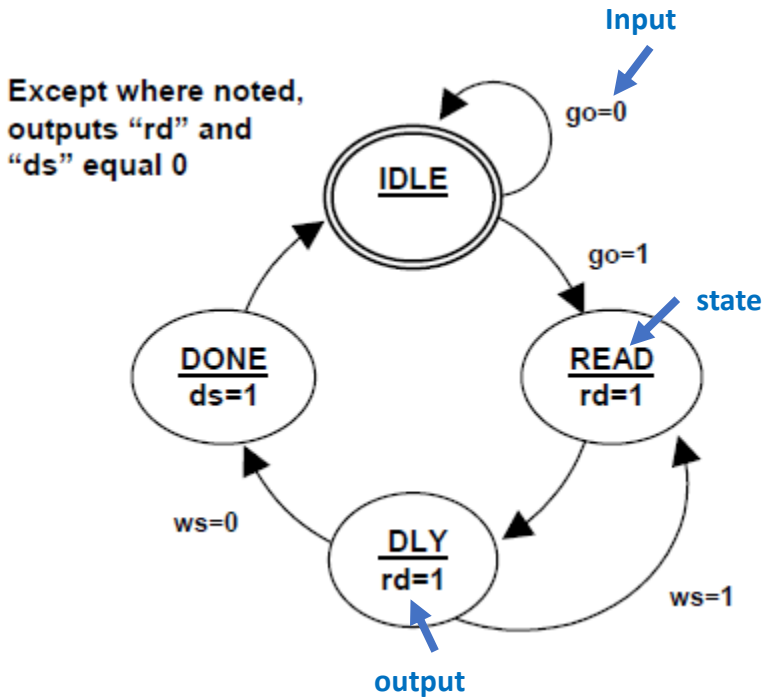

```
module fsm1a (ds, rd, go, ws, clk, rst_n);
  output ds, rd;
  input  go, ws;
  input  clk, rst_n;

  parameter [1:0] IDLE = 2'b00,
                  READ = 2'b01,
                  DLY  = 2'b10,
                  DONE = 2'b11;

  reg [1:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) state <= IDLE;
    else        state <= next;

  always @(state or go or ws) begin
    next = 2'bx;
    case (state)
      IDLE: if (go) next = READ;
            else    next = IDLE;

      READ:         next = DLY;

      DLY:  if (ws) next = READ;
            else    next = DONE;

      DONE:         next = IDLE;

    endcase
  end

  assign rd = (state==READ || state==DLY);
  assign ds = (state==DONE);
endmodule
```

State encoding

State register, sequential always block

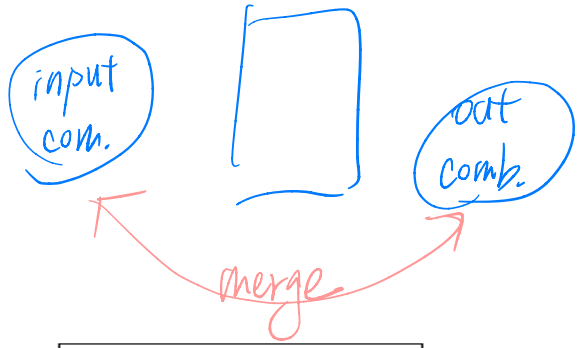
Next state, combinational always block

Continuous assignment outputs



Merge output & next state logic in one always block

```
module fsm1 (ds, rd, go, ws, clk, rst_n);
  output ds, rd;
  input  go, ws;
  input  clk, rst_n;
  reg    ds, rd;

  parameter [1:0] IDLE = 2'b00,
                  READ = 2'b01,
                  DLY  = 2'b10,
                  DONE = 2'b11;

  reg [1:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) state <= IDLE;
    else        state <= next;

  always @(state or go or ws) begin
    next = 2'bx;
    ds = 1'b0;
    rd = 1'b0;
    case (state)
      IDLE: if (go)    next = READ;
            else       next = IDLE;

      READ: begin      rd   = 1'b1;
                       next = DLY;
            end

      DLY:  begin      rd   = 1'b1;
              if (ws)  next = READ;
              else     next = DONE;
            end

      DONE: begin      ds   = 1'b1;
                       next = IDLE;
            end

    endcase
  end
endmodule
```



State register, sequential always block

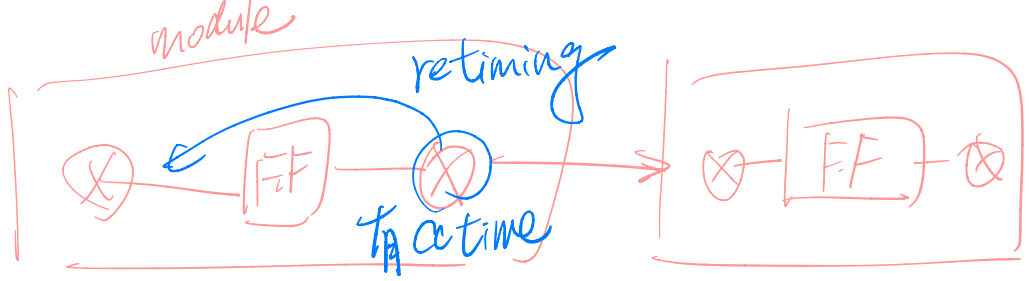
Next state & outputs, combinational always block

Problem with the two implementation

1. Combinational outputs can **glitch** between states.

```
assign rd = (state==READ || state==DLY);
assign ds = (state==DONE);
```

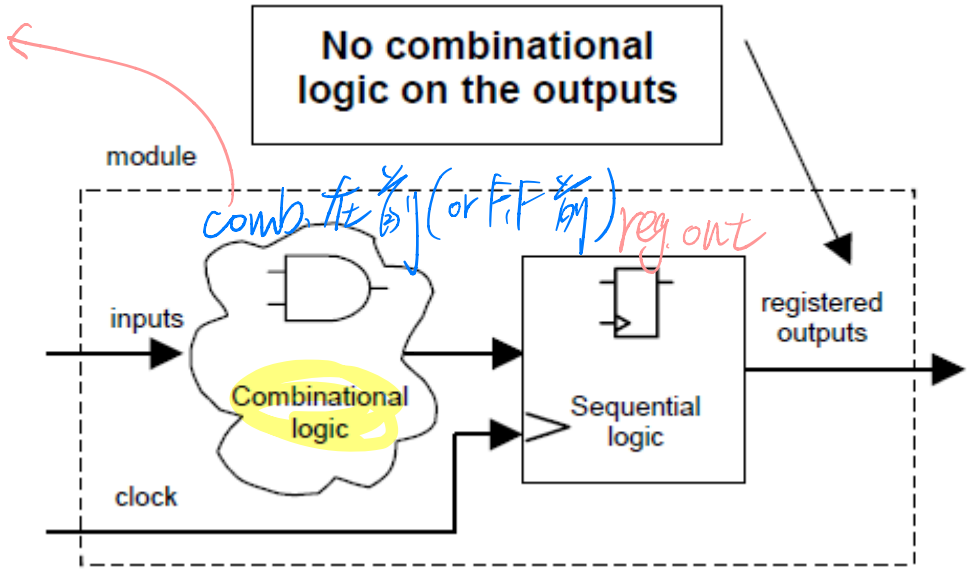
2. Combinational outputs **consume part of the overall clock cycle** that would have been available to the block of logic that is driven by the FSM outputs.

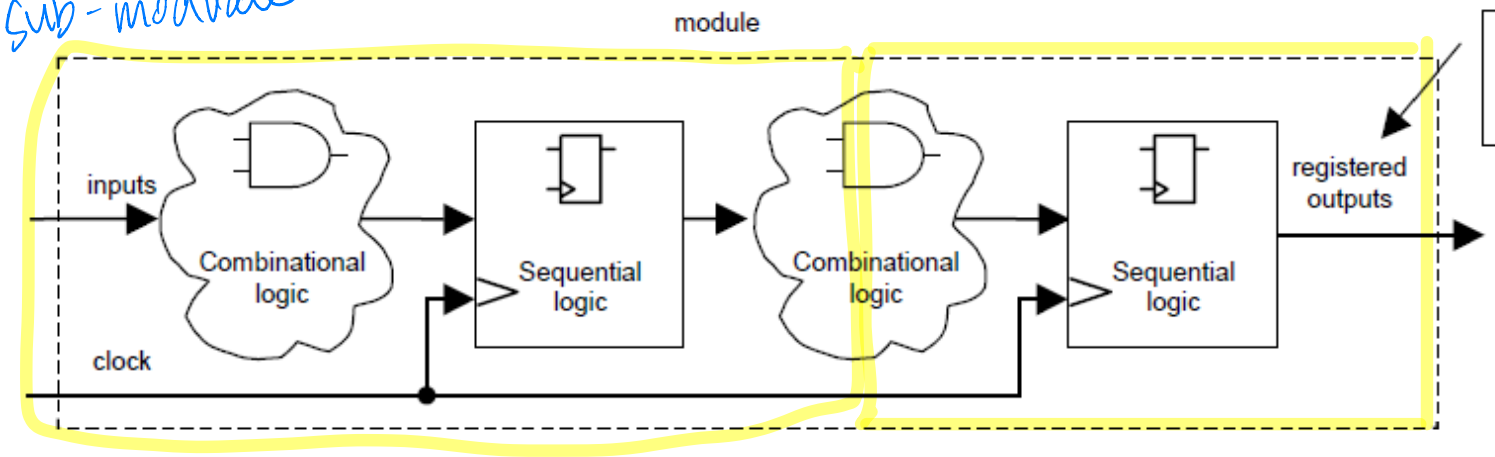


Module Partition for Synthesis – Registered out

Registered out

- All output are registered
- All combinational logic is on the input-side or between registered stages
- Simplify the task of constraining a design for synthesis
- Better meet timing requirement

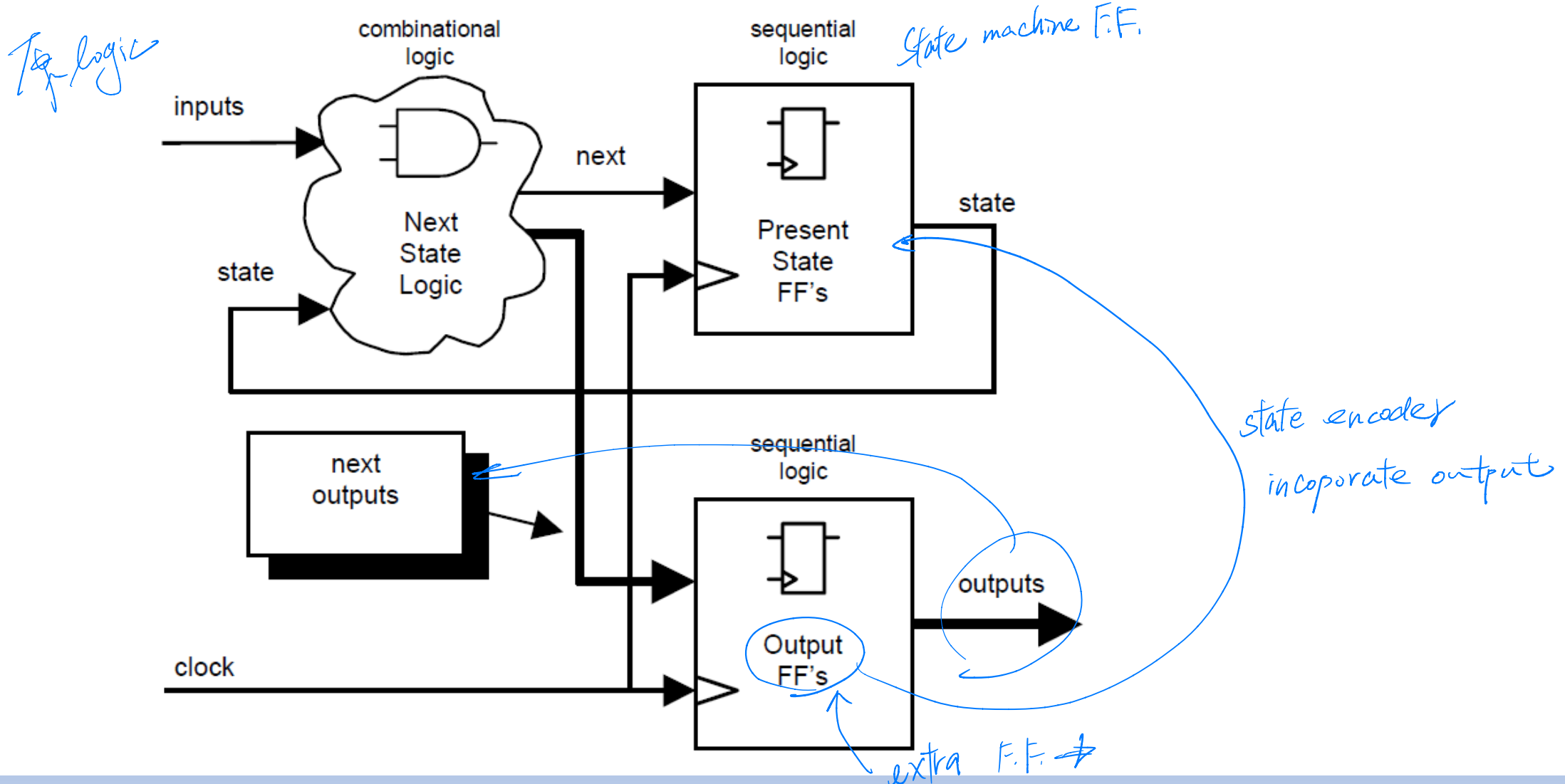




No combinational logic on the outputs

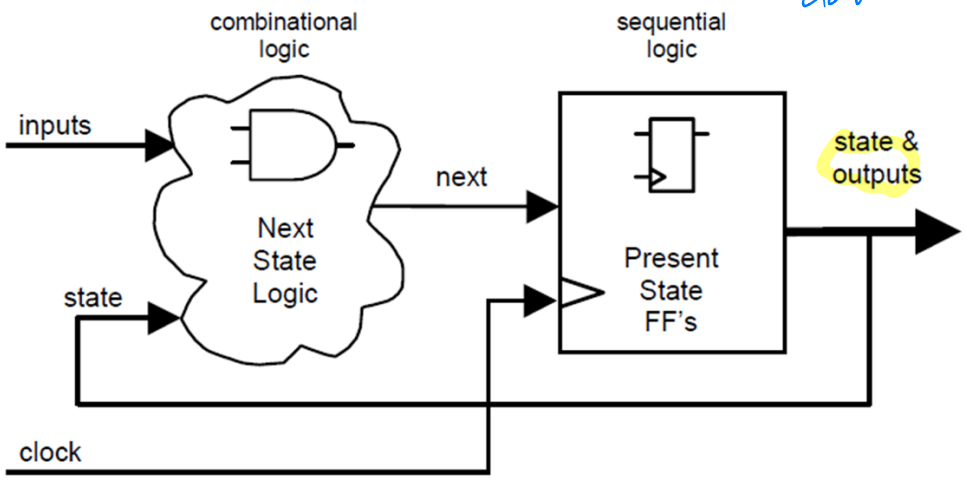
Why not both registered in and registered out?

Generate and Registering “next output”



Output Encoded FSM

select a state encoding that forces the outputs to be driven by individual state-register bits




```
module fsm1a_ffo1 (ds, rd, go, ws, clk, rst_n);
  output ds, rd;
  input  go, ws;
  input  clk, rst_n;

  // state bits = x0    ds rd
  parameter [2:0] IDLE = 3'b0_00,
                  READ = 3'b0_01,
                  DLY  = 3'b1_01,
                  DONE = 3'b0_10;

  reg [2:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) state <= IDLE;
    else        state <= next;

  always @(state or go or ws) begin
    next = 3'bx;
    case (state)
      IDLE: if (go) next = READ;
            else    next = IDLE;
      READ:         next = DLY;
      DLY:  if (ws) next = READ;
            else    next = DONE;
      DONE:         next = IDLE;
    endcase
  end

  assign {ds,rd} = state[1:0];
endmodule
```

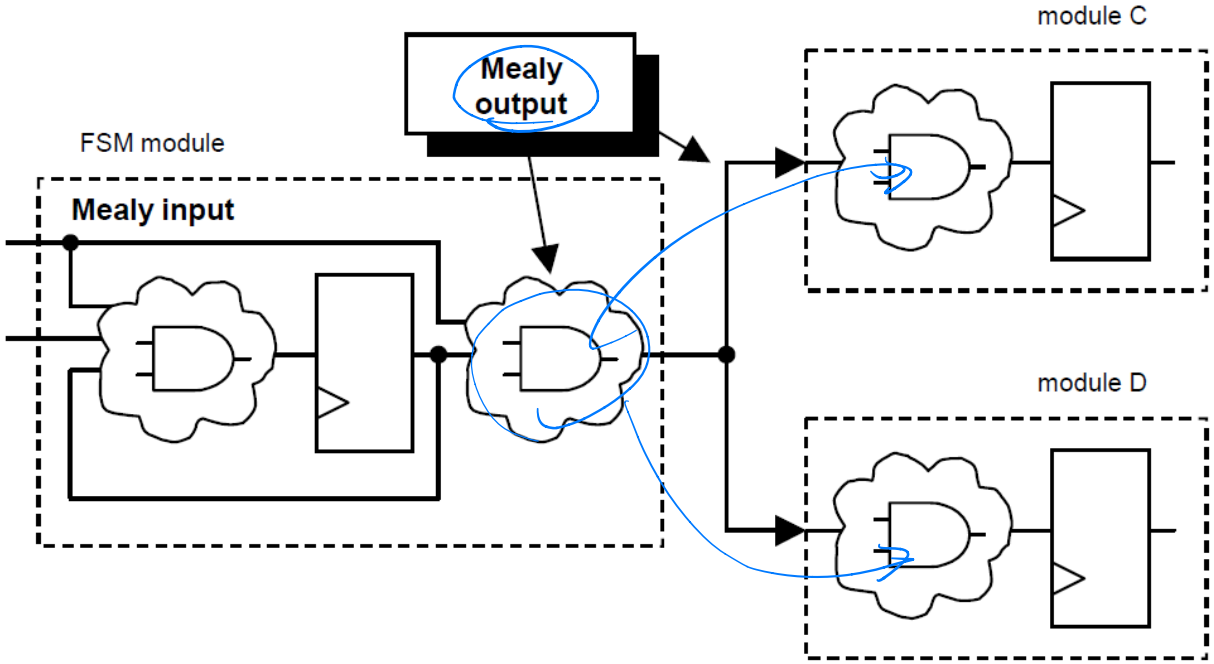
State encoding

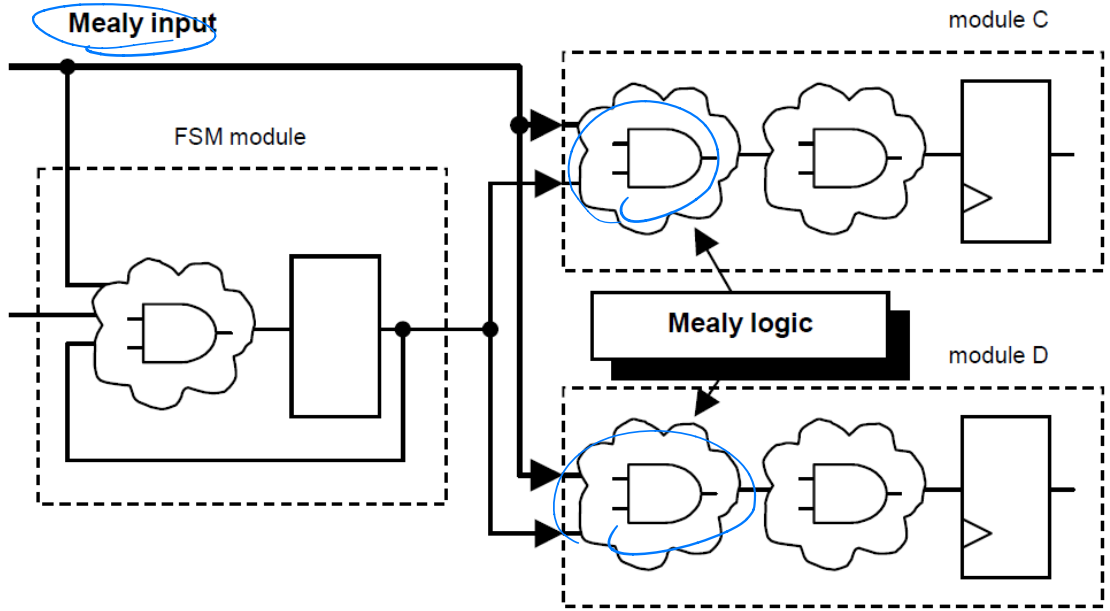
State register, sequential always block

Next state, combinational always block

Outputs are assigned directly from the state-register bits

Mealy Outputs





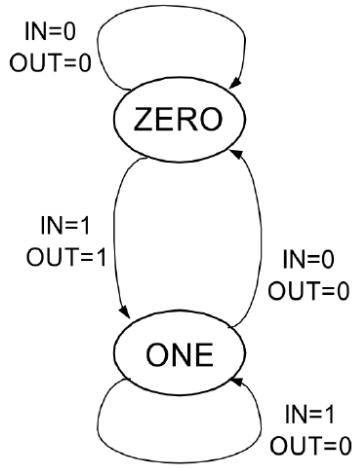
Edge Detector Implementation : Mealy & Moore FSM

Input: A bit stream that is received one bit at a time.

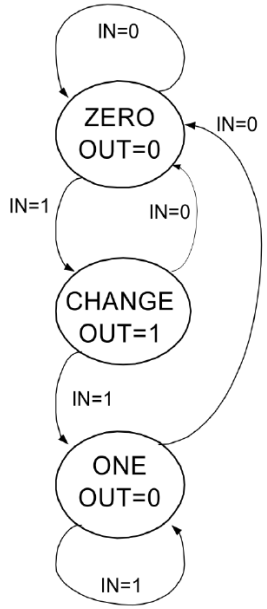
Output: Asserts its output to be true when the input bit stream changes from 0 to 1.

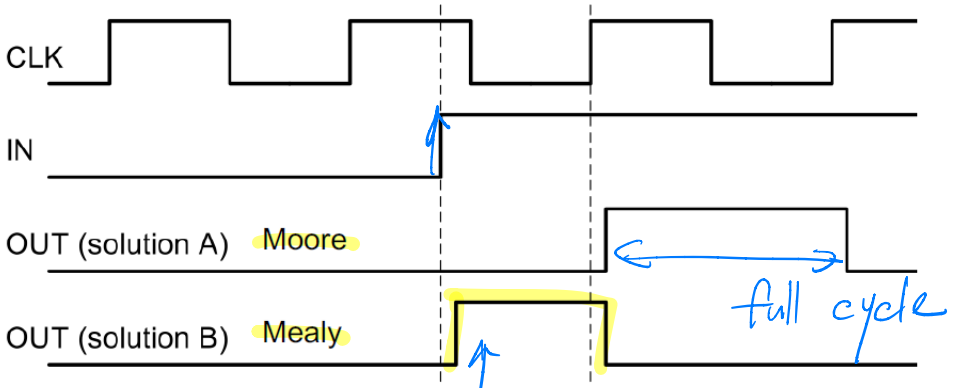
detect rising edge → gen. pulse

Mealy FSM



Moore FSM





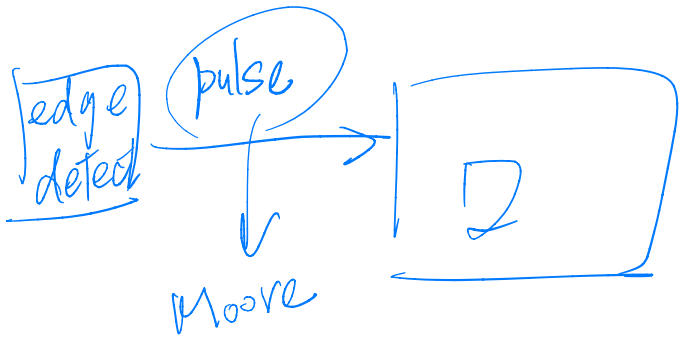
pulse width

IN strict timing require

FSM Comparison

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



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):

Summary: FSM Design Process

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