



# Digital Circuits and Systems

## Lecture 3 Model Sequential Logic

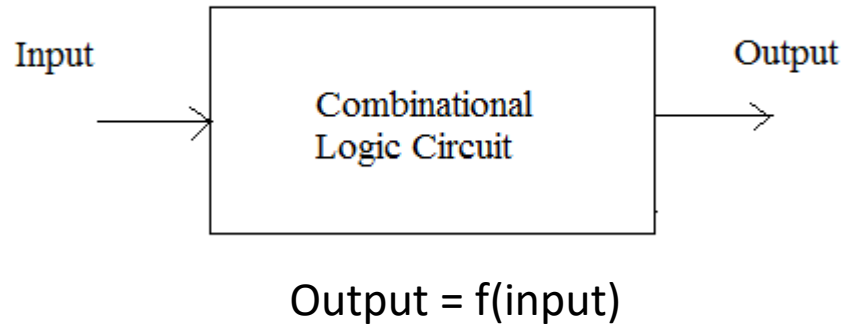
---

Tian Sheuan Chang

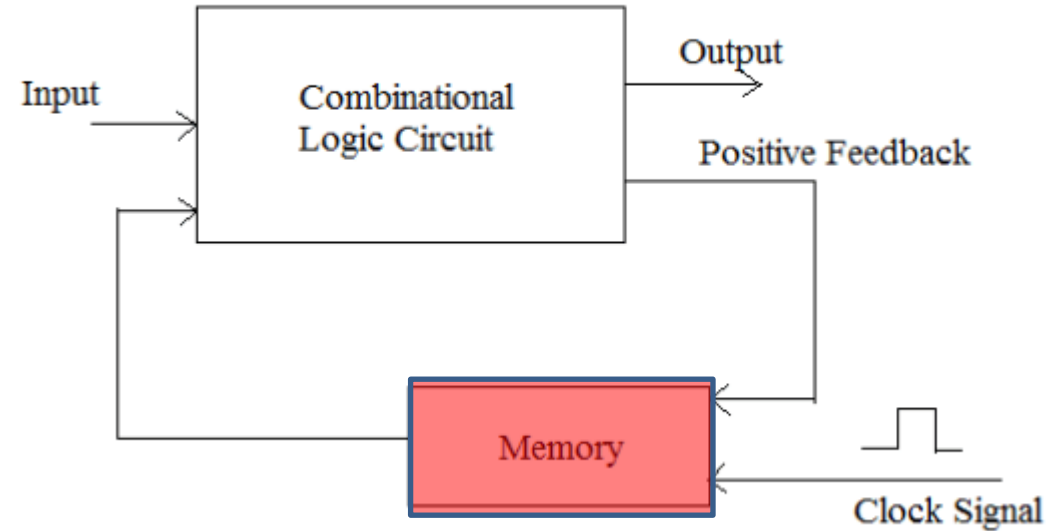
有了記憶，人生變彩色

# SEQUENTIAL CIRCUIT

# Sequential logic has state



Combinational Logic

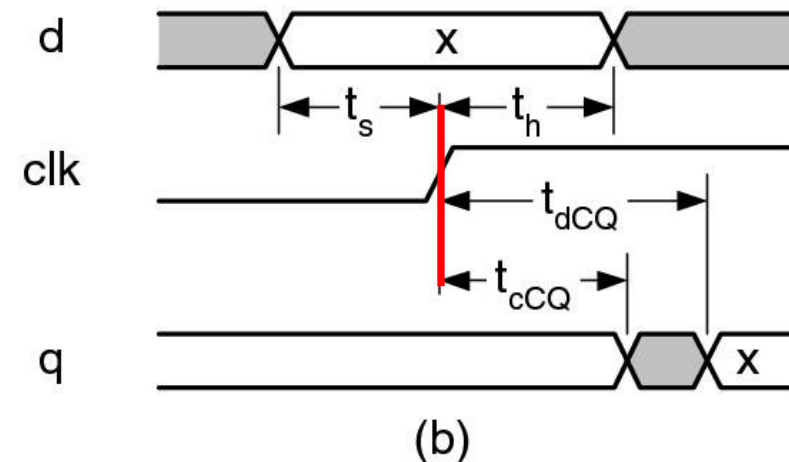
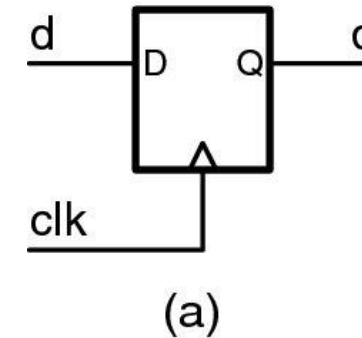


Sequential Logic

**\*without storage elements, the CL circuits with feedback may Become unstable! => oscillator**

# D Flip-Flop

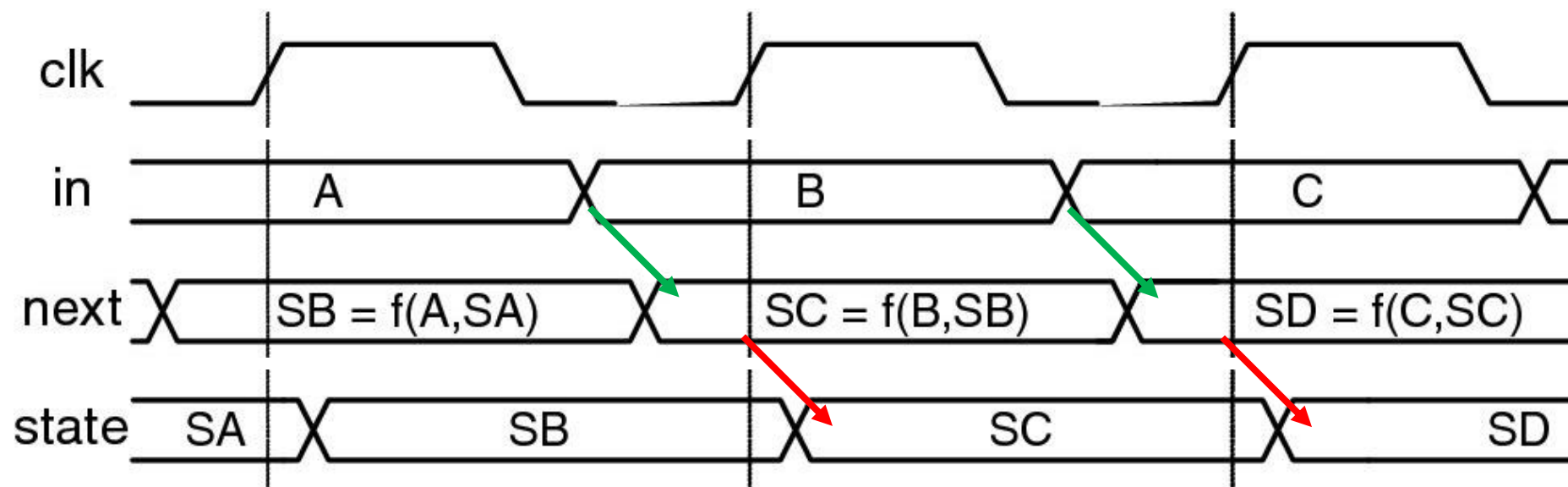
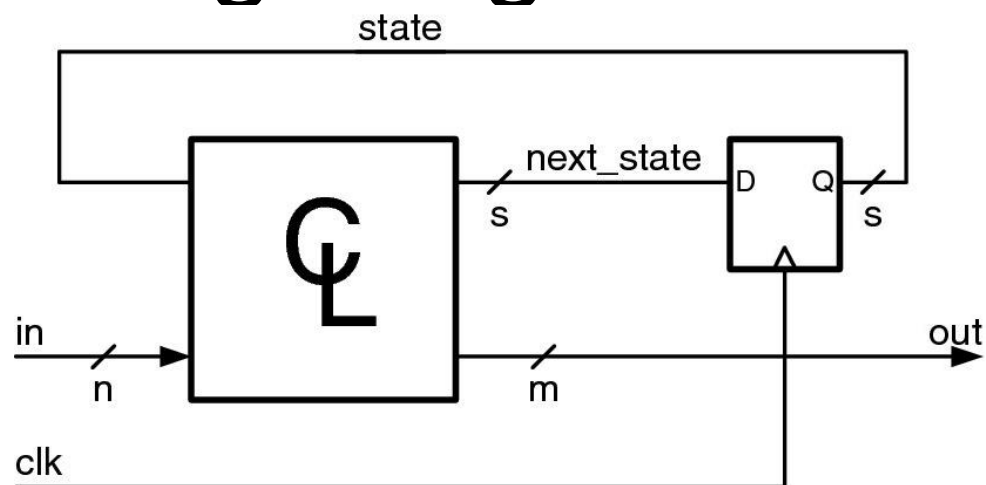
- Input: D
- Output: Q
- Clock
- Q outputs a steady value
- changes Q to be D **at clock edge**
- Flip-flop stores state
- Allows sequential circuits to iterate



**\*D-type FF is often exploited as “register” due to simple connectivity; Latch is exploited as well but level sensitive; thus only half period is available.**

一般數位電路不用latch  
Why?

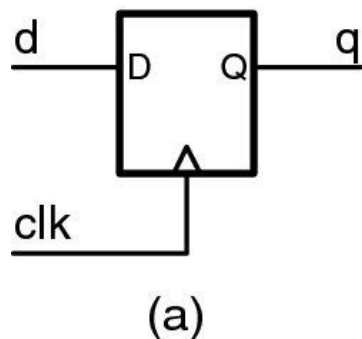
# Timing Diagram of Sequential Circuit



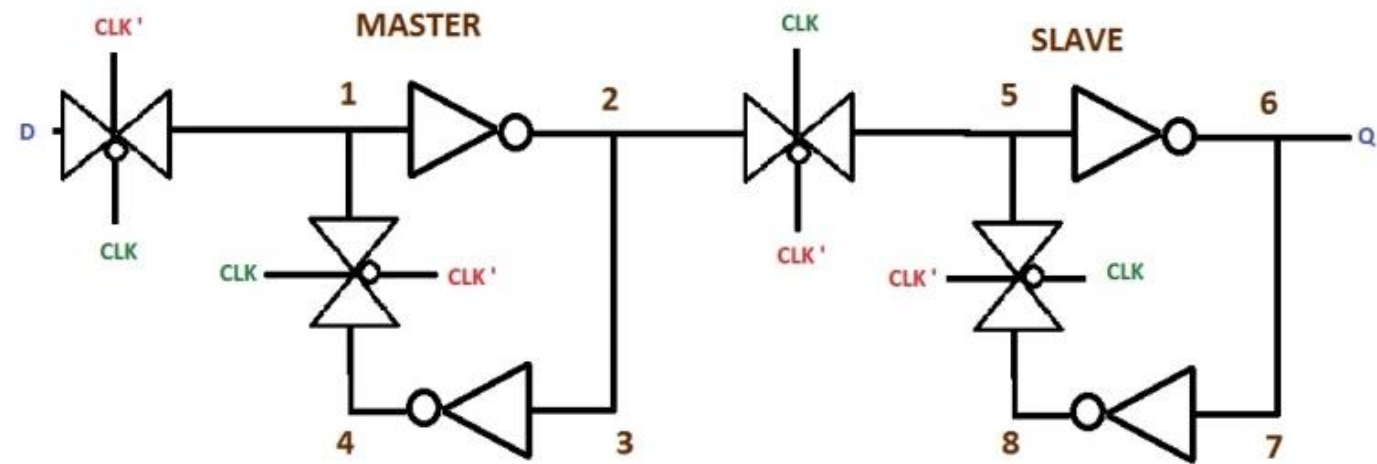
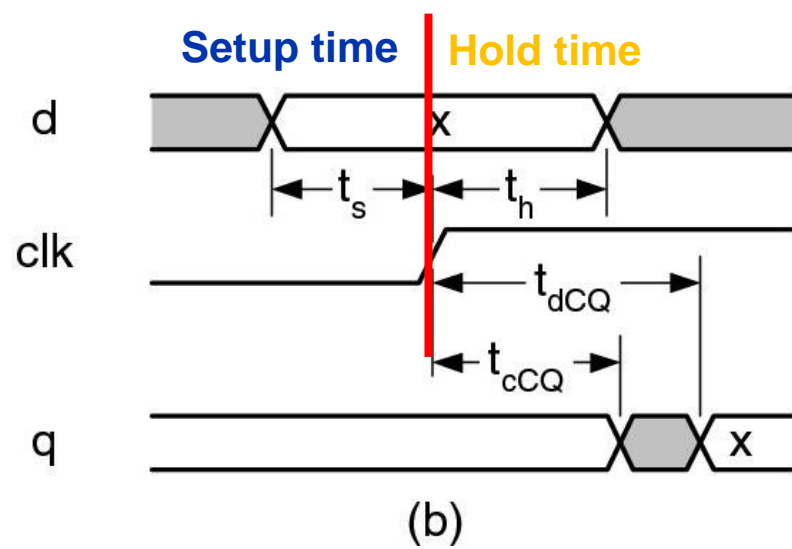
輸入變，輸出就跟著變

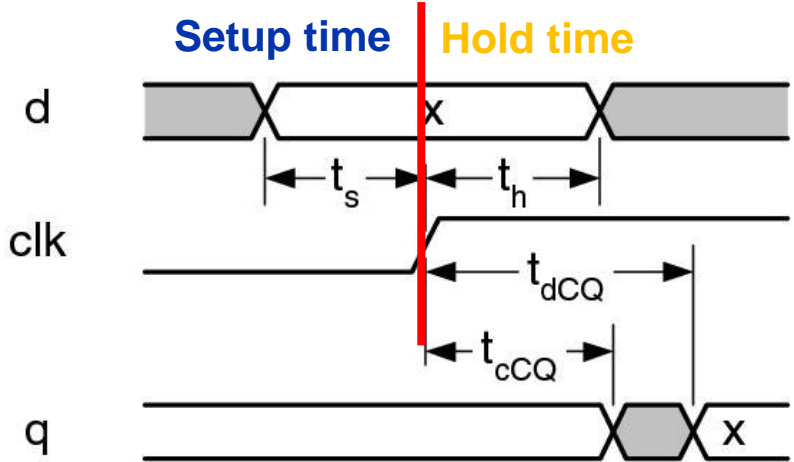
輸入變，輸出等到  
Clock edge 才變

# Timing Diagram of Sequential Circuit

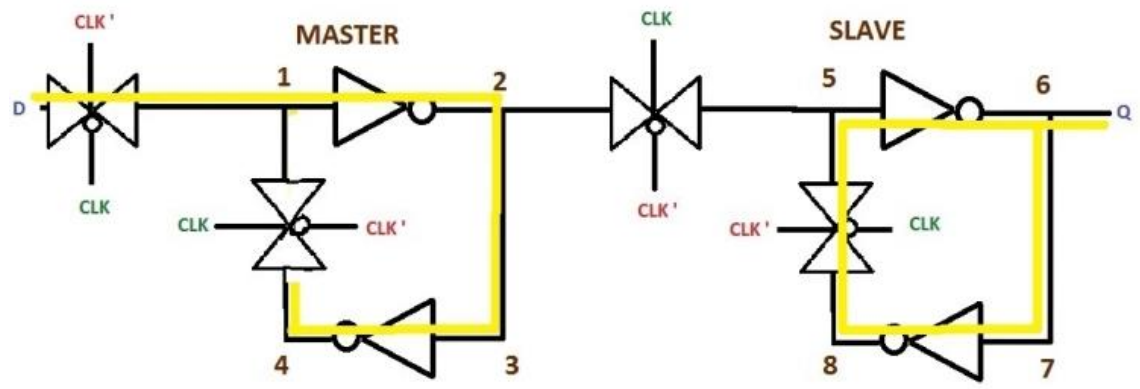


**Setup time:** inputs become stable before rising clock;  
**Hold time:** inputs remain stable after rising clock;

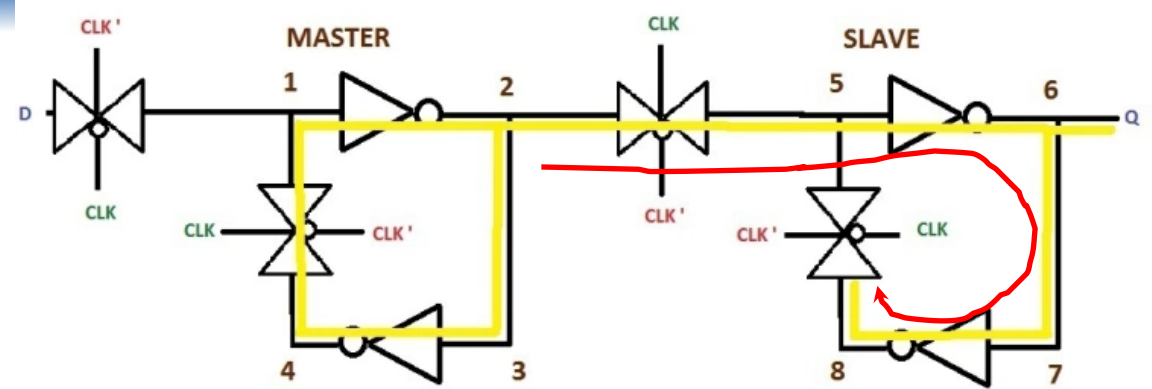




(b)

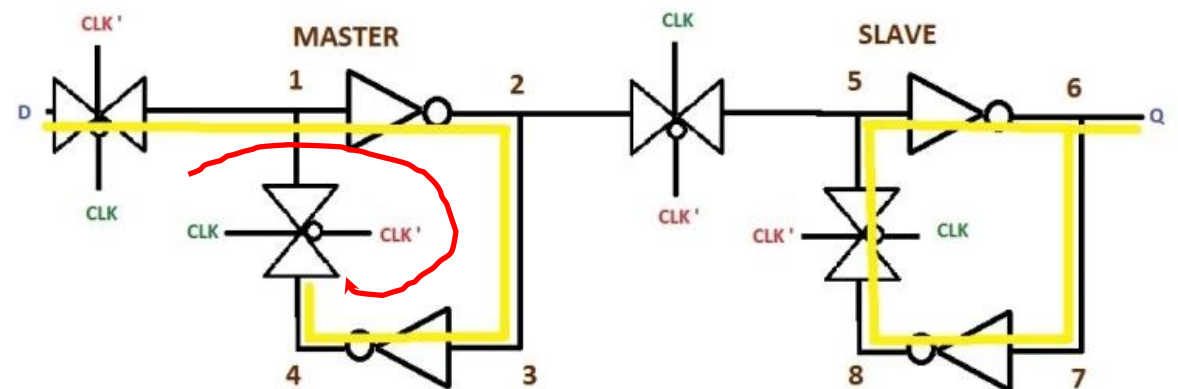


Initially D=0. CLK = LOW.  
Path D - 1 - 2 - 3 - 4. And now 4 = LOW  
Ignore the SLAVE for now.      (1)



CLK = HIGH. So SLAVE latches to LOGIC 1.  
Q = LOGIC 1.  
Active Path is 1 - 2 - 3 - 4 - 1 and 2 - 5 - 6 - 7 - 8      (2)

Hold time  
要讓輸入資料穩定到達第二個 feedback loop的控制gate



When CLK = LOW, SLAVE latches D  
Any change in D, is seen in 4 and latches on to Q during the next positive edge      (3)

Setup time: 要讓輸入資料穩定到達第一個 feedback loop的控制gate

# CODING STYLE FOR FSM

Source: C. Cummings, Synthesizable Finite State Machine Design Techniques  
Using the New SystemVerilog 3.1 Enhancements



# Moore v.s. Mealy

- Mealy and Moore FSM
  - A **Moore FSM** is a state machine where the outputs are only a function of the present **state**.
  - A **Mealy FSM** is a state machine where one or more of the outputs is a function of the present **state** and one or more of the **inputs**.

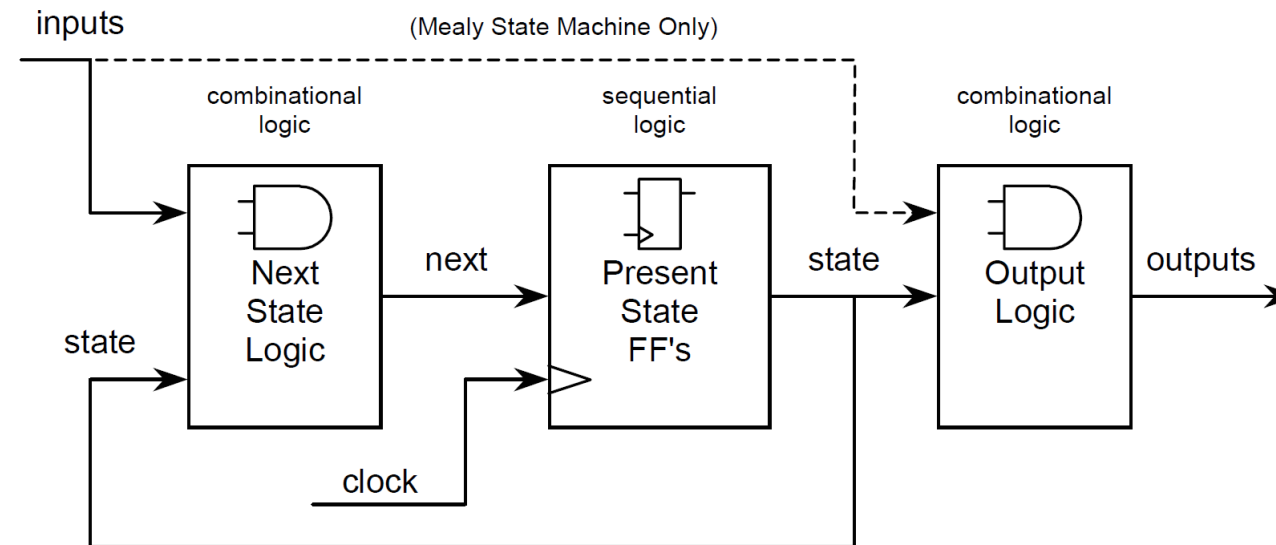
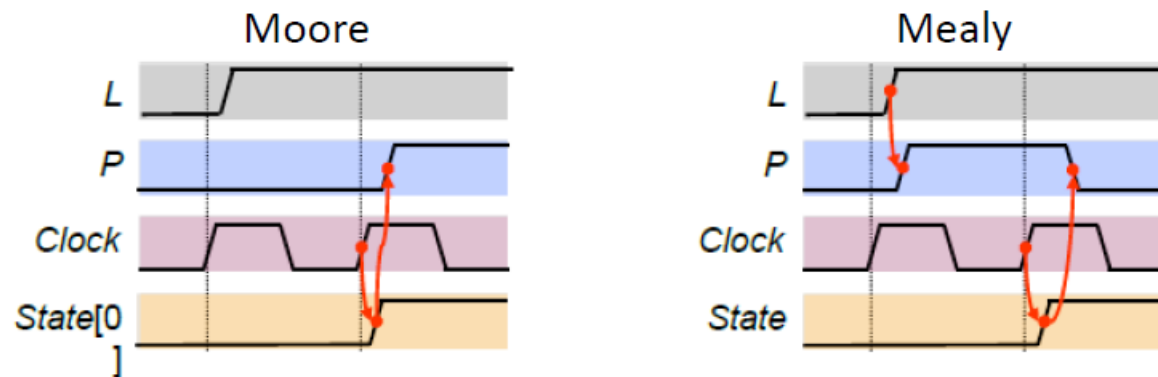


Figure 1 - FSM Block Diagram

# FSM Implementation: *Moore/Mealy trade-off*

- Remember that the difference is in the output:
  - Moore outputs are based on state only
  - Mealy outputs are based on state and input
  - Therefore, Mealy outputs generally occur **one cycle earlier** than a Moore



- Compared to a Moore FSM, a Mealy FSM might...
  - Be more difficult to conceptualize and design
  - Have fewer states

# Asynchronous Mealy

- **Registered output** to ease logic synthesis

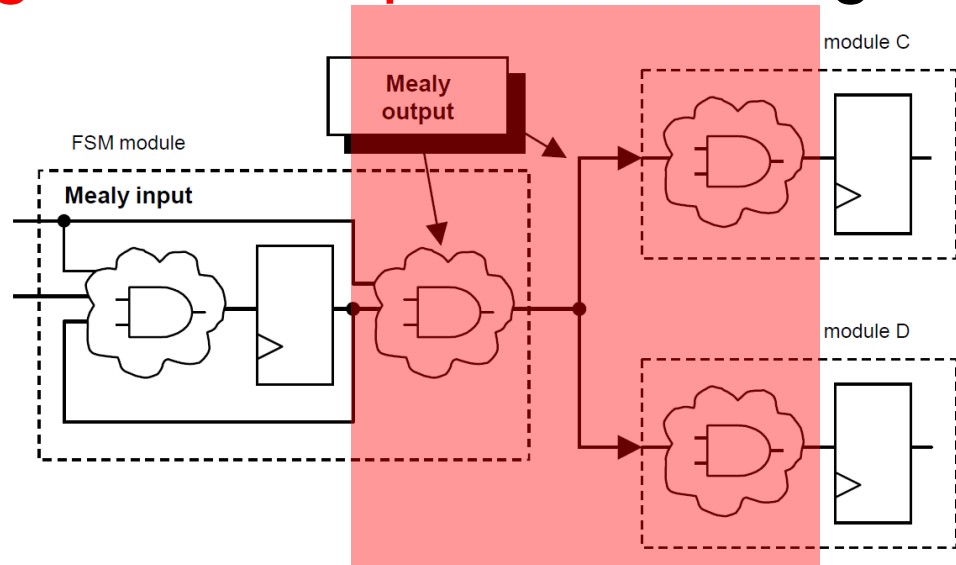


Figure 11 - FSM Mealy output driving combinational inputs

Logic will not be minimized across module boundaries unless specified in synthesis tools

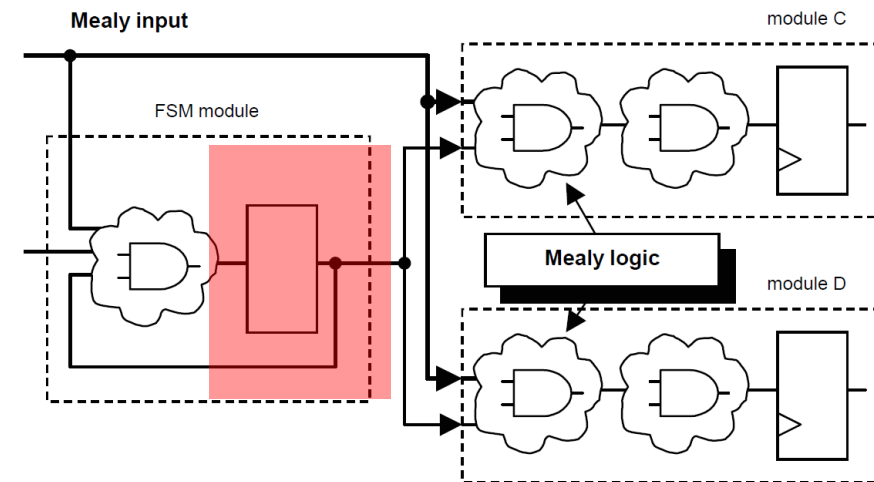
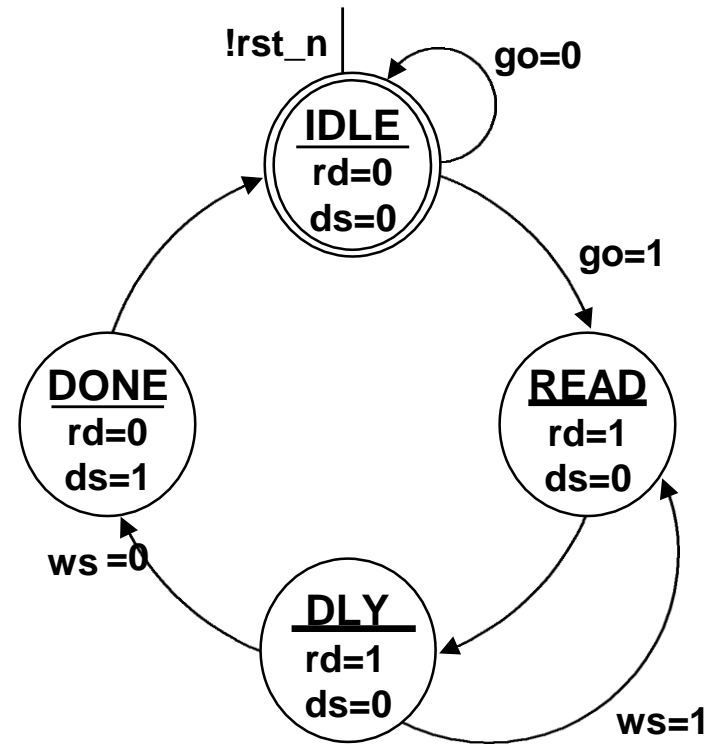


Figure 12 - Mealy logic partitioned separate from the FSM output

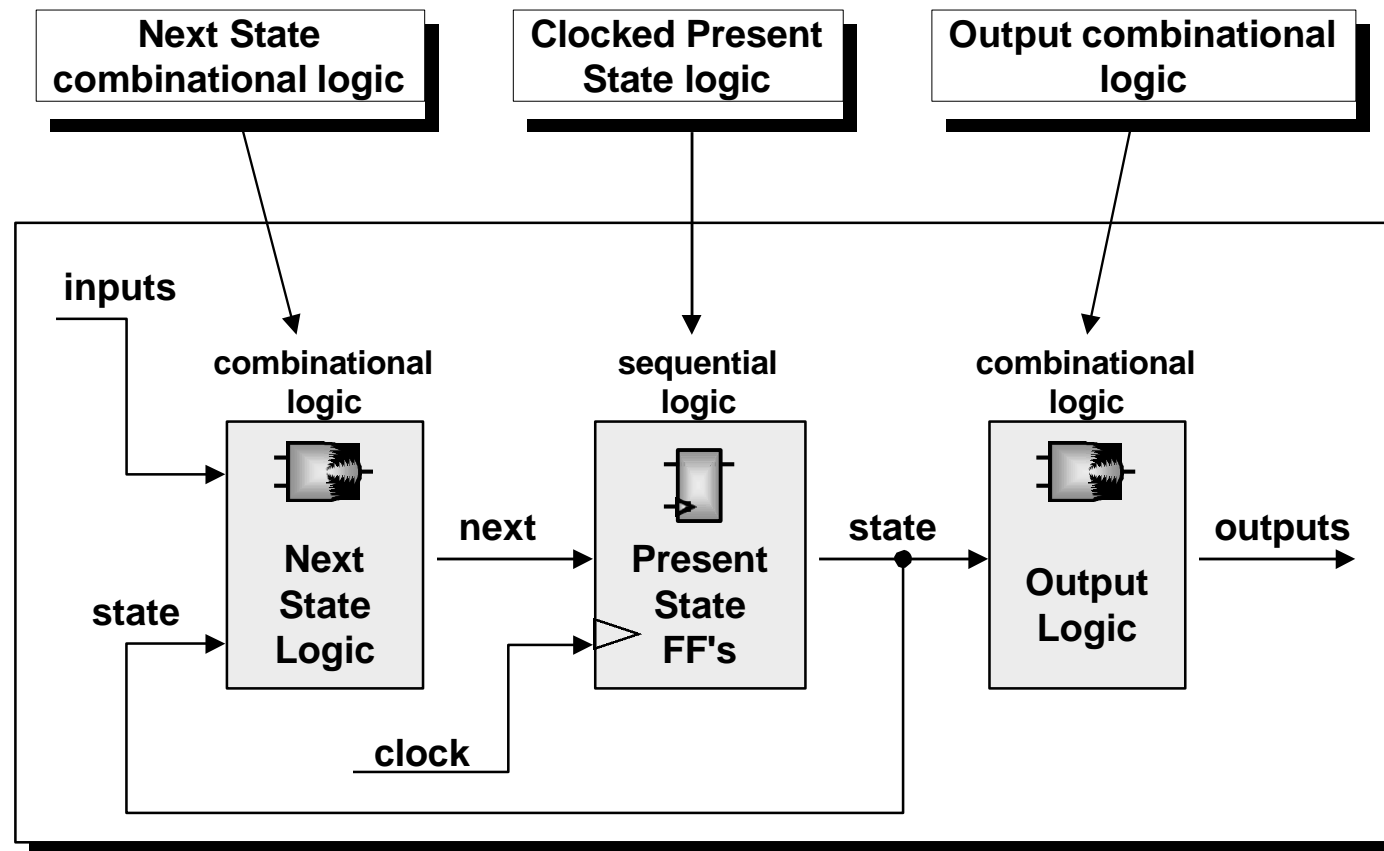
# FSM: Simple Moore Machine Example

- 4-State Moore State Machine
- Asynchronous low-true reset  
`rst_n`
- Clock signal name  
`clk`
- Two inputs  
`go, ws (wait-state)`
- Two Moore outputs  
`rd, ds (read & done-strobe)`  
(`rd=0 & ds=0` on reset)



# Moore State Machine **Two Always Blocks**

- Next state and output logic use one **always\_comb** block
- Present state FF use **always\_ff**



# Two always block

```

module fsm_cc1_2
(output logic rd, ds,
input go, ws, clk, rst_n);
//binary encoding
parameter IDLE = 2'b00,
          READ = 2'b01,
          DLY = 2'b11,
          DONE = 2'b10;

```

```

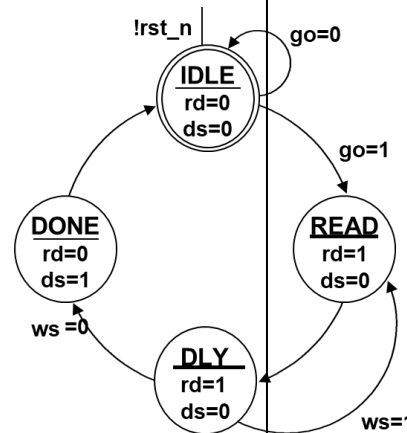
logic [1:0] state, next;

```

```

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

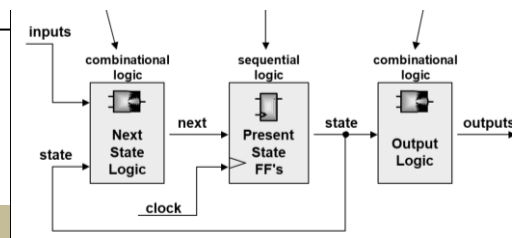
```



```

always_comb begin //next state and output logic
  next = 'bx;
  rd = 1'b0;
  ds = 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 = DONE;
              else    next = READ;
            end
    DONE : begin
              ds = 1'b1;
              next = IDLE;
            end
          endcase
end
endmodule

```



# 1. State definition

```
module fsm_cc1_2
(output logic rd, ds,
input go, ws, clk, rst_n);

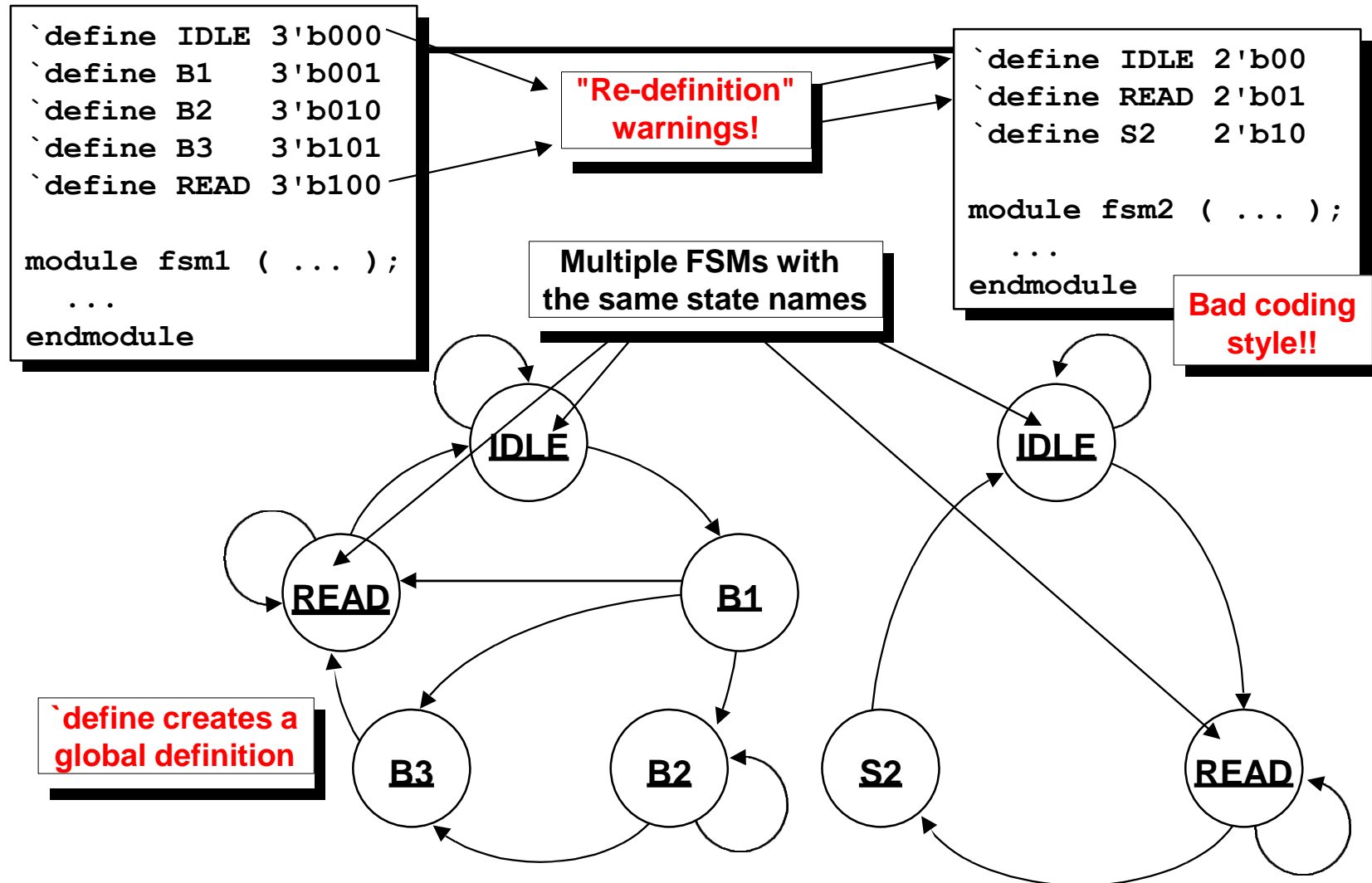
parameter IDLE = 2'b00,
          READ = 2'b01,
          DLY = 2'b11,
          DONE = 2'b10;

logic [1:0] state, next;

always_ff@(posedge clk, negedge rst_n)
    if (!rst_n) state <= IDLE;
    else state <= next;
```

State assignment? Synopsys FSM compiler can help

# Why not use define? It's a global variable





# State definition: use **parameter (local variable)**

```

module fsm1 ( ... );
  ...
  parameter
    IDLE = 3'b000,
    B1   = 3'b001,
    B2   = 3'b010,
    B3   = 3'b101,
    READ = 3'b100;
  ...
endmodule

```

No state definition  
problems!

Multiple FSMs with  
the same state names

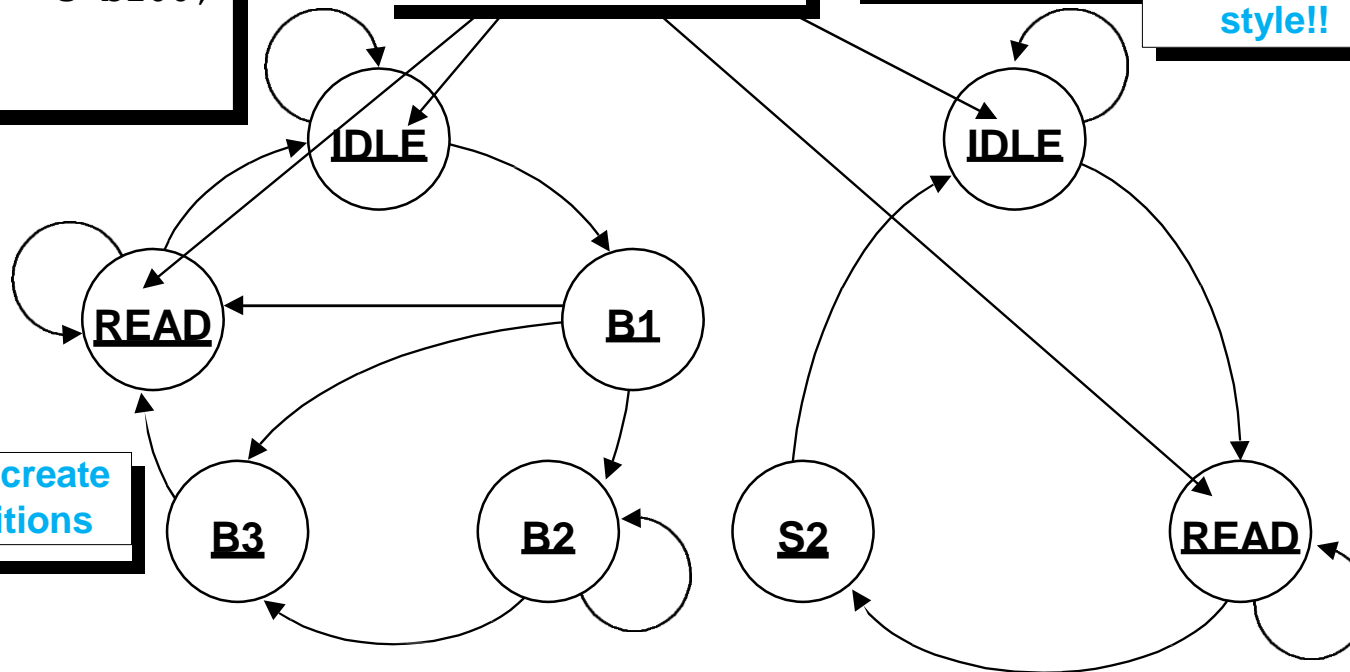
```

module fsm2 ( ... );
  ...
  parameter IDLE = 2'b00,
            READ  = 2'b01,
            S2    = 2'b10;
  ...
endmodule

```

Good coding  
style!!

parameters create  
local definitions



## 2. Present state DFF

```
module fsm_cc1_2
(output logic rd, ds,
input go, ws, clk, rst_n);

parameter      IDLE = 2'b00,
                READ = 2'b01,
                DLY  = 2'b11,
                DONE = 2'b10;

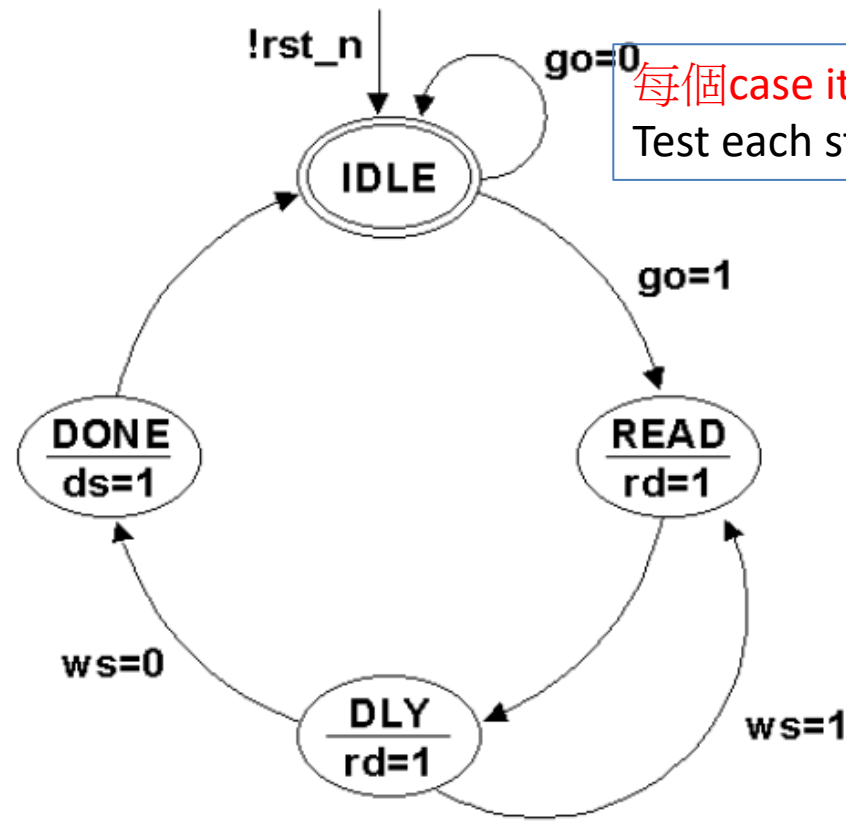
logic [1:0] state, next;

always_ff@(posedge clk, negedge rst_n)
    if (!rst_n) state <= IDLE;
    else state <= next;
```

State variable

Remember to add reset state

# 3. Next state/ output logic



Default case  
Use 'bx for logic min

每個case item放不同state  
Test each state

Test input conditions

Output logic

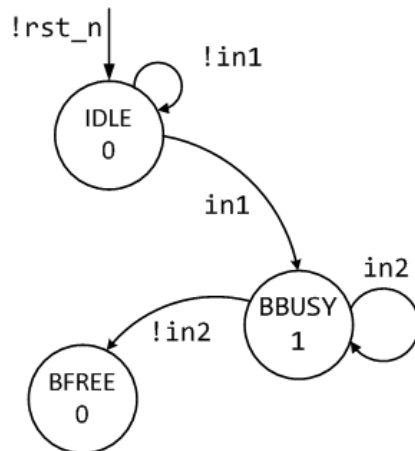
```

always_comb begin //next state and output logic
    next = 'bx;
    rd = 1'b0;
    ds = 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 = DONE;
        else
            next = READ;
    end
    DONE : begin
        ds = 1'b1;
        next = IDLE;
    end
    endcase
end
endmodule
  
```

Assign next state output

# Two Always Coding Styles

- One of the best Verilog coding styles
- Code the FSM design using two always blocks,
  - One for the sequential state register
  - One for the combinational next-state and combinational output logic.



```

parameter [1:0]
IDLE=2'b00, BBUSY=2'b01, BFREE=2'b10;
reg [1:0] state, next;

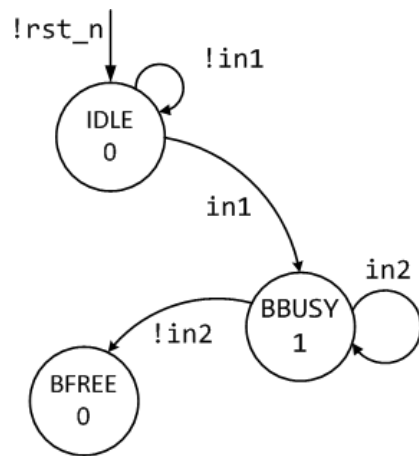
always @(posedge clk or negedge rst_n)
if (!rst_n) state <= IDLE;
else      state <= next;

always @(state or in1 or in2) begin
next = 2'bx; out1 = 1'b0;
case (state)
  IDLE : if (in1) next = BBUSY;
         else      next = IDLE;
  BBUSY: begin
         out1 = 1'b1;
         if (in2) next = BBUSY;
         else      next = BFREE;
       end
  //...
endcase
end
  
```

Combinational logic 和 sequential logic 分開

# One Always Block FSM Style *(Avoid This Style!)*

- One of the most common FSM coding styles in use today
  - It is more verbose
  - It is **more confusing**
  - It is more error prone (comparable two always block coding style)



```


parameter [1:0]
IDLE=2'b00, BBUSY=2'b01, BFREE=2'b10;
reg [1:0] state;
always @(posedge clk or negedge rst_n)
if (!rst_n) begin
    state <= IDLE;
    out1 <= 1'b0;
end
else begin
    state <= 2'bx; out1 <= 1'b0;
    case (state)
        IDLE : if (in1) begin
                    state <= BBUSY;
                    out1 <= 1'b1;
                end
            else state <= IDLE;
        BBUSY: if (in2) begin
                    state <= BBUSY;
                    out1 <= 1'b1;
                end
            else state <= BFREE;
    endcase
end
  
```

Combinational logic 和 sequential logic 放在一起

# One Always Block FSM Style *(Avoid This Style!)*

1. A declaration is made for state.  
Not for next.
  2. The state assignments do not correspond to the current state of the case statement, but the state that case statement is transitioning to.
- This is **error prone** (but it does work if coded correctly).


```
parameter [1:0]
IDLE=2'b00, BBUSY=2'b01, BFREE=2'b10;
reg [1:0] state;
always @(posedge clk or negedge
rst_n) if (!rst_n) begin
    state <= IDLE;
    out1 <= 1'b0;
end
else begin
    state <= 2'bx; out1 <= 1'b0;
    case (state)
        IDLE : if (in1) begin
            state <= BBUSY;
            out1 <= 1'b1;
        end
        else state <= IDLE;
        BBUSY: if (in2) begin
            state <= BBUSY;
            out1 <= 1'b1;
        end
        else state <= BFREE;
    endcase
end
```



# One Always Block FSM Style *(Avoid This Style!)*

3. There is just one sequential always block, coded using nonblocking assignments.
  4. All outputs will be registered (unless the outputs are placed into a separate combinational always block or assigned using continuous assignments).
- **No asynchronous Mealy outputs** can be generated from a single synchronous always block.

```
parameter [1:0]
IDLE=2'b00, BBUSY=2'b01, BFREE=2'b10;
reg [1:0] state;
always @(posedge clk or negedge
rst_n) if (!rst_n) begin
    state <= IDLE;
    out1 <= 1'b0;
end
else begin
    state <= 2'bx; out1 <= 1'b0;
    case (state)
        IDLE : if (in1) begin
                    state <= BBUSY;
                    out1 <= 1'b1;
                end
            else state <= IDLE;
        BBUSY: if (in2) begin
                    state <= BBUSY;
                    out1 <= 1'b1;
                end
            else state <= BFREE;
    endcase
end
```



# TLDR

- Use two always or three always coding style
  - Two always: one for state DFF, one for next state and output
  - Three always: one for state DFF, one for next state, one for output