Imperial College London

Lecture 11

Finite State Machines

Peter Cheung
Imperial College London

URL: www.ee.imperial.ac.uk/pcheung/teaching/EE2_CAS/
E-mail: p.cheung@imperial.ac.uk

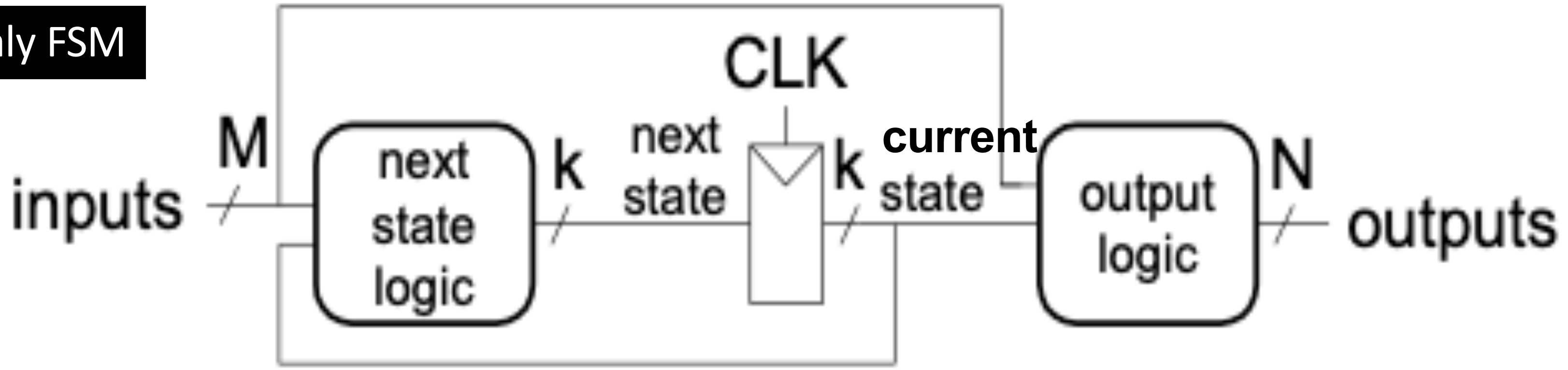
Lecture Objectives

- To learn how to **analyse** a state machine
- To learn how to **design** a state machine to meet specific objectives
- Learn how to specify a FSM in SystemVerilog
- How to combine a FSM with a counter to control state transition

Synchronous State Machines

◆ **Synchronous State Machine (also called Finite State Machine FSM)**

Mealy FSM



- The ***current state*** is defined by the register contents
- Register has *k* flipflops $\Rightarrow 2^k$ possible states
- The state only ever changes on CLOCK↑
  - We stay in a state for an exact number of CLOCK cycles
- The state is the only memory of the past
- Output can depend on both current state and current input – **Mealy FSM**

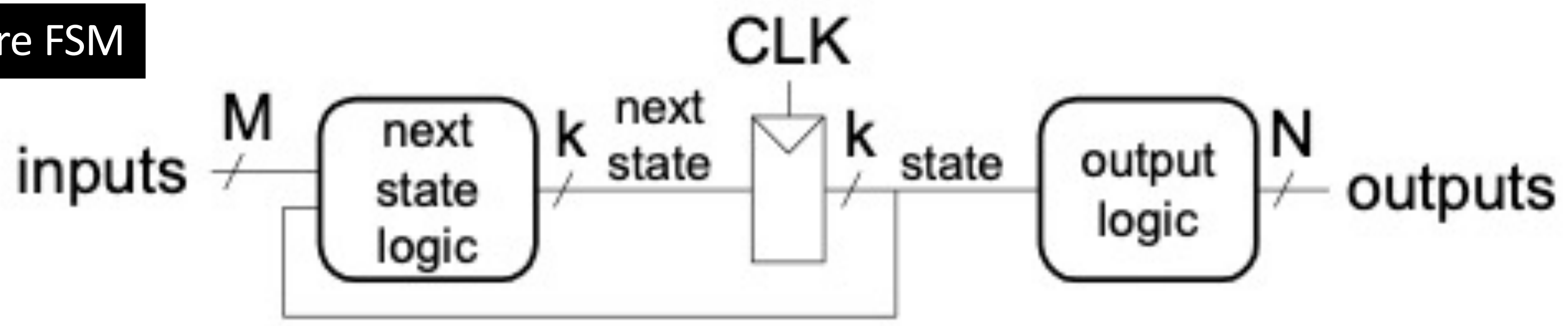
Rules:

❑ Never mess around with the clock signal

❑ Always initialise the FSM to a known initial state on reset or power ON.

Simple FSM – Moore FSM

Moore FSM



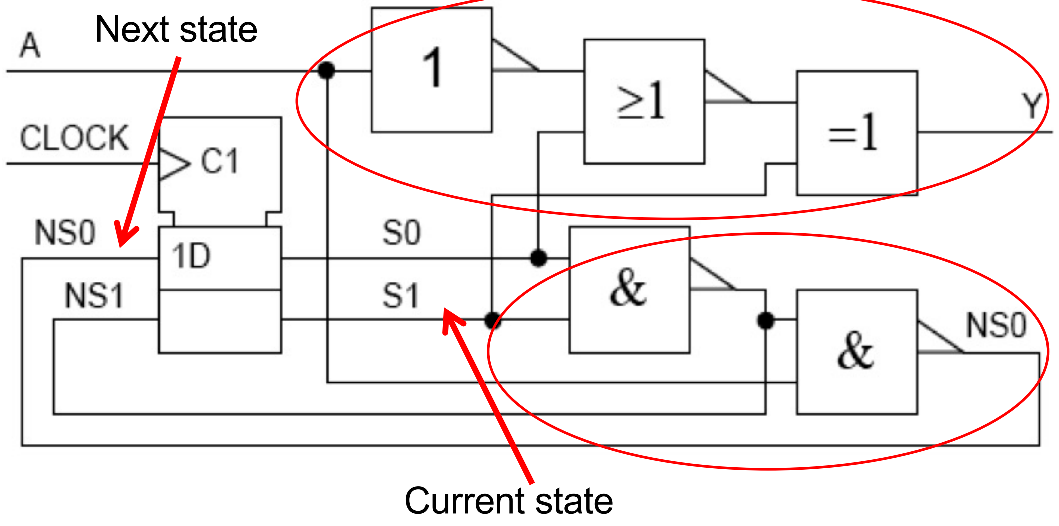
- Three parts:
  - **State registers**
  - **Next state logic**
  - **Output logic**
- **Moore FSM** – special case of Mealy FSM, output depends only on current state

Analysing a State Machine

State Table:

- Truth table for the combinational logic:
  - One row per state: n flipflops $\Rightarrow 2^n$ rows
  - One column per input combination: m input signals $\Rightarrow 2^m$ columns

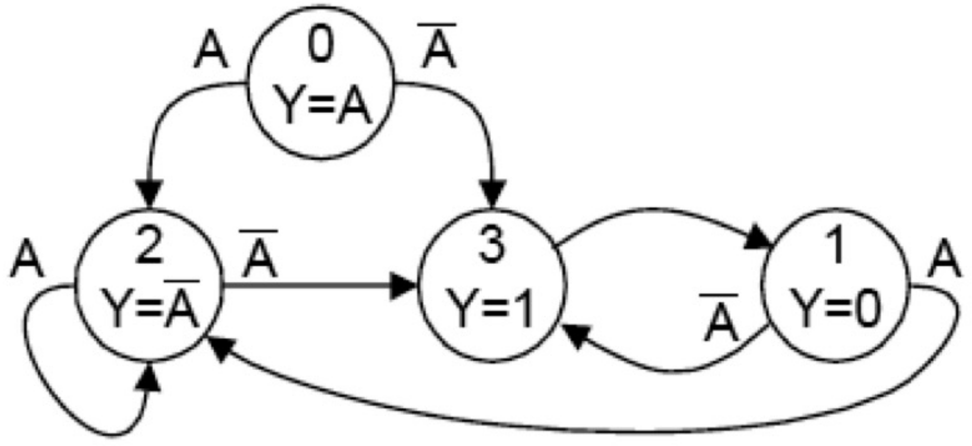
  - Each cell specifies the ***next state*** and the ***output signals during the current state***
    - for clarity, we separate the two using a /



S1,S0	NS1,NS0/Y A=0	NS1,NS0/Y A=1
00	11/0	10/1
01	11/0	10/0
10	11/1	10/0
11	01/1	01/1

Drawing the State Diagram

- Split state table into two parts: next state table and output table



NS1,NS0/Y

1,S0	A=0	A=1
00	11/0	10/1
01	11/0	10/0
10	11/1	10/0
11	01/1	01/1

Next State: NS1:0

S1:0	A=0	A=1
0	3	2
1	3	2
2	3	2
3	1	1

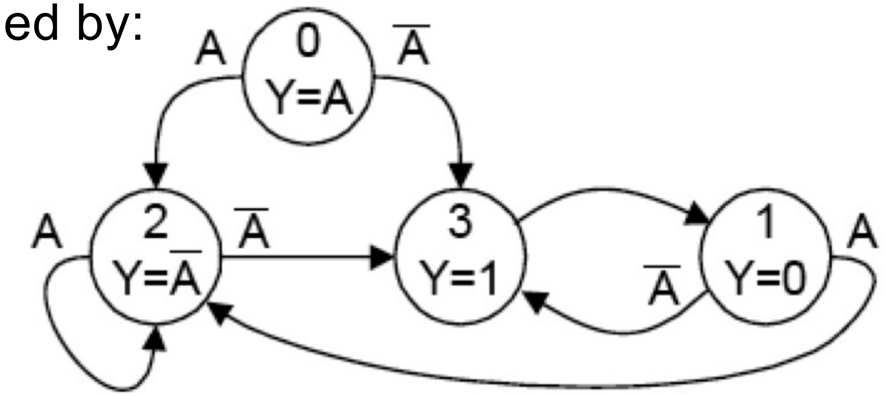
- Transition arrows are marked with Boolean expressions saying when they occur
  - Every input combination has exactly one destination.
  - Unlabelled arrows denote unconditional transitions
- Output Signals: Boolean expressions within each state

Output Signal: /Y

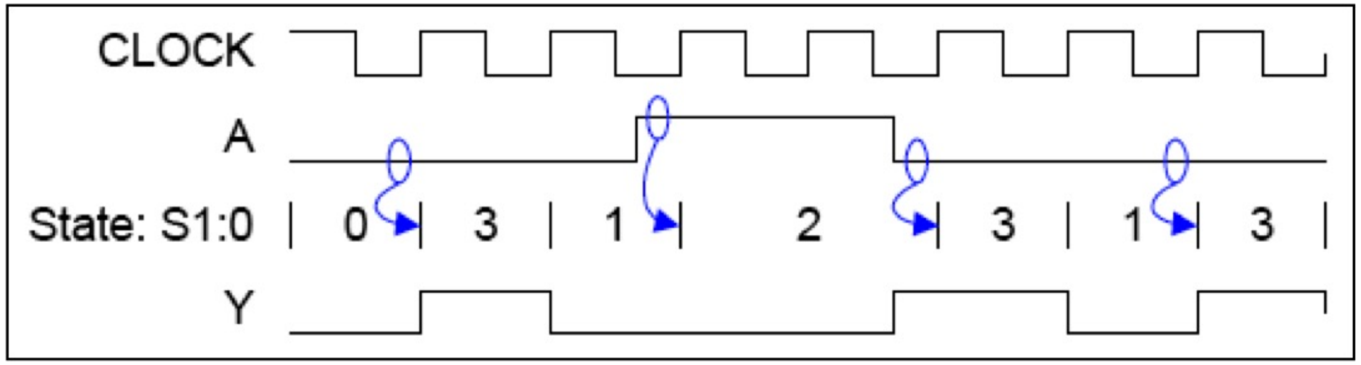
S1:0	A=0	A=1	
0	/0	/1	Y=A
1	/0	/0	Y=0
2	/1	/0	Y=!A
3	/1	/1	Y=1

Timing Diagram

- State machine behaviour is entirely determined by:
  - The initial state
  - The input signal waveforms



- **State Sequence:**
  - *Determine this first*. Next state depends on input values just before CLOCK



- **Output Signals:**

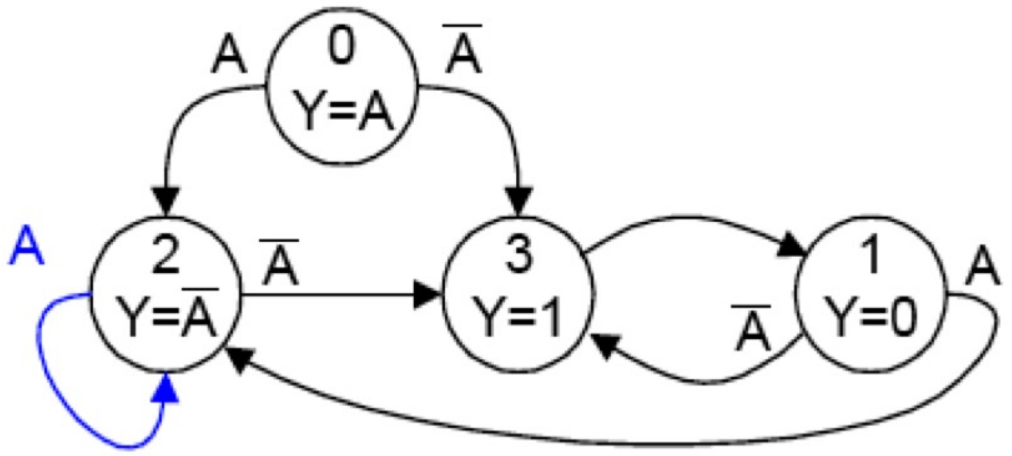
  Defined by Boolean expressions within each state.

  If all the expressions are constant 0 or 1 then outputs only ever change on clock . (***Moore machine***)

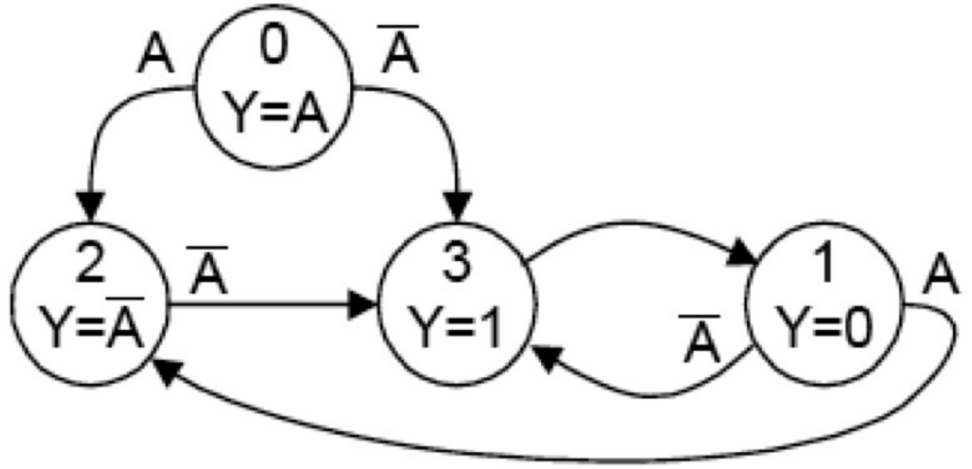
  If any expressions involve the inputs (e.g. Y=A) then it is possible for the outputs to change in the middle of a state. (***Mealy machine***)

Self-Transitions

- We can omit transitions from a state to itself
  - Aim: to save clutter on the diagram

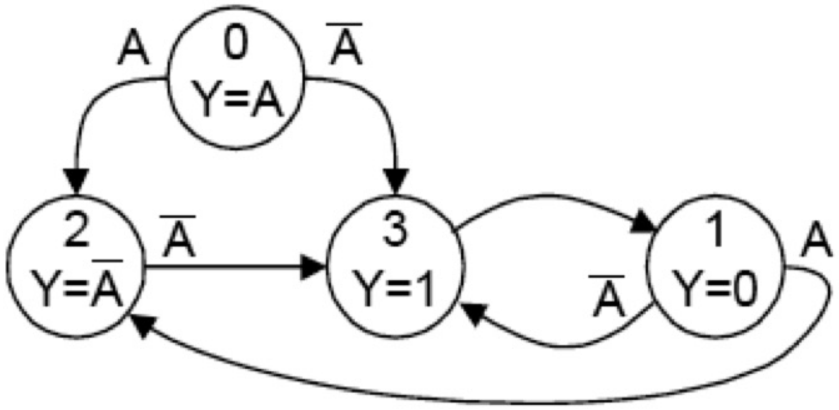


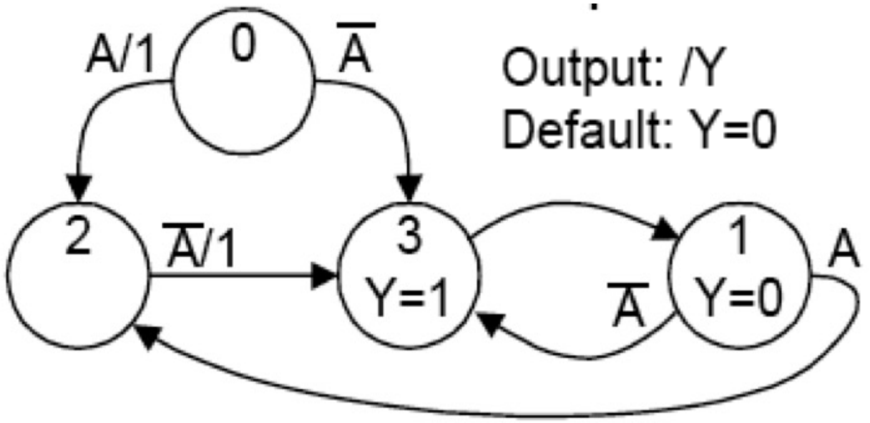
- The state machine remains in its current state if none of the transition-arrow conditions are satisfied
  - From state 2, we go to state 3 if !A occurs, otherwise we remain in state 2



Output Expressions on Arrows

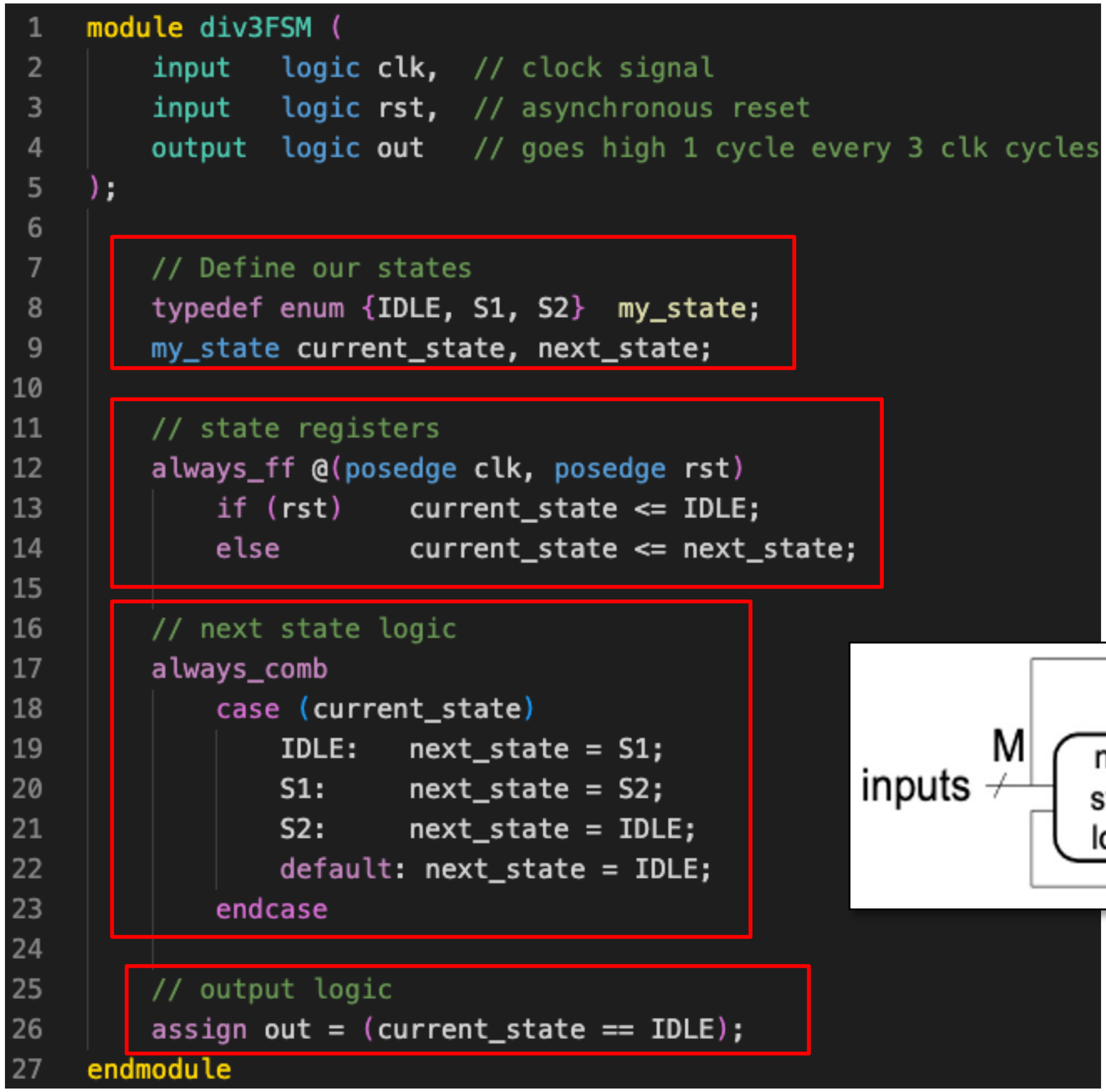
- It may make the diagram clearer to put output expressions on the arrows instead of within the state circles:
  - Useful if the same Boolean expression determines both the ***next state*** and the ***output signals***
  - For each state, the output specification must be ***either*** inside the circle *or else* on ***every*** emitted arrow
  - If self transitions are omitted, we must declare default values for the outputs



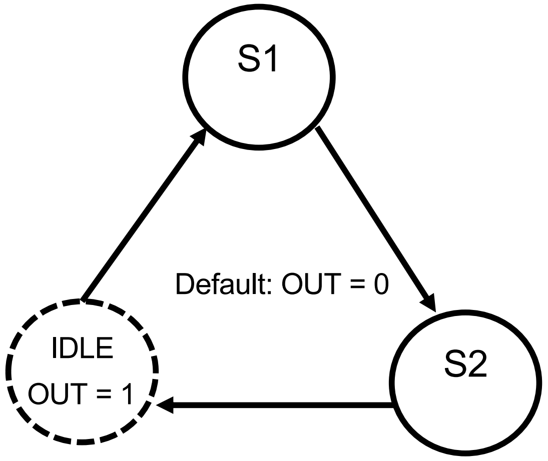


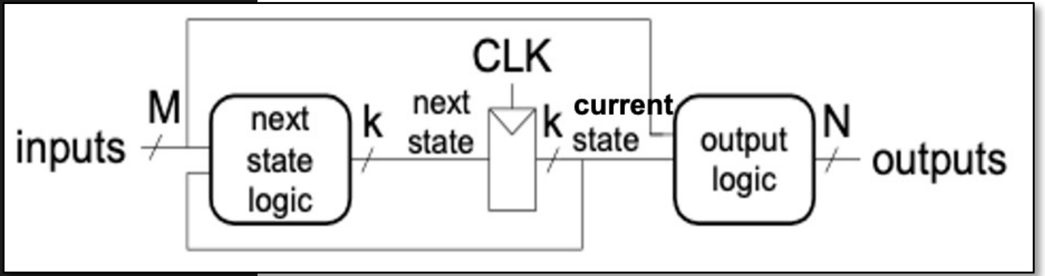
- Outputs written on an arrow apply to the state ***emitting*** the arrow.
- Outputs still apply for the entire time spent in a state
- This does not affect the Moore/Mealy distinction
- This is a notation change only

Example 1: Divide by 3 FSM (Moore)



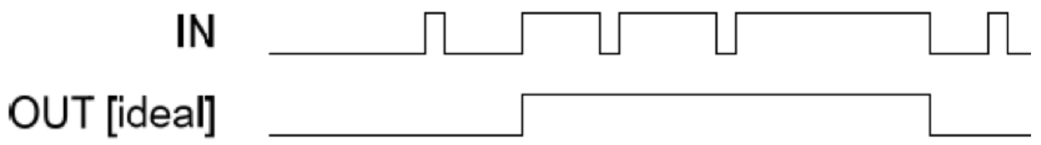
```
module div3FSM (
    input   logic clk,  // clock signal
    input   logic rst,  // asynchronous reset
    output  logic out   // goes high 1 cycle every 3 clk cycles
);

    // Define our states
    typedef enum {IDLE, S1, S2}  my_state;
    my_state current_state, next_state;

    // state registers
    always_ff @(posedge clk, posedge rst)
        if (rst)     current_state <= IDLE;
        else         current_state <= next_state;

    // next state logic
    always_comb
        case (current_state)
            IDLE:    next_state = S1;
            S1:      next_state = S2;
            S2:      next_state = IDLE;
            default: next_state = IDLE;
        endcase

    // output logic
    assign out = (current_state == IDLE);
endmodule
```





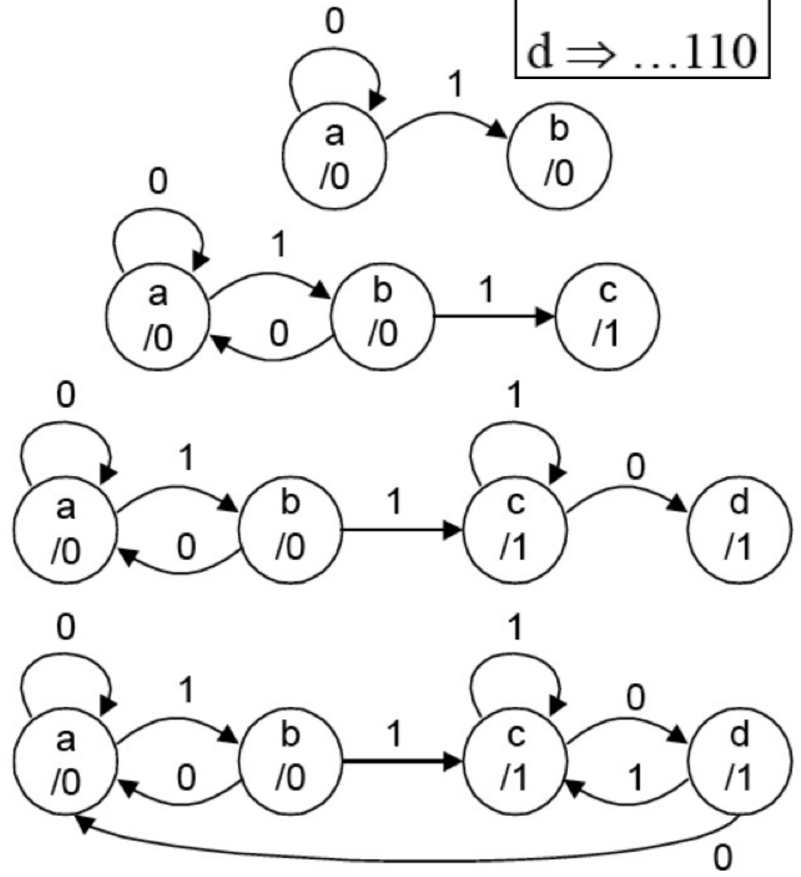
Example 2: Design a Noise Pulse Eliminator (1)

- **Design Problem**: Noise elimination circuit
  - We want to remove pulses that last only one clock cycle



$a \Rightarrow \ldots 00$
$b \Rightarrow \ldots 001$
$c \Rightarrow \ldots 11$
$d \Rightarrow \ldots 110$

- Use letters a,b,… to label states; we choose numbers later.
- Decide what action to take in each state for each of the possible input conditions.
- Use a Moore machine (i.e. output is constant in each state). Easier to design but needs more states & adds output delay.



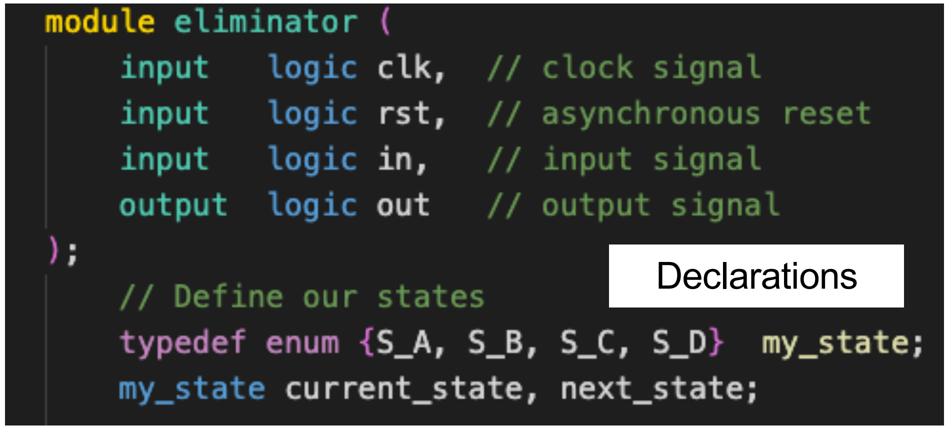
Design a Noise Pulse Eliminator (2)

1. If IN goes high for two (or more) clock cycles then OUT must go high, whereas if it goes high for only one clock cycle then OUT stays low. It follows that the two histories "IN low for ages" and "IN low for ages then high for one clock" are different because if IN is high for the next clock we need different outputs. Hence we need to introduce state b.
2. If IN goes high for one clock and then goes low again, we can forget it ever changed at all. This glitch on IN will not affect any of our future actions and so we can just return to state a.
   If on the other hand we are in state b and IN stays high for a second clock cycle, then the output must change. It follows that we need a new state, c.
3. The need for state d is exactly the same as for state b earlier. We reach state d at the end of an output pulse when IN has returned low for one clock cycle. We don't change OUT yet because it might be a false alarm.
4. If we are in state d and IN remains low for a second clock cycle, then it really is the end of the pulse and OUT must go low. We can forget the pulse ever existed and just return to state a.

Each state represents a particular history that we need to distinguish from the others:

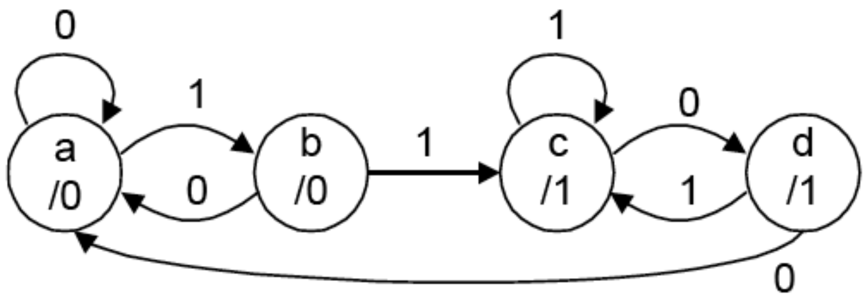
state **a**: IN=0 for >1 clock	state **b**: IN=1 for 1 clock
state **c**: IN=1 for >1 clock	state **d**: IN=0 for 1 clock

Eliminator design in SystemVerilog

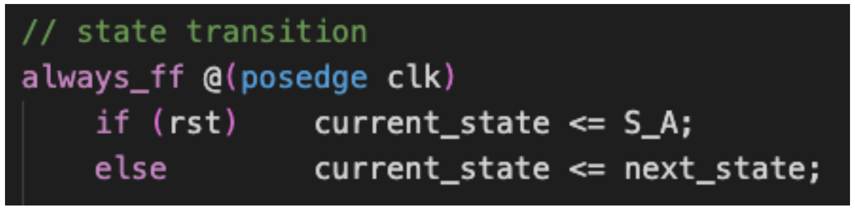


```
module eliminator (
    input   logic clk,  // clock signal
    input   logic rst,  // asynchronous reset
    input   logic in,   // input signal
    output  logic out   // output signal
);
    // Define our states
    typedef enum {S_A, S_B, S_C, S_D}  my_state;
    my_state current_state, next_state;
```

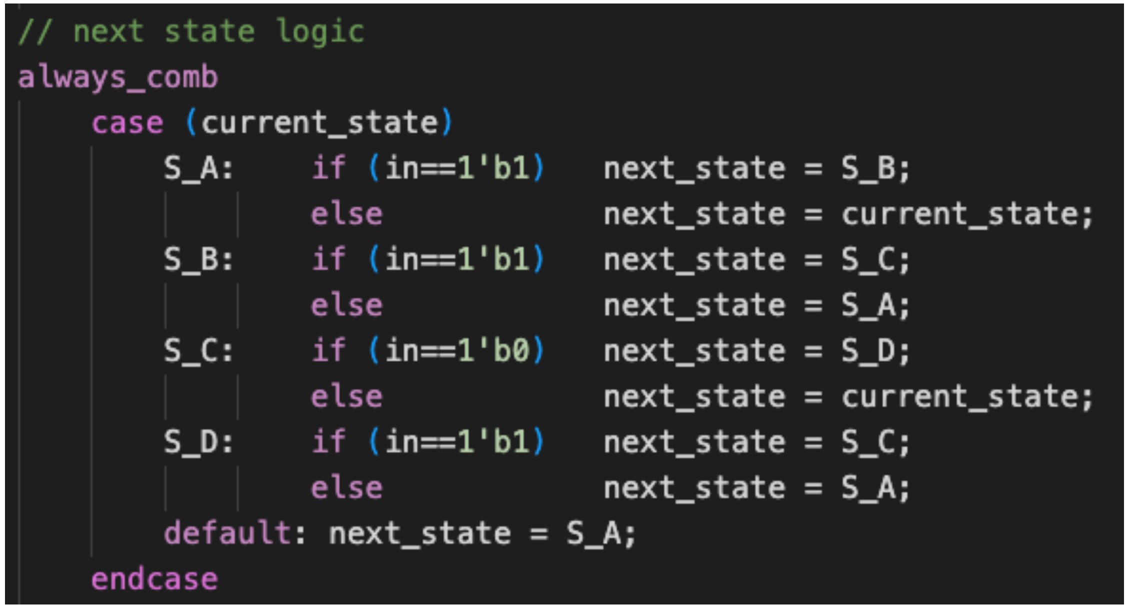
Declarations



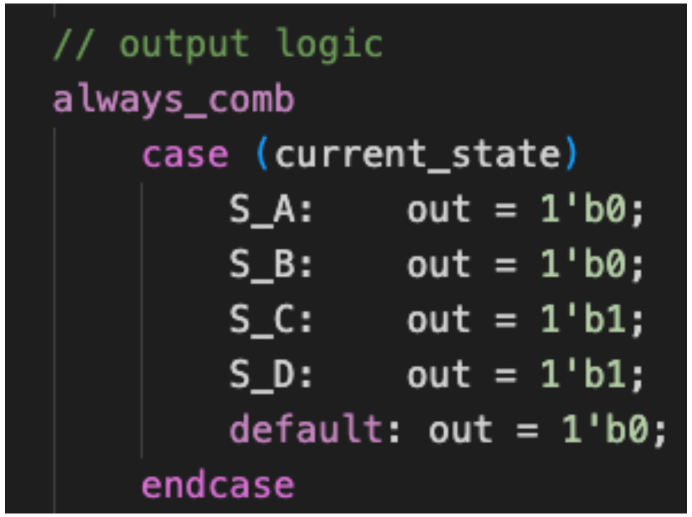
$$\begin{tikzcd} a/0 \arrow[loop above, "0"] \arrow[r, bend left, "1"] & b/0 \arrow[l, bend left, "0"] \arrow[r, "1"] & c/1 \arrow[loop above, "1"] \arrow[r, bend left, "0"] & d/1 \arrow[l, bend left, "1"] \arrow[lll, bend left, "0"] \end{tikzcd}$$



```
// state transition
always_ff @(posedge clk)
    if (rst)      current_state <= S_A;
    else          current_state <= next_state;
```



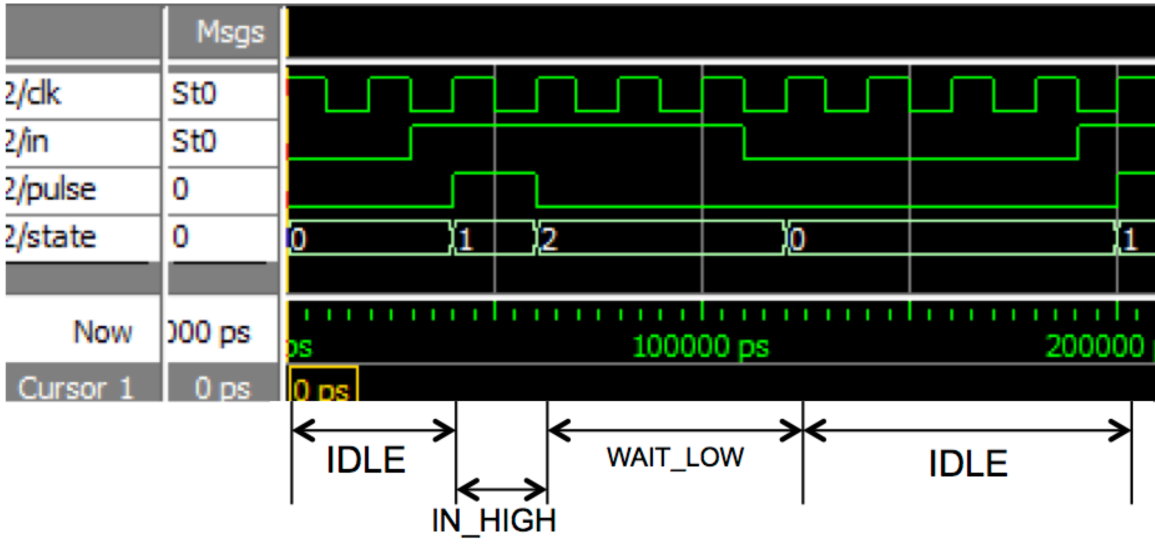
```
// next state logic
always_comb
    case (current_state)
        S_A:    if (in==1'b1)   next_state = S_B;
                else            next_state = current_state;
        S_B:    if (in==1'b1)   next_state = S_C;
                else            next_state = S_A;
        S_C:    if (in==1'b0)   next_state = S_D;
                else            next_state = current_state;
        S_D:    if (in==1'b1)   next_state = S_C;
                else            next_state = S_A;
        default: next_state = S_A;
    endcase
```

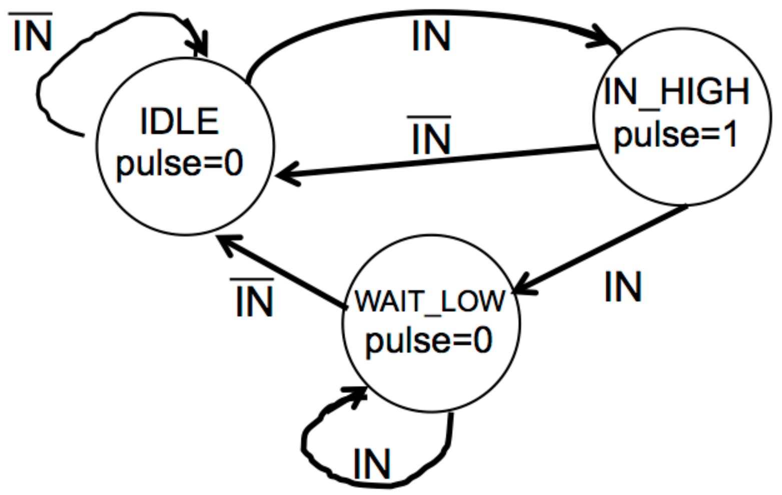


```
// output logic
always_comb
    case (current_state)
        S_A:    out = 1'b0;
        S_B:    out = 1'b0;
        S_C:    out = 1'b1;
        S_D:    out = 1'b1;
        default: out = 1'b0;
    endcase
```

Example 3 – A pulse generator

- Design a module pulse_gen.v which does the following: on each positive edge of the input signal **IN**, it generates a pulse lasting for one period of the input **clock**.

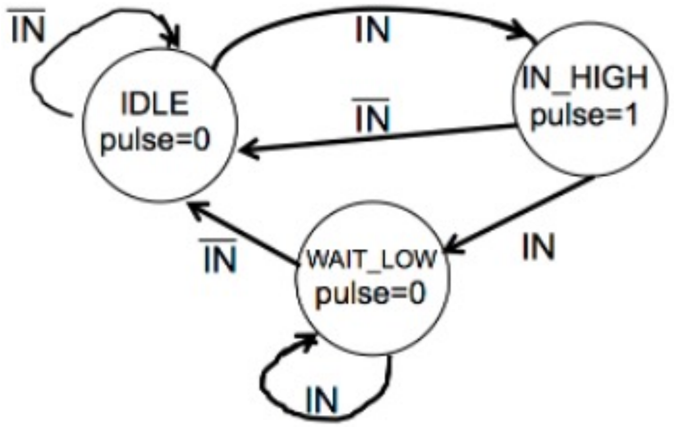


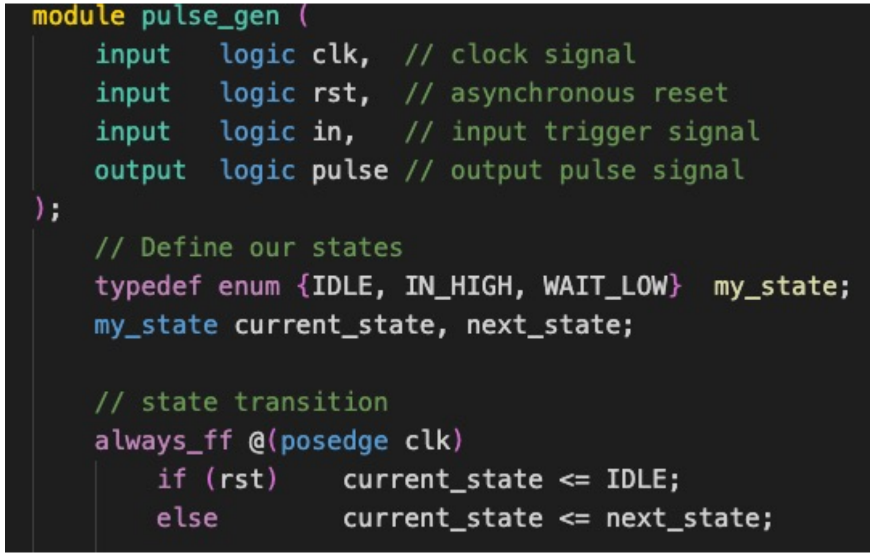


- Needs THREE states (not two).

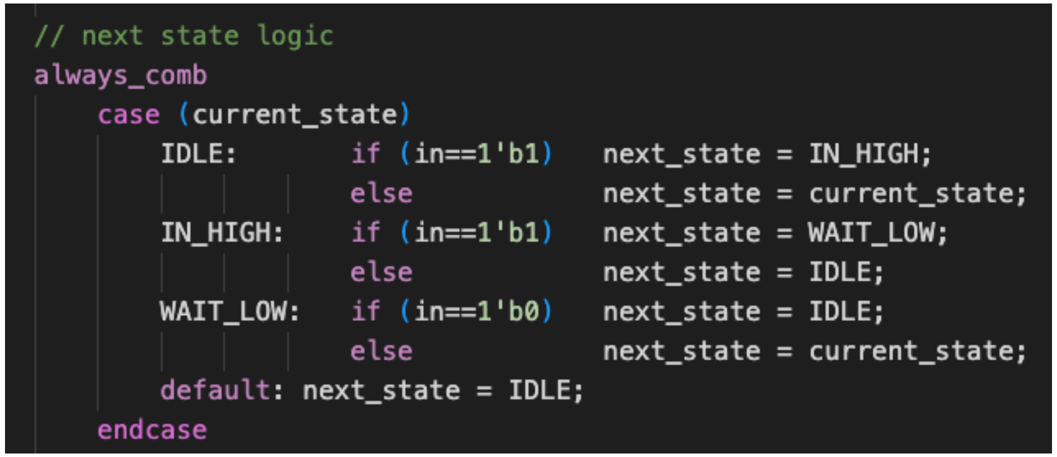
Pulse Generator in SV

- Design a module pulse_gen.v which does the following: on each positive edge of the input signal **IN**, it generates a pulse lasting for one period of the input **clk**.

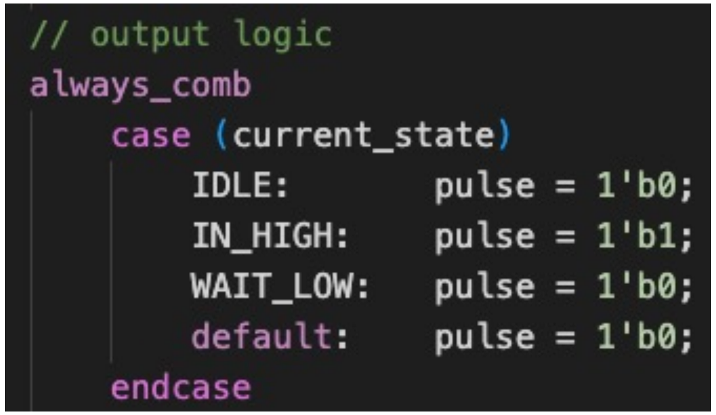




```
module pulse_gen (
    input   logic clk,   // clock signal
    input   logic rst,   // asynchronous reset
    input   logic in,    // input trigger signal
    output  logic pulse  // output pulse signal
);
    // Define our states
    typedef enum {IDLE, IN_HIGH, WAIT_LOW}  my_state;
    my_state current_state, next_state;

    // state transition
    always_ff @(posedge clk)
        if (rst)     current_state <= IDLE;
        else         current_state <= next_state;
```



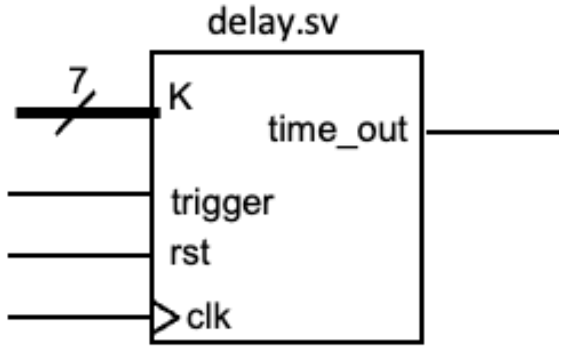
```
// next state logic
always_comb
    case (current_state)
        IDLE:       if (in==1'b1)   next_state = IN_HIGH;
                    else            next_state = current_state;
        IN_HIGH:    if (in==1'b1)   next_state = WAIT_LOW;
                    else            next_state = IDLE;
        WAIT_LOW:   if (in==1'b0)   next_state = IDLE;
                    else            next_state = current_state;
        default: next_state = IDLE;
    endcase
```

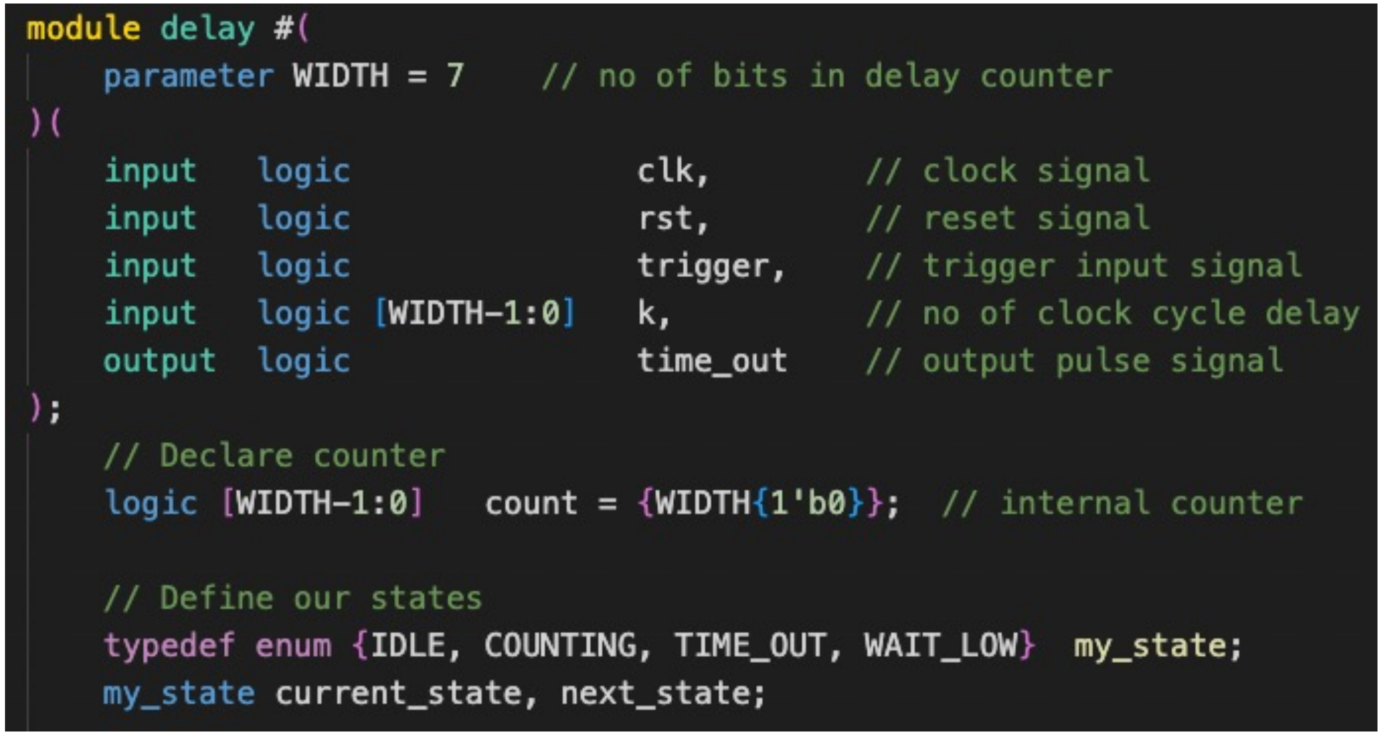


```
// output logic
always_comb
    case (current_state)
        IDLE:       pulse = 1'b0;
        IN_HIGH:    pulse = 1'b1;
        WAIT_LOW:   pulse = 1'b0;
        default:    pulse = 1'b0;
    endcase
```

Example 4: delay module (1)

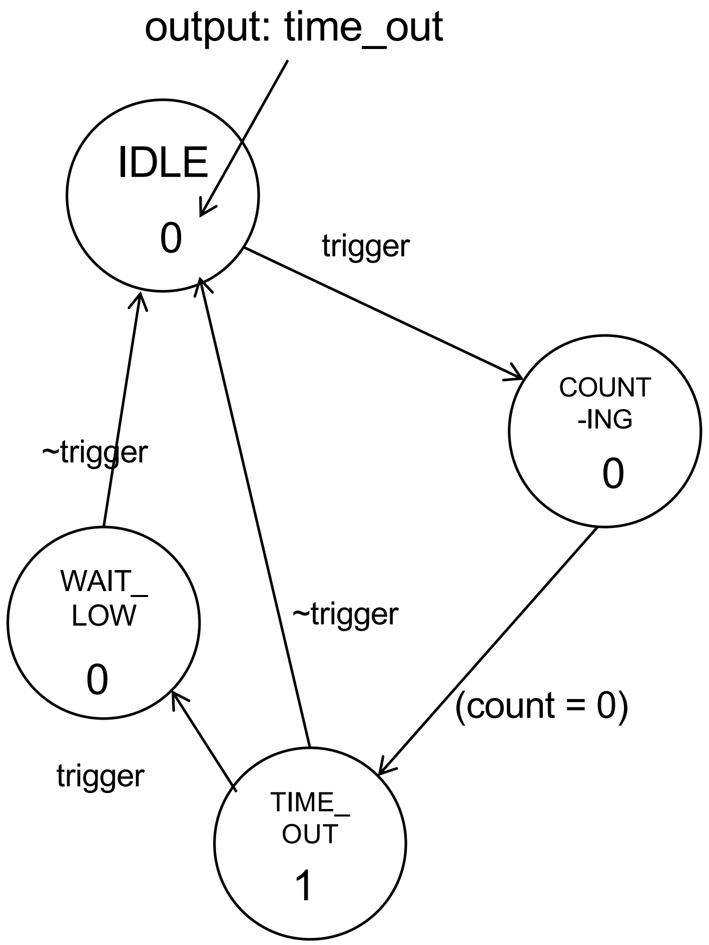
- Here is a very useful module that combines a FSM with a counter.
- It detects the rising edge on trigger, then wait (delay) for n clk cycles before producing a 1-cycle pulse on time_out.
- The external port interface for this module is shown below. We assume that n is a 7-bit number, or a maximum of 127 sysclk cycles delay.

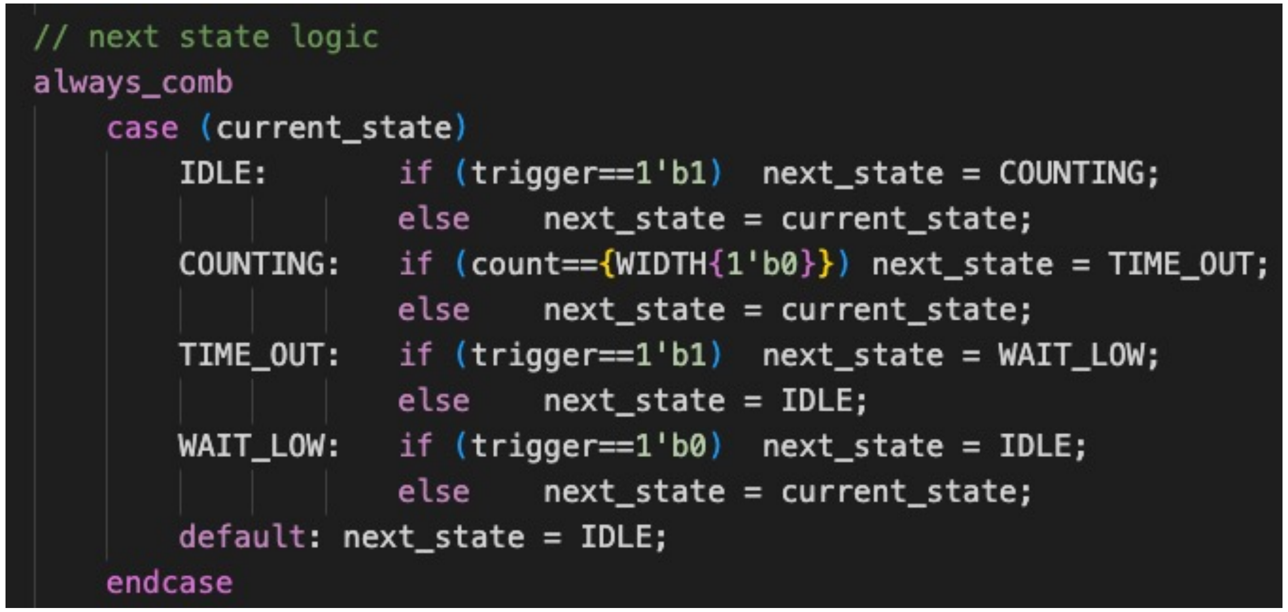




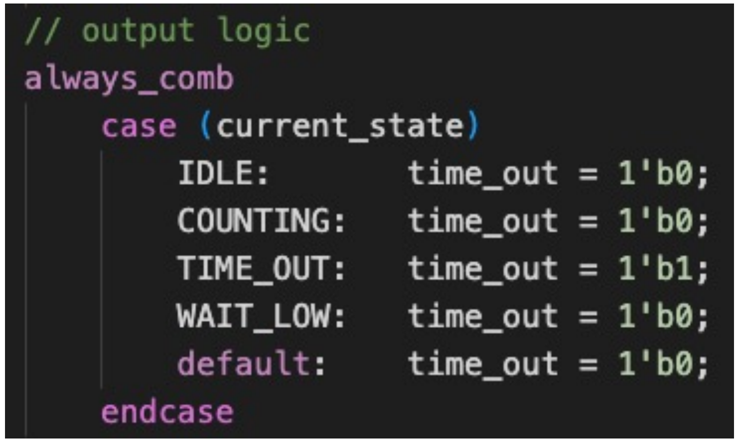
```
module delay #(
    parameter WIDTH = 7    // no of bits in delay counter
)(
    input   logic               clk,        // clock signal
    input   logic               rst,        // reset signal
    input   logic               trigger,    // trigger input signal
    input   logic [WIDTH-1:0]   k,          // no of clock cycle delay
    output  logic               time_out    // output pulse signal
);
    // Declare counter
    logic [WIDTH-1:0]   count = {WIDTH{1'b0}};  // internal counter

    // Define our states
    typedef enum {IDLE, COUNTING, TIME_OUT, WAIT_LOW}  my_state;
    my_state current_state, next_state;
```

Example 4: delay module (2)



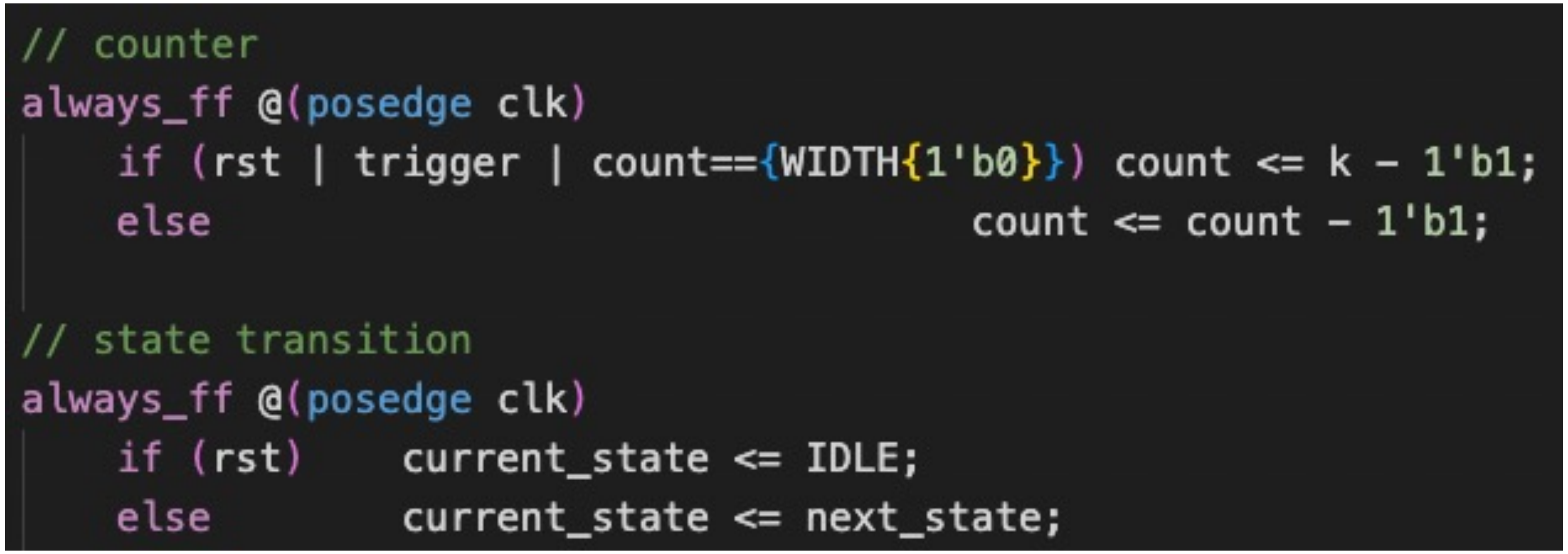


```
// next state logic
always_comb
    case (current_state)
        IDLE:       if (trigger==1'b1)  next_state = COUNTING;
                    else     next_state = current_state;
        COUNTING:   if (count=={WIDTH{1'b0}}) next_state = TIME_OUT;
                    else     next_state = current_state;
        TIME_OUT:   if (trigger==1'b1)  next_state = WAIT_LOW;
                    else     next_state = IDLE;
        WAIT_LOW:   if (trigger==1'b0)  next_state = IDLE;
                    else     next_state = current_state;
        default: next_state = IDLE;
    endcase
```



```
// output logic
always_comb
    case (current_state)
        IDLE:       time_out = 1'b0;
        COUNTING:   time_out = 1'b0;
        TIME_OUT:   time_out = 1'b1;
        WAIT_LOW:   time_out = 1'b0;
        default:    time_out = 1'b0;
    endcase
```

Example 4: delay module (3)



```
// counter
always_ff @(posedge clk)
    if (rst | trigger | count=={WIDTH{1'b0}}) count <= k - 1'b1;
    else                                      count <= count - 1'b1;

// state transition
always_ff @(posedge clk)
    if (rst)    current_state <= IDLE;
    else        current_state <= next_state;
```