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

In this section of the course, we will consider the design and specification of finite state machine (FSM). FSM is one of the most important topics in digital design. It provides a formal methodology for a designer to translate specification of a digital **control specification** to actual circuits.

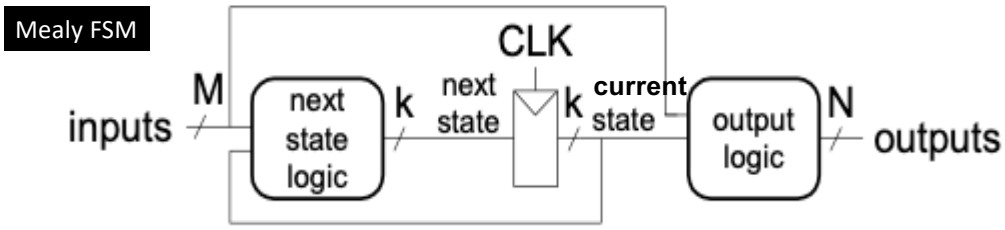
Lecture Objectives

- 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

This is a list of learning outcomes for this lecture. Most of the learning will be through a number of examples.

Synchronous State Machines

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



  - The ***current state*** is defined by the register contents
  - Register has *k* flipflops ⇒ 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.

Here is a simplified generic diagram of a finite (or synchronous) state machine (FSM or SSM). A set of D-flipflips are used to store the current state value. The current state together with external inputs are fed to a combinational logic circuit to evaluate two things: the **next state** and the **current outputs**.

With an n-bit register and using binary state encoding (i.e. coding states as binary numbers), such machine can have a maximum of 2^n states.

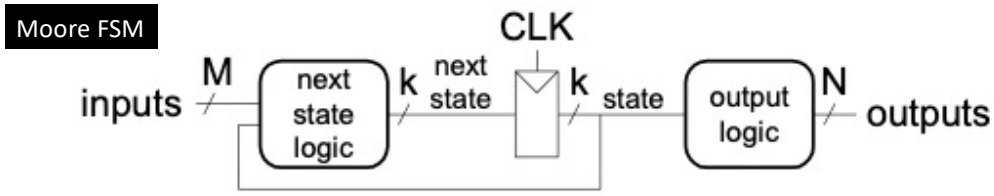
This is a synchronous state machine because the transition to the next state is synchronous with the rising edge of the clock signal.

The output signals, on the other hand is derived from both the current state and the current input. This property makes this a "Mealy" machine. Beware, the output signals are only synchronous to the clock IF all inputs are also synchronous.

There are two basic rules in designing a FSM that operates reliably:

1. Do not put logic in front of the clock signal. Doing so is likely to cause timing issues when the FSM is used in conjunction with the rest of the system.
2. Avoid using asynchronous SET or RESET signals unless absolutely required by the specification. Doing so would make the rest of the system NOT synchronous to the CLOCK signal.

Simple 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

To summarize, a FSM has three parts:

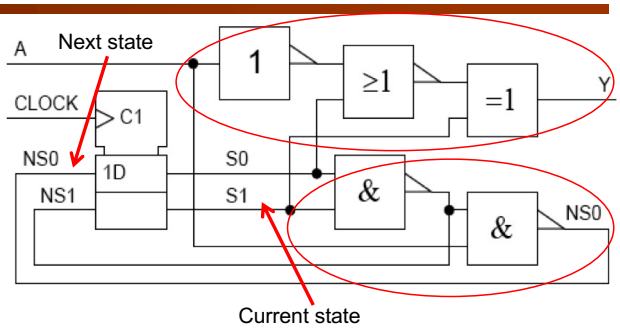
1. A set of registers to store the **current state** value.
2. Combinational logic to determine what the **next state value** should be, i.e. the state transition of the FSM.
3. Combinational logic to compute **the output signals.** These signals can be derived **ONLY** from the **current state** value. In which case, this is called "**Moore**" FSM. Alternatively, the output signals can be derived from both the **current state** and the **current input.** This is called a "**Mealy**" FSM, and the output signals could change in the middle of a clock cycle if input signals are NOT synchronized with the CLOCK.

.

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 /



NS1,NS0/Y		
S1,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

Shown here is a simple FSM in details. The upper group of gates are used to compute the output signal Y. The lower group of gates are used to work out the next state values NS0 and NS1.

We will now analyse how this circuit works. One powerful tool that we can use is to the state transition table. It is similar to the truth table used for combinational circuit, but is used to show the function of the FSM.

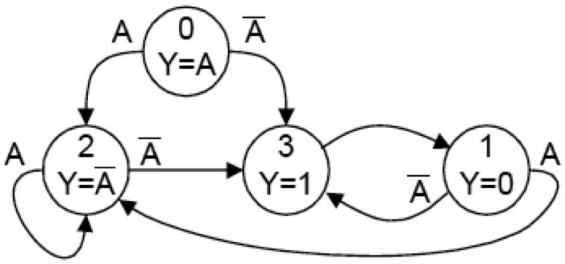
Each row in this table represents one state. Since this FSM has 2 state bits, there are 4 possible states.

There is one column devoted to each input combination. In this case, there is only one input A. There would be four columns if there were two inputs.

The contents of the table shows the next state transition, followed by the output signal(s) during the current state. A '/' character is used to separate the two.

Drawing the State Diagram

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



NS1,NS0/Y

S1,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

Another very powerful tool to show the function of a FSM is to use state diagram (one that uses “bubbles”). For clarity, let us split the state transition table into two tables: one for next state NS1:0, and another for the output signal Y.

We now draw a bubble for each state and label this with the state name (which happens in this case to be the same as the state value). Transition arrows are draw between the states with a Boolean expression as a label to indicate the condition required for the transition to occur ON THE ACTIVE CLOCK EDGE (positive edge in this case). The transitions are derived direclty from the next state table. Consider state 0, on rising edge of CLOCK, if A=0, go to state 3, else if A=1, go to state 2.

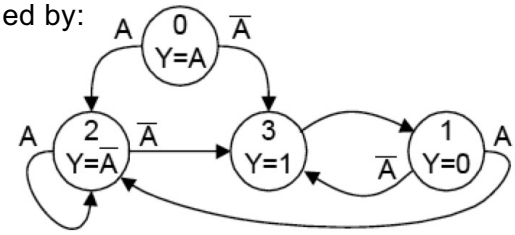
Inside the bubble, we now indicate the value of Y as another Boolean expression.

In this example, we perform analysis of a circuit designed by someone else. Therefore we derive the transition table from the circuit, then the state diagram from the state transition table.

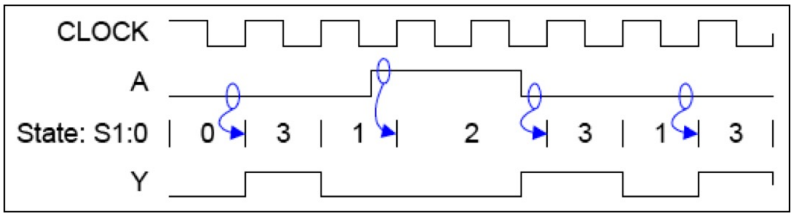
When we are designing a FSM from a specification, we usual do this the other way round, i.e. design the state diagram from the specification, then draw up the state transition table as required and derive the circuit from that.

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***)

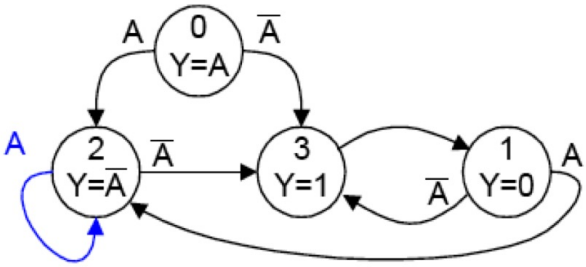
It is important to note that the behaviour of a FSM is determined by the initial state. Given the state diagram and the initial state (assumed here to be state 0), and waveform of the input A, we can easily trace the subsequent states S1:0 and the output Y.

For our course, we generally use a reset signal to force the FSM to go to an initial state.

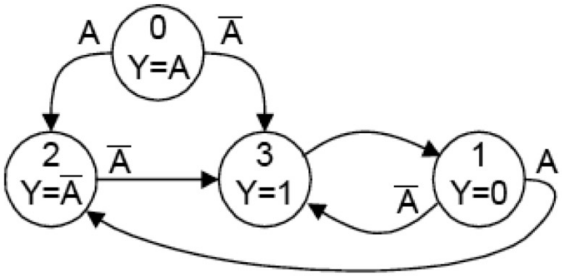
The FSM here is a **Mealy** machine because the output Y inside state 0 and 2 are Boolean expressions. If A changes in the middle of a clock cycle, the output Y will change immediately. So the output is NOT dependent on the state of the machine alone.

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

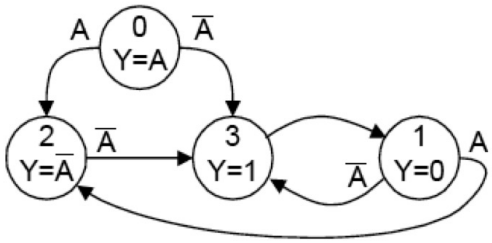


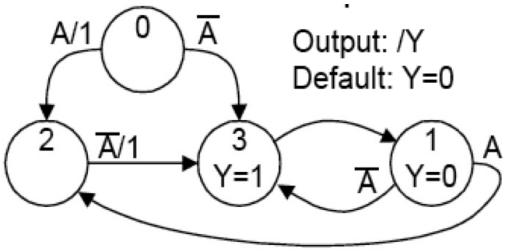
In order to make the state diagram less cluttered, you can omit the self transition arrows. Therefore the rule is that a state machine stays in its current state unless the conditions of an exiting arrow is satisfied.

In this example, we stay in state 2 until A = 0 on the rising edge of CLOCK. Then we go to state 3.

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





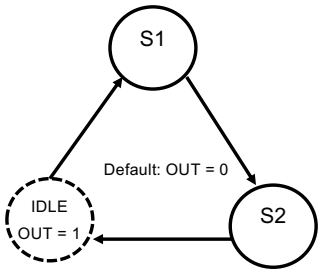
- Outputs written on an arrow apply to the state ***emitting*** the arrow.
- Outputs still apply for the entire time spent in a state
- This does not affect the Moore/Mealy distinction
- This is a notation change only

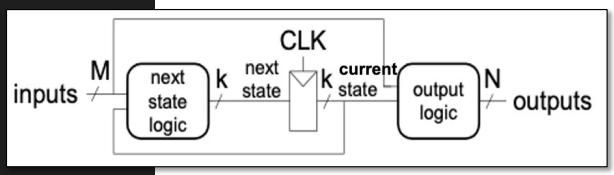
Instead of specifying outputs inside the state bubble, it is also possible to specify outputs on the transition arrow. There are a few rules that you must follow:

1. For each state, you must specify the output either inside the bubble or on EVERY emitted arrow from the state.
2. You can mix the two conventions in a state diagram, but you must use only one method for each (and not mixing them).
3. If you use self transition, as in state 2 here, you must declare the default values for each outputs.
4. Output written on an arrow always applies to the state **EMITTING** the arrow (i.e. source not destination).

Example 1: Divide by 3 FSM (Moore)

```systemverilog
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
```





PYKC 19 Nov 2024 EE2 Circuits & Systems Lecture 11 Slide 10

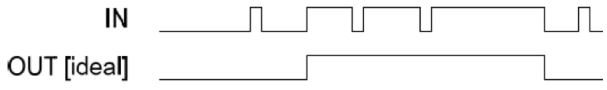
Here is a very simple three states FSM. The intial state is the IDLE state (here shown in dotted lines). The purpose of this machine is to divide clock signal by three – out goes high for one cycle every three.

To specify this FSM in SystemVerilog, we divide the SV file into five parts:

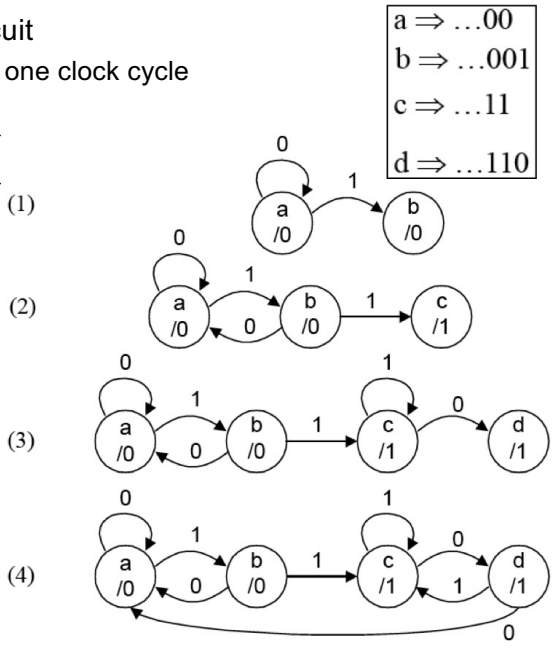
1. Interface declaration – define the input and output signals.
2. State enumeration – specify an enumated type, here we call it my_state, and give all states a state name (e.g. IDLE, S1, … etc), and then declare two state variables: current_state, and next_state.
3. State registers – specify the registers that advance the FSM from state to state.
4. State transition logic – specify the combinational logic that computes the next state values.
5. Output logic – specify the combinational logic that computes the output signals of the FSM.

Example 2: Design a Noise Pulse Eliminator (1)

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



- 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.



PYKC 19 Nov 2024 EE2 Circuits & Systems Lecture 11 Slide 11

We will now consider the design of a FSM to do some defined function:

Design a circuit to eliminate noise pulses. A noise pulse (high or low) is one that lasts only for one clock cycle. Therefore, in the waveform shown above, IN goes from low to high, but included with some high and some low noise pulses. The goal is to clean this up and produce ideally the output OUT as shown.

Here we label the states with letters **a, b, c** …. Starting with a when IN = 0, and we are waiting for IN -> 1. Then we transit to **b**. However, this could be a noise pulse. Therefore we wait for IN to stay as 1 for another close cycle before transiting to **c** and output a 1. If IN goes back to zero after one cycle, we go to **a**, and continue to output a 0.

Similar for state **c**, where we have detect a true 1 for IN. If IN -> 0, we go to **d**, but wait for another cycle for IN staying in 0, before transiting back to state **a**.

Therefore this FSM has four states. Note that in reality, OUT is delayed by ONE clock cycle. There is in fact no way around this – we have to wait for two cycles of IN=0 or IN=1 before deciding on the value of OUT.

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

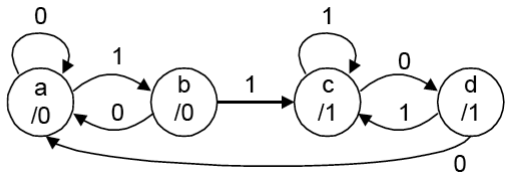
This example illustrates how each state represents a particular history that needs to be recorded.

This slide reiterates how we arrive at the state diagram and what each state means.

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




```
// 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
```

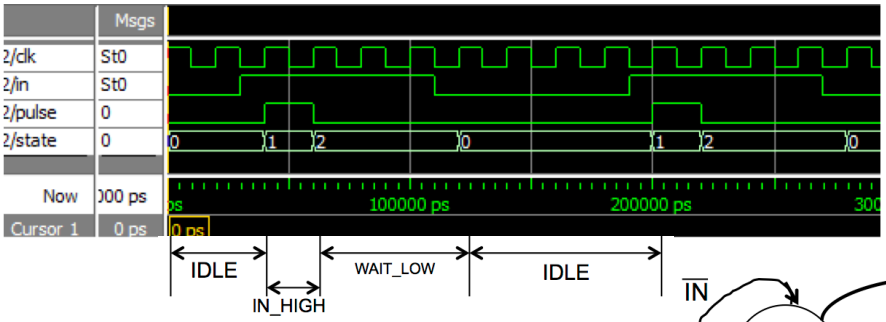

Instead of manually designing a state machine, we usually rely on SystemVerilog specification and synthesis CAD tools.

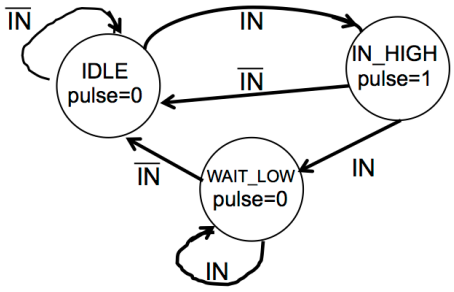
Here we use an EXPLICIT reset signal **rst** to put the state machine in a known state.

One importance lesson here is that in the transition logic specification, we normally use the **always_comb + case** statements to define the state transition logic. Further, if you want to stay in the current state, you just assign **current_state** to **next_state** as shown here.

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).

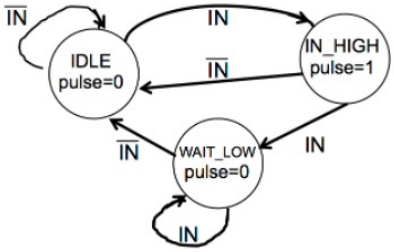
Let us now consider another example, which will appear in the Lab Experiment later. You are required to design a pulse generator circuit that, on the positive edge of the input **IN**, a pulse lasting for one clock period is produced.

The state diagram for this circuit is shown here. There has to be three state: IDLE (waiting for IN to go high), the IN_HIGH state when a rising edge is detected for IN, and WAIT_LOW state, where we wait for the IN to go low again.

Shown here is the timing diagram for this design. This module is very useful. It effective detects a rising edge of a signal, and then produces a pulse at the output which is one clock cycle in width.

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;
```

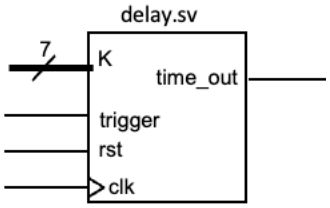
```
// next state logic
always_comb
    case (current_state)
        IDLE:       if (in==1'b1)   next_state = IN_HIGH;
                    else            next_state = current_state;
        IN_HIGH:    if (in==1'b1)   next_state = WAIT_LOW;
                    else            next_state = IDLE;
   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


This FSM has three states: IDEL, IN_HIGH and WAIT_LOW. Mapping the state diagram to SystemVerilog follows the same pattern as the previous example

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.



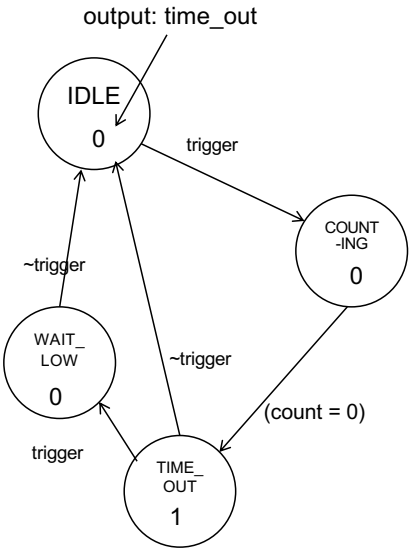

```
module delay #(
    parameter WIDTH = 7    // no of bits in delay counter
)(
    input   logic               clk,        // clock signal
    input   logic               rst,        // reset signal
    input   logic               trigger,    // trigger input signal
    input   logic [WIDTH-1:0]   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;
```


Finally, here is a very useful module that uses a four-state FSM and a counter. It is the combination of the previous example with a counter embedded inside the FSM.

The module detects a rising edge on the **trigger** input, internally counts **K** clock cycles, then outputs a pulse on **time_out**. This effectively delay the trigger rising edge by **K** clock cycles.

Shown here are the interface and state declaration for the module. Note that we need to also declare the **count** internal logic, which will eventually synthesized as a counter circuit. Note also the way that this counter is initialized to zero without using reset.

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
```


Here is the implementation. Note how the counter value is used in the state transition logic.

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;
```


Finally, we need the sequential circuit specifications for state transistion, and for the counter logic itself.

The two **always_ff** and **always_comb** blocks can be specified in any order. Remember, SystemVerilog is NOT C++ or normal software language. The statements aren blocks are specifications for hardware and they run in PARALLEL at the same time.

As a result, when specifying these separate blocks of hardware, be aware that you may drive the same signal in different “blocks” simultaneously. Such contentions should be picked up by Verilator and a warning or error will be generated.