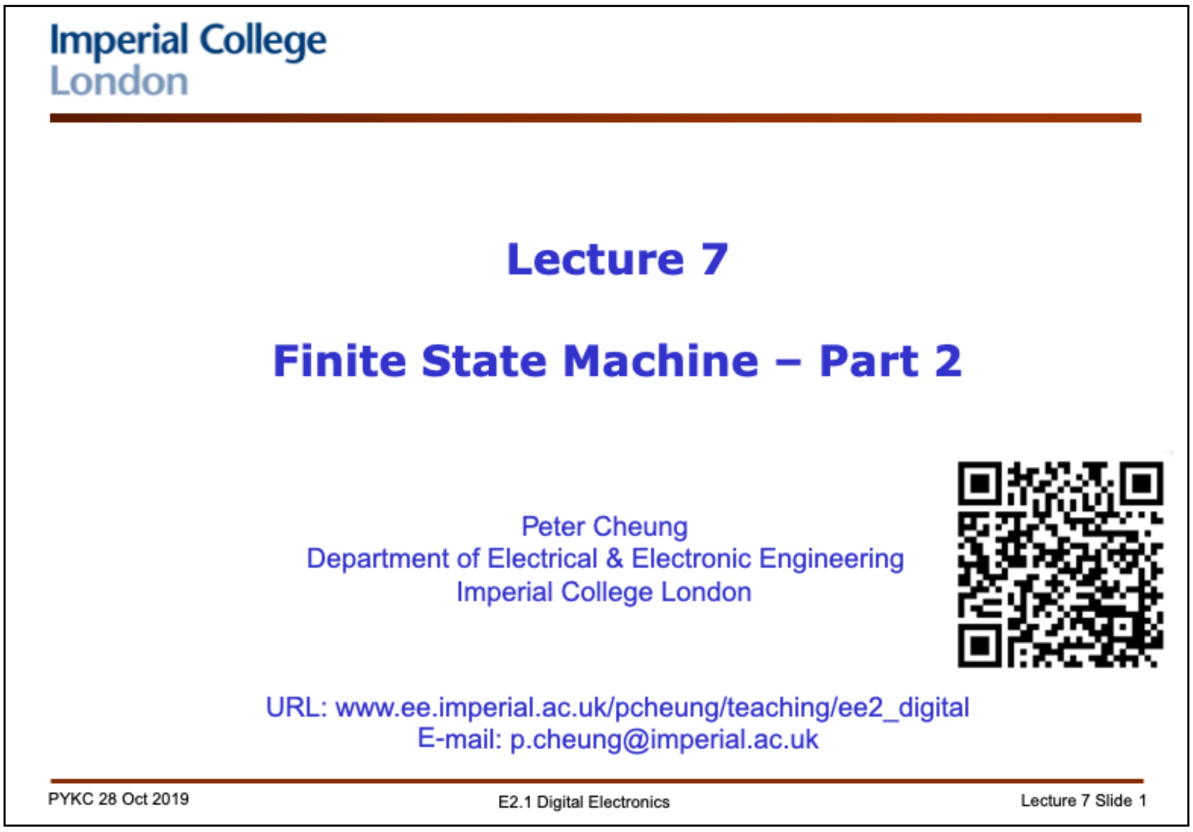


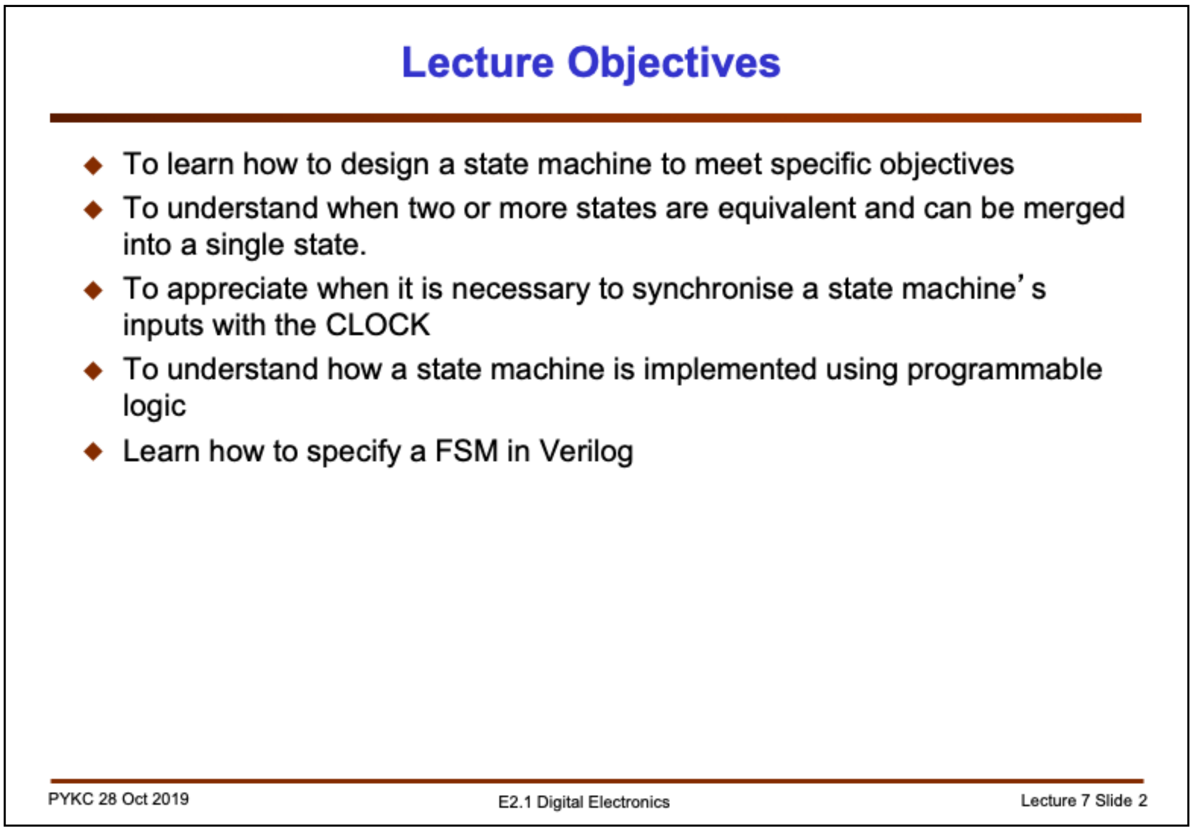
Imperial College London

Lecture 7

Finite State Machine – Part 2

Peter Cheung
Department of Electrical & Electronic Engineering
Imperial College London

URL: www.ee.imperial.ac.uk/pcheung/teaching/ee2_digital
E-mail: p.cheung@imperial.ac.uk

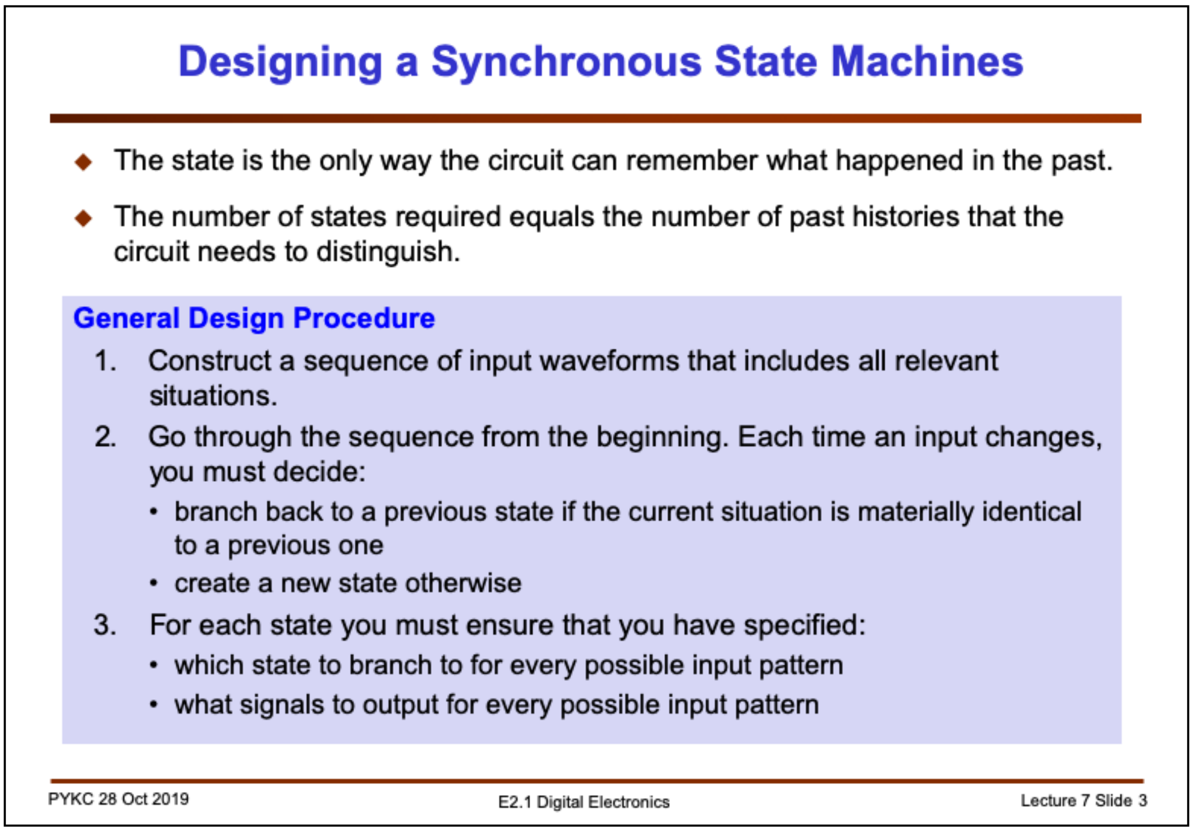


Lecture Objectives

- To learn how to design a state machine to meet specific objectives
- To understand when two or more states are equivalent and can be merged into a single state.
- To appreciate when it is necessary to synchronise a state machine's inputs with the CLOCK
- To understand how a state machine is implemented using programmable logic
- Learn how to specify a FSM in Verilog

PYKC 28 Oct 2019 E2.1 Digital Electronics Lecture 7 Slide 2

In the previous lecture, we examined how to analyse a FSM using state table, state diagram and waveforms. In this lecture we will learn how to design a fininte state machine in order to produce the desired output signals for control purposes.



Designing a Synchronous State Machines

- The state is the only way the circuit can remember what happened in the past.
- The number of states required equals the number of past histories that the circuit needs to distinguish.

General Design Procedure

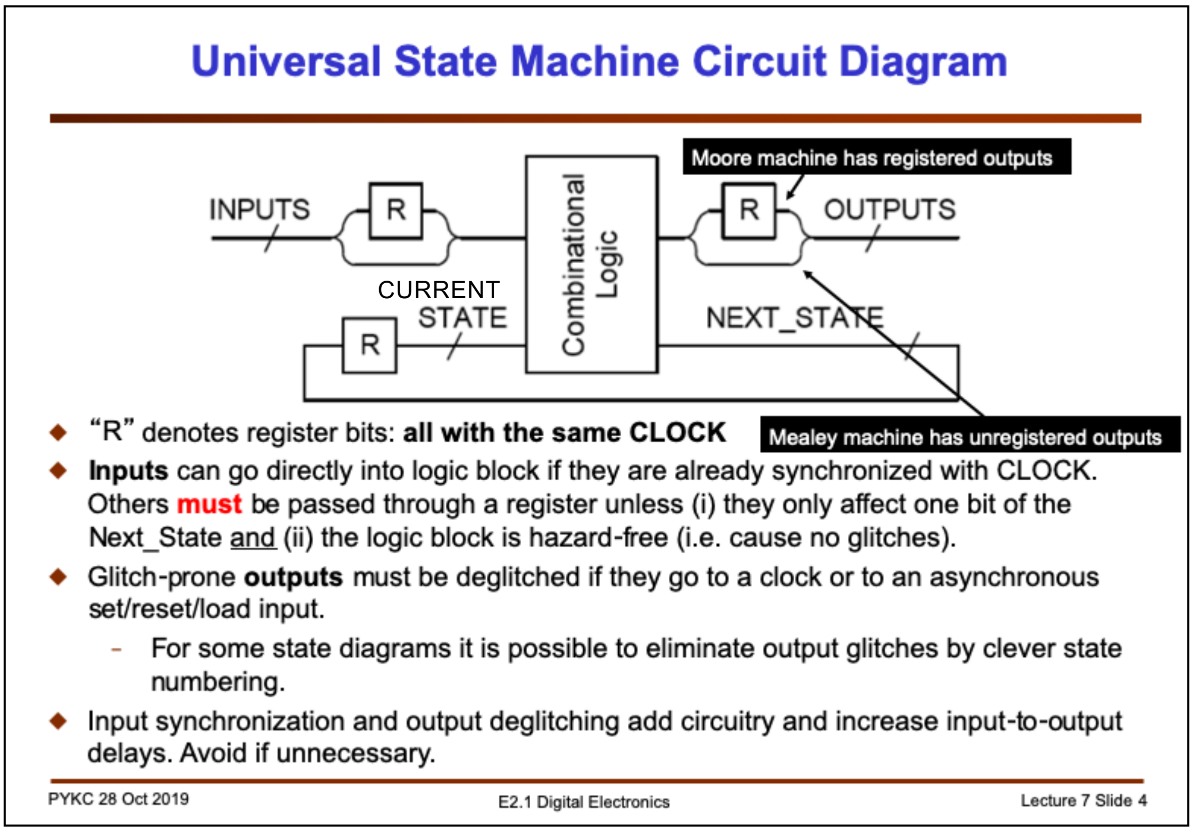
1. Construct a sequence of input waveforms that includes all relevant situations.
2. Go through the sequence from the beginning. Each time an input changes, you must decide:
   - branch back to a previous state if the current situation is materially identical to a previous one
   - create a new state otherwise
3. For each state you must ensure that you have specified:
   - which state to branch to for every possible input pattern
   - what signals to output for every possible input pattern

PYKC 28 Oct 2019 E2.1 Digital Electronics Lecture 7 Slide 3

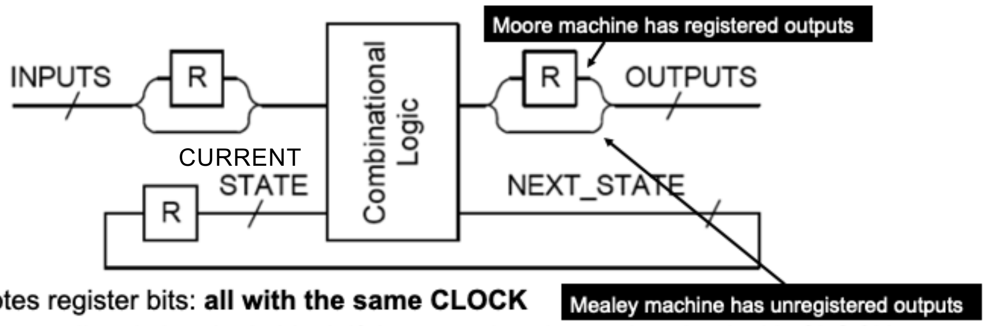
There are various ways that we can approach the FSM design problem. Here we will assume that we start with input waveforms that includes all the relevant situations that the circuit would need to go through. Another (better approach) is to consider the FSM in terms of the algorithm that it implements – in a way that is more similar to thinking about software.

Construct a state diagram to capture what you want to design, making sure that all transitions and outputs are as intended.

Let us consider an example.



Universal State Machine Circuit Diagram


Moore machine has registered outputs
INPUTS
R
Combinational Logic
R
OUTPUTS
CURRENT STATE
R
NEXT_STATE
Mealey machine has unregistered outputs

- "R" denotes register bits: **all with the same CLOCK**
- **Inputs** can go directly into logic block if they are already synchronized with CLOCK. Others **must** be passed through a register unless (i) they only affect one bit of the Next_State and (ii) the logic block is hazard-free (i.e. cause no glitches).
- Glitch-prone **outputs** must be deglitched if they go to a clock or to an asynchronous set/reset/load input.
  - For some state diagrams it is possible to eliminate output glitches by clever state numbering.
- Input synchronization and output deglitching add circuitry and increase input-to-output delays. Avoid if unnecessary.

PYKC 28 Oct 2019 E2.1 Digital Electronics Lecture 7 Slide 4

Here is a diagram showing a universal state machine. Not shown here is the clock signal on all the registers.

Moore FSM

We generally use Moore FSM in our design on this course. Remember in Moore machines, the output does not change in the middle of a clock cycle. Therefore the output of the machine is determined by the current state of the FSM.

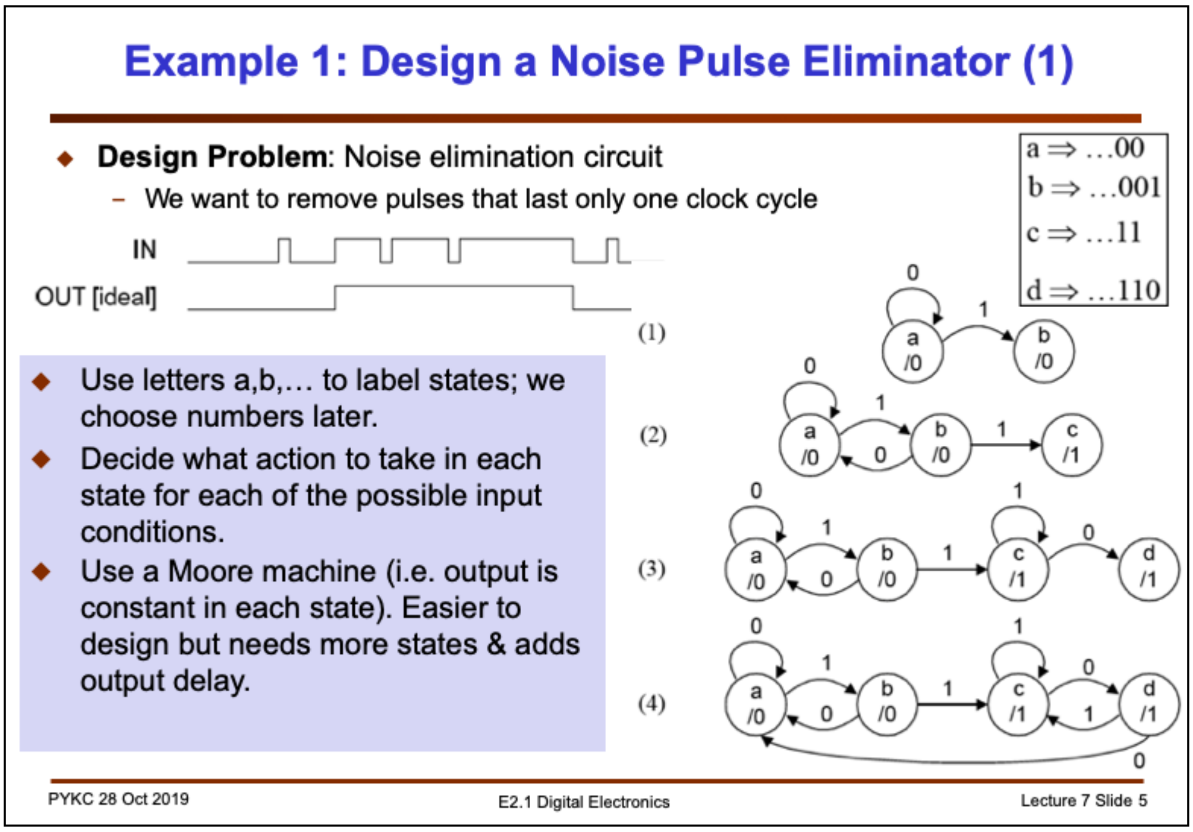
For Moore machines, the output is driven by a D-FF as shown in the slide.

Mealey FSM

For Mealey machines, the output is dependent on both the current state of the FSM and the input signals. Therefore a Mealey machine output can change in the middle of a clock cycle.

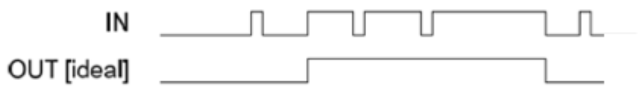
We prefer to use Moore FSM in our course because its output do not have glitches. In contract, a Mealy FSM output is produced directly from the combinational circuits without going through a D-FF. Therefore it may contain glitches.

One disadvantage of a Moore FSM is that the output may have one cycle delay (because of the output D-FF).



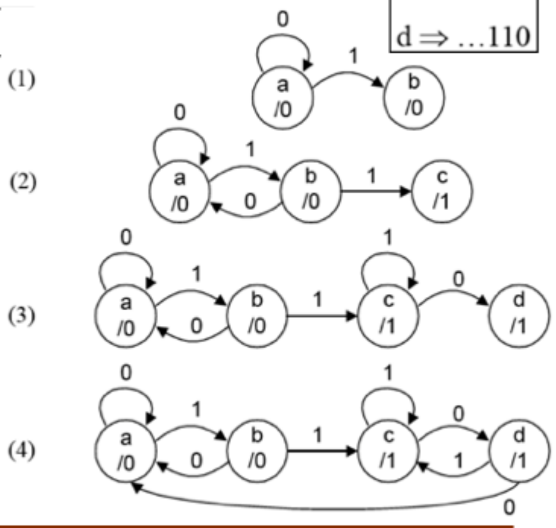
Example 1: Design a Noise Pulse Eliminator (1)

- **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.

a ⇒ …00
b ⇒ …001
c ⇒ …11
d ⇒ …110



PYKC 28 Oct 2019 E2.1 Digital Electronics Lecture 7 Slide 5

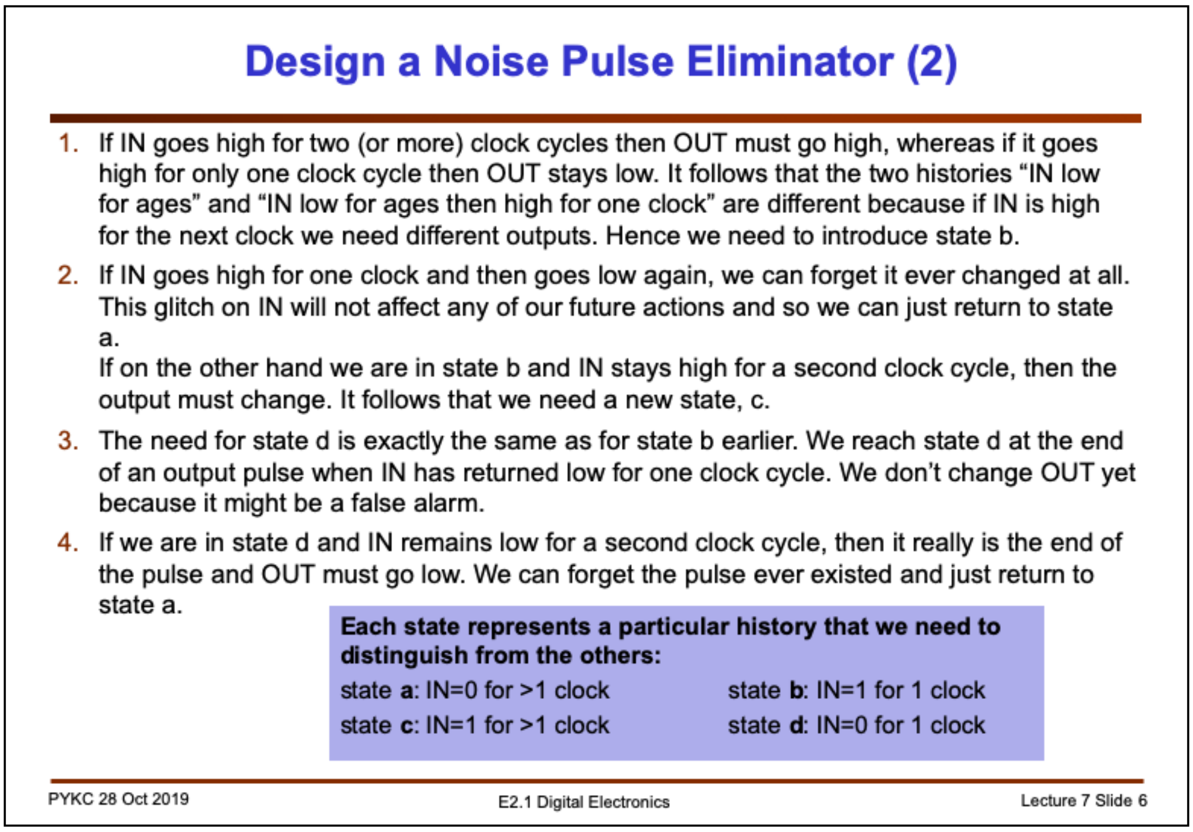
We will now consider the design of a FSM to do some defined function:

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.

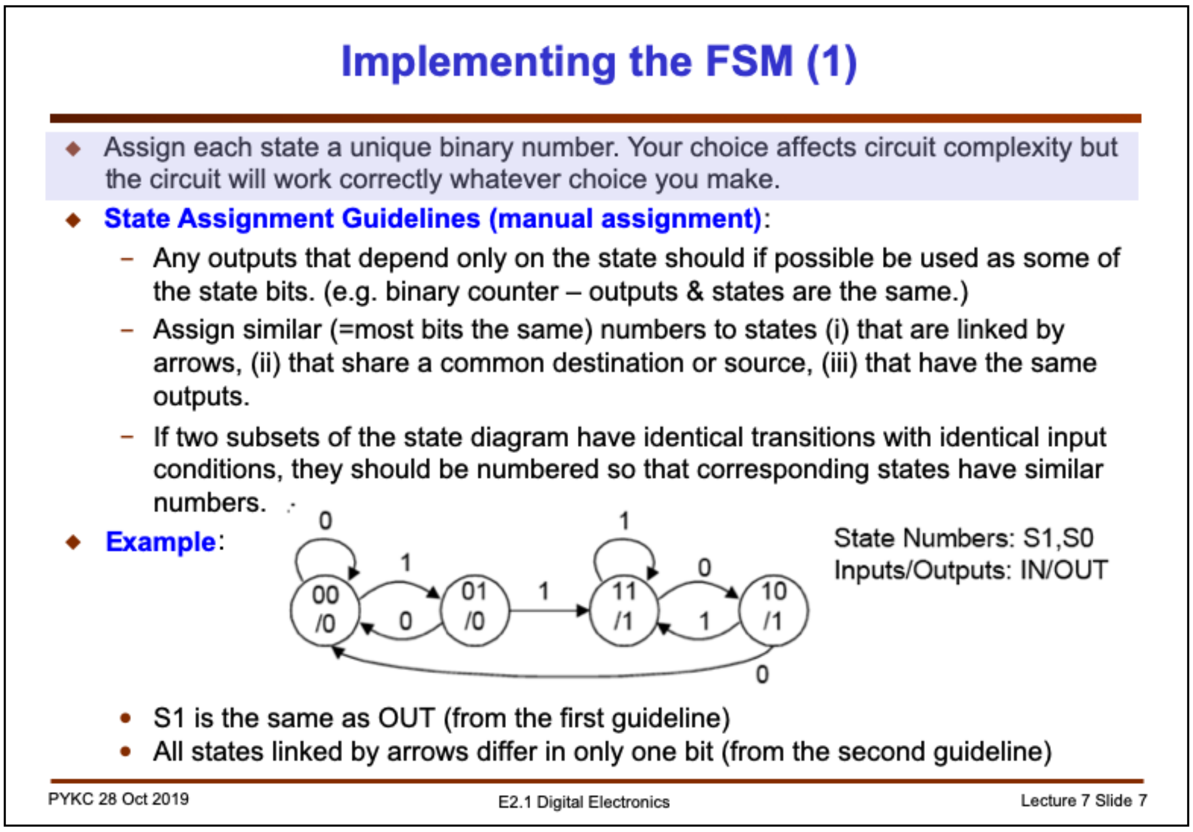
Each state represents a particular history that we need to distinguish from the others:

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

PYKC 28 Oct 2019 E2.1 Digital Electronics Lecture 7 Slide 6

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

This slide reiterates who we arrives at the state diagram and what each state means.



Implementing the FSM (1)

- Assign each state a unique binary number. Your choice affects circuit complexity but the circuit will work correctly whatever choice you make.
- **State Assignment Guidelines (manual assignment)**:
  - Any outputs that depend only on the state should if possible be used as some of the state bits. (e.g. binary counter – outputs & states are the same.)
  - Assign similar (=most bits the same) numbers to states (i) that are linked by arrows, (ii) that share a common destination or source, (iii) that have the same outputs.
  - If two subsets of the state diagram have identical transitions with identical input conditions, they should be numbered so that corresponding states have similar numbers.
- **Example**:

$$\begin{tikzcd} 00/0 \arrow[loop above, "0"] \arrow[r, bend left, "1"] & 01/0 \arrow[l, bend left, "0"] \arrow[r, "1"] & 11/1 \arrow[loop above, "1"] \arrow[r, bend left, "0"] & 10/1 \arrow[l, bend left, "1"] \arrow[lll, bend left, "0"] \end{tikzcd}$$

State Numbers: S1,S0
Inputs/Outputs: IN/OUT

- S1 is the same as OUT (from the first guideline)
- All states linked by arrows differ in only one bit (from the second guideline)

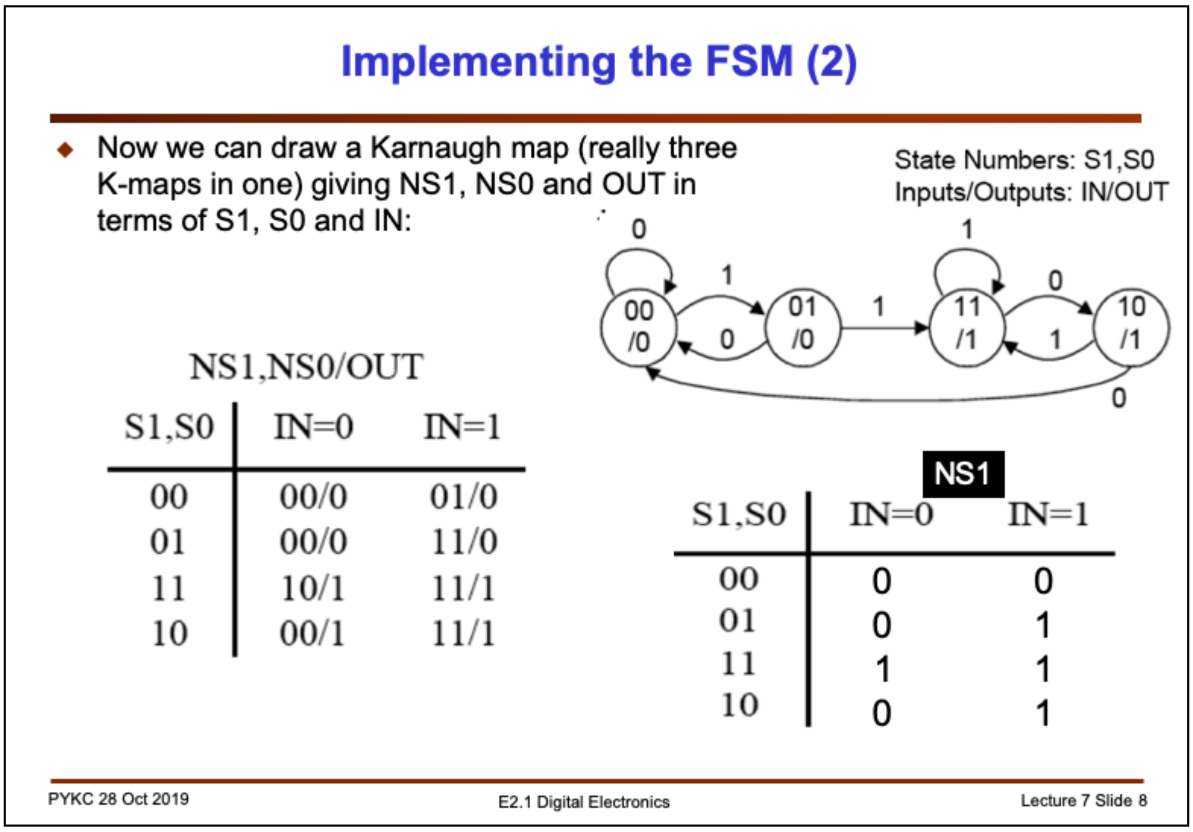
PYKC 28 Oct 2019 E2.1 Digital Electronics Lecture 7 Slide 7

Before mapping the state diagram to hardware, we need to perform **state encoding** – giving each state a unique binary value. For the noise eliminator, we have four states and therefore if we use binary encoding, we need two state bits to encode all four states. Here we assign values S1:S0 of 00, 01, 11 and 10 to states a, b, c and d respectively.

Note that you could assign ANY binary number to any state – and the implemented FSM will work. However, different state encoding will result in different implementations, affecting the complexity of the digital logic.

In the assignment above, we deliberating make S1 the same as OUT – this simplifies the output logic.

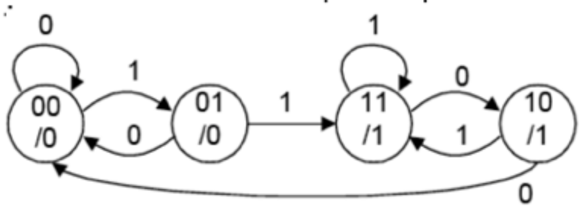
We deliberately make all states linked by arrows only having one bit changing (hence 01 -> 11). This tends to simply the transition logic and reduce glitches.



Implementing the FSM (2)

- Now we can draw a Karnaugh map (really three K-maps in one) giving NS1, NS0 and OUT in terms of S1, S0 and IN:

State Numbers: S1,S0
Inputs/Outputs: IN/OUT



NS1,NS0/OUT

S1,S0	IN=0	IN=1
00	00/0	01/0
01	00/0	11/0
11	10/1	11/1
10	00/1	11/1

NS1

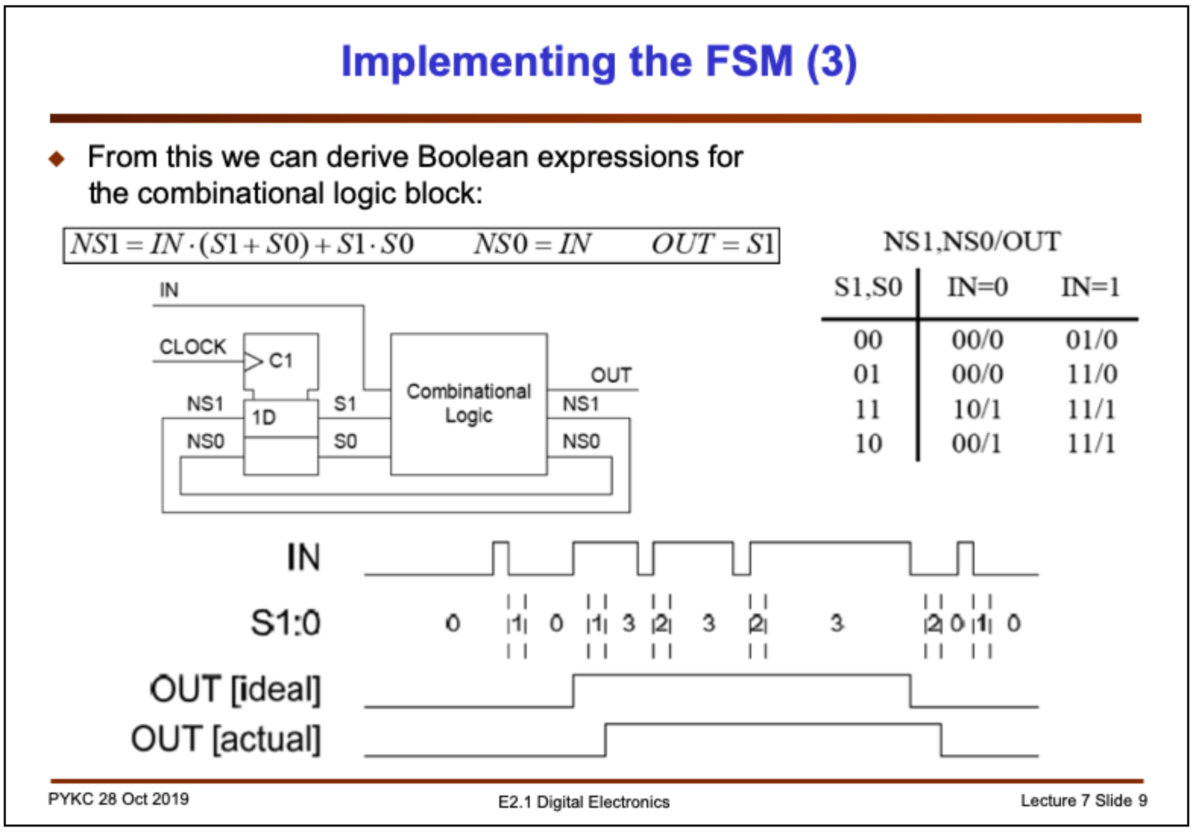
S1,S0	IN=0	IN=1
00	0	0
01	0	1
11	1	1
10	0	1

PYKC 28 Oct 2019 E2.1 Digital Electronics Lecture 7 Slide 8

Once we have completed state encoding, we can fill in the state transition table with binary values for the current state values S1:0, the next state values NS1:0 and the output OUT. This is shown on the left.

If you were to design this FSM by hand, you would need to generate Boolean equations for the next state values NS1 and NS2, and the output signal OUT.

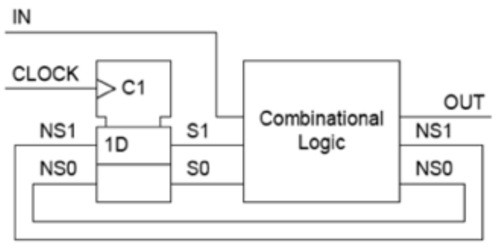
You may even use K-map to perform Boolean simplification.



Implementing the FSM (3)

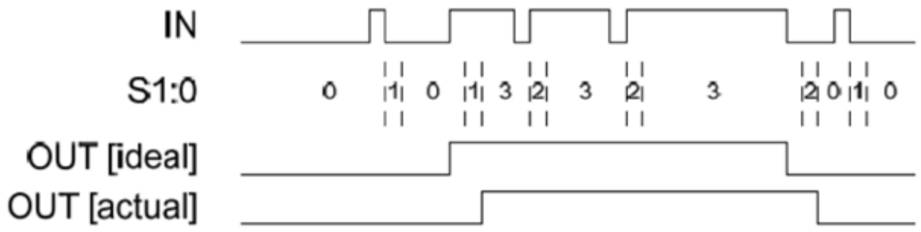
- From this we can derive Boolean expressions for the combinational logic block:

$$\boxed{NS1 = IN \cdot (S1 + S0) + S1 \cdot S0 \qquad NS0 = IN \qquad OUT = S1}$$



NS1,NS0/OUT

S1,S0	IN=0	IN=1
00	00/0	01/0
01	00/0	11/0
11	10/1	11/1
10	00/1	11/1

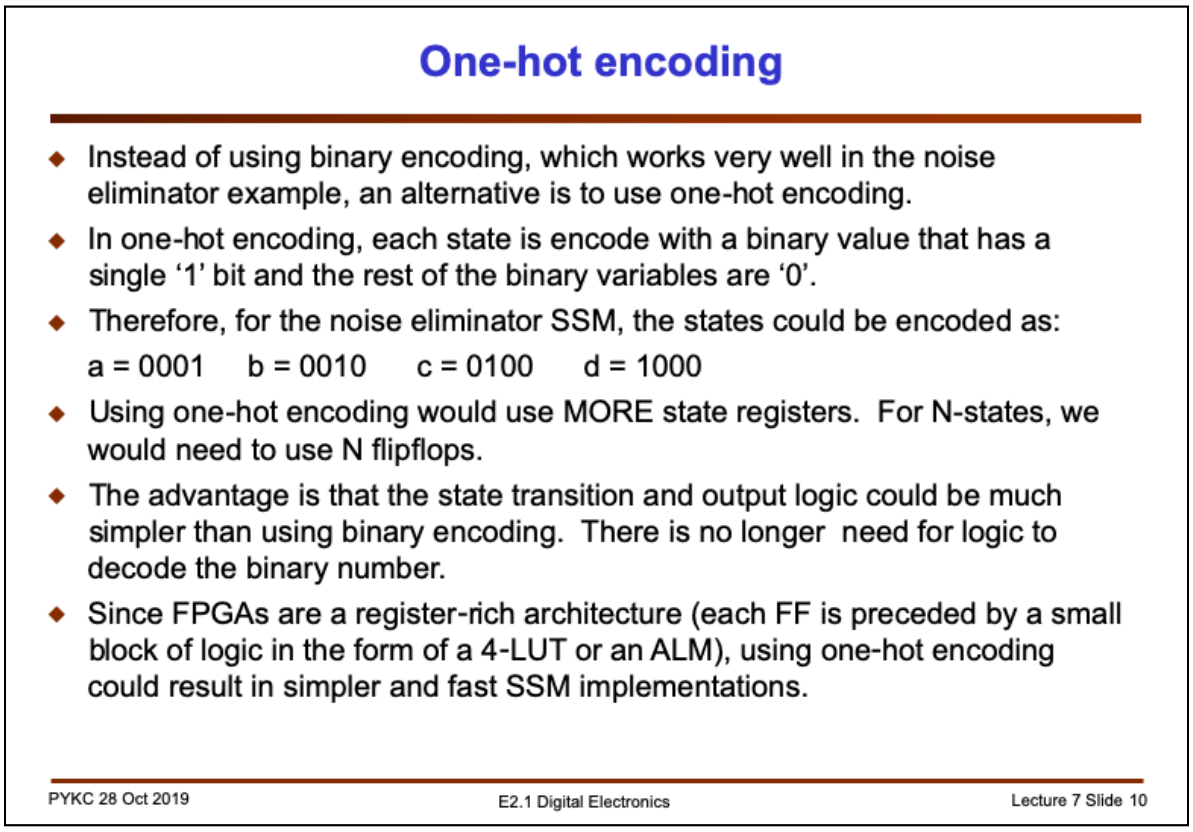


Now we can derive the Boolean express for NS1, NS0 and OUT in the usual way.

Since in general FPGA architecture, the logic elements can handle many inputs (at least 4 input signals) and is much more complex than a simple logic gate, implementing the Boolean equation for NS1 would only use ONE logic block.

Furthermore, each logic element also include its own registers. So implementing FSM in FPGAs is easy and efficient.

Note that the actual output waveforms shows that OUT has a one clock cycle delay.



One-hot encoding

- Instead of using binary encoding, which works very well in the noise eliminator example, an alternative is to use one-hot encoding.
- In one-hot encoding, each state is encode with a binary value that has a single '1' bit and the rest of the binary variables are '0'.
- Therefore, for the noise eliminator SSM, the states could be encoded as:

  a = 0001   b = 0010   c = 0100   d = 1000
- Using one-hot encoding would use MORE state registers. For N-states, we would need to use N flipflops.
- The advantage is that the state transition and output logic could be much simpler than using binary encoding. There is no longer need for logic to decode the binary number.
- Since FPGAs are a register-rich architecture (each FF is preceded by a small block of logic in the form of a 4-LUT or an ALM), using one-hot encoding could result in simpler and fast SSM implementations.

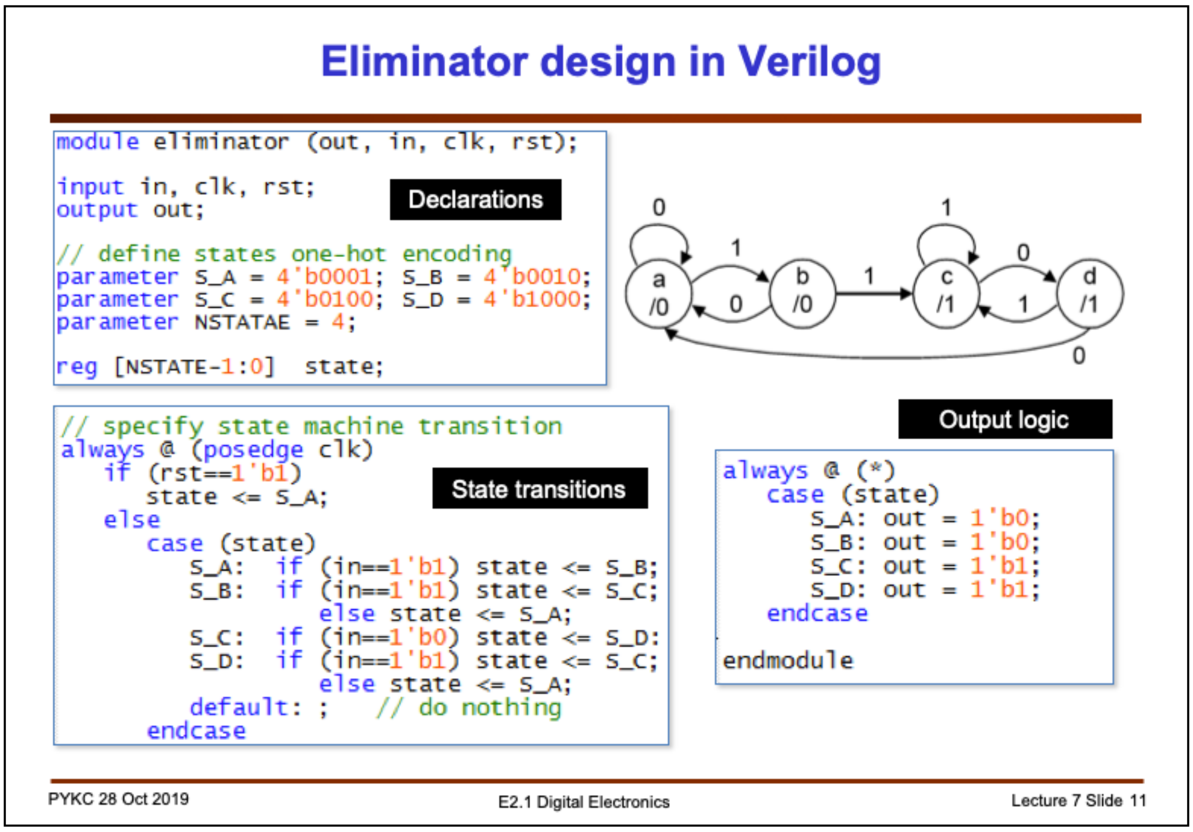
PYKC 28 Oct 2019 E2.1 Digital Electronics Lecture 7 Slide 10

In implementing FSMs using FPGAs, we often use a form of state encoding different from simple **binary encoding**. It is known as **one-hot encoding**.

With **one-hot encoding**, only one-bit in the state value is "hot" (i.e. set to '1'), and all the other bits are "cold" (i.e. reset to '0').

Using one-hot encoding matches the FPGA architecture well. Each FPGA logic element contains a combinational logic module and one or more registers. Therefore FPGA is a register-rich architecture.

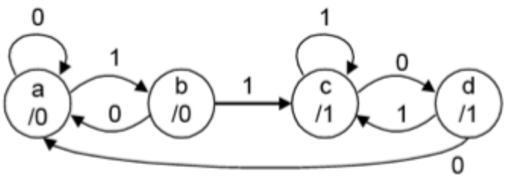
As an exercise, please implement the noise eliminator using one-hot encoding instead of binary encoding as we have in the previous slides by hand (i.e. without using CAD tools). You will appreciate why one-hot encoding is efficient with FPGAs.



Eliminator design in Verilog

Declarations

```
module eliminator (out, in, clk, rst);

input in, clk, rst;
output out;

// define states one-hot encoding
parameter S_A = 4'b0001; S_B = 4'b0010;
parameter S_C = 4'b0100; S_D = 4'b1000;
parameter NSTATAE = 4;

reg [NSTATE-1:0]  state;
```



State transitions

```
// specify state machine transition
always @ (posedge clk)
   if (rst==1'b1)
      state <= S_A;
   else
      case (state)
         S_A:  if (in==1'b1) state <= S_B;
         S_B:  if (in==1'b1) state <= S_C;
               else state <= S_A;
         S_C:  if (in==1'b0) state <= S_D:
         S_D:  if (in==1'b1) state <= S_C;
               else state <= S_A;
         default: ;    // do nothing
      endcase
```

Output logic

```
always @ (*)
   case (state)
      S_A: out = 1'b0;
      S_B: out = 1'b0;
      S_C: out = 1'b1;
      S_D: out = 1'b1;
   endcase

endmodule
```

PYKC 28 Oct 2019 E2.1 Digital Electronics Lecture 7 Slide 11

Instead of manually designing a state machine, we usually rely on Verilog specification and synthesis CAD tools such as Altera's Quartus software.

Here we use an EXPLICIT reset signal **rst** to put the state machine in a known state. We also use **one-hot** instead of binary encoding of the states. This is specified in the **parameter block**.

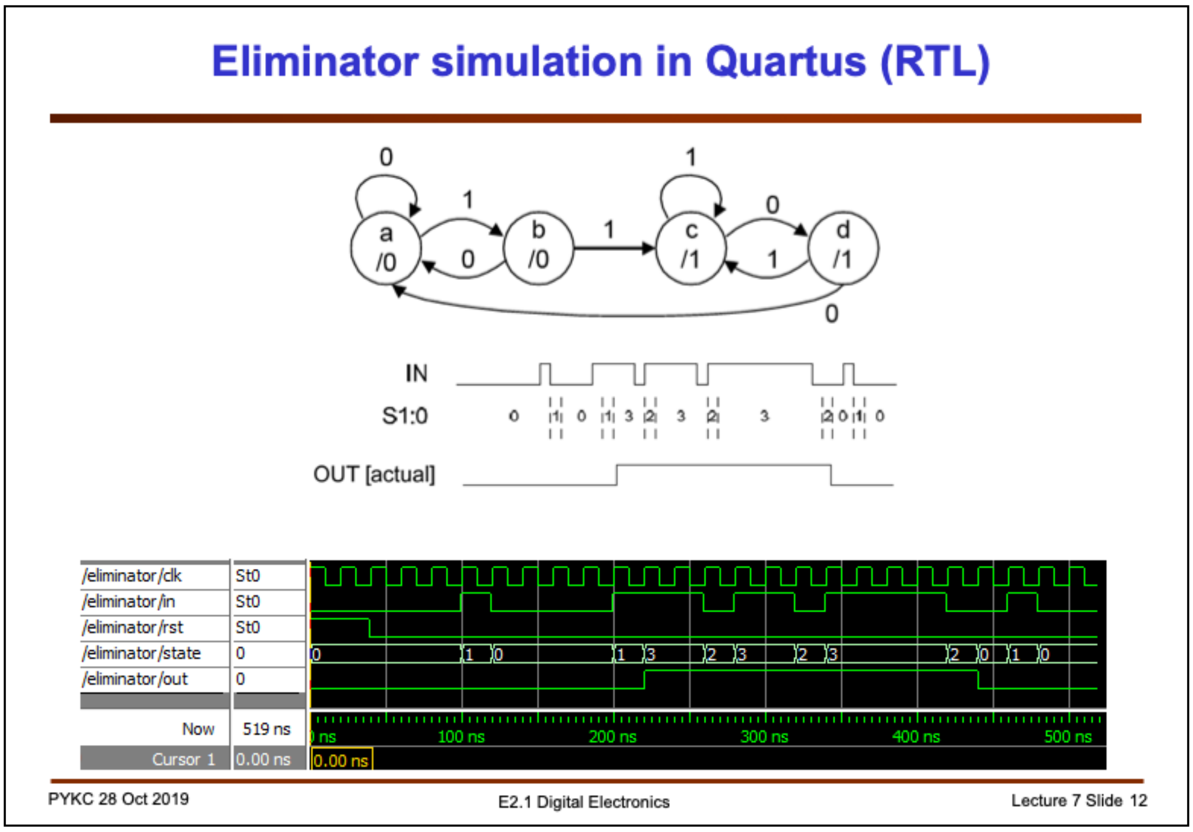
Using parameter block to give a name to each of the states has many benefits: the Verilog design is much easier to read; you can change state assignment values without needing to change any codes. In general, parameter block allows you to use **symbols** (names) to replace **numbers**. This makes the code easier to read and easier to maintain, and it is a good habit to get into.

The state variable declaration **reg [NSTATE-1:0]** is used here to show that you there are 4 states (S_A to S_D).

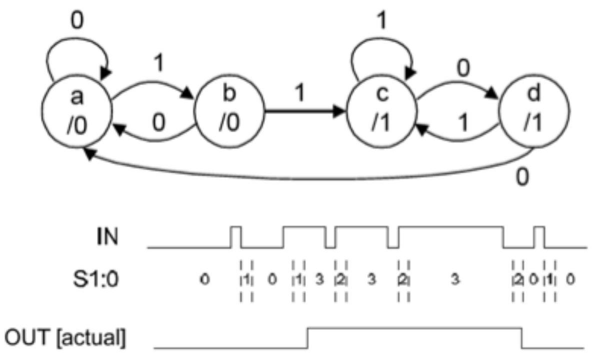
When specifying FSM in Verilog, you should following the following convention:

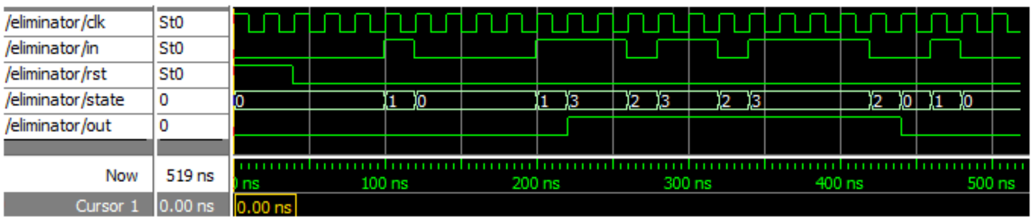
•Use always @ (posedge clk) block to specify the state transition. Note that we use the <= assignments (non-blocking) in this always block because you are responding to clock edges.

•Use a separate always @ (*) block to specify the the output logic. We use normal assignments (blocking) here because this is actually a combinational logic block, not sequential circuit.



Eliminator simulation in Quartus (RTL)

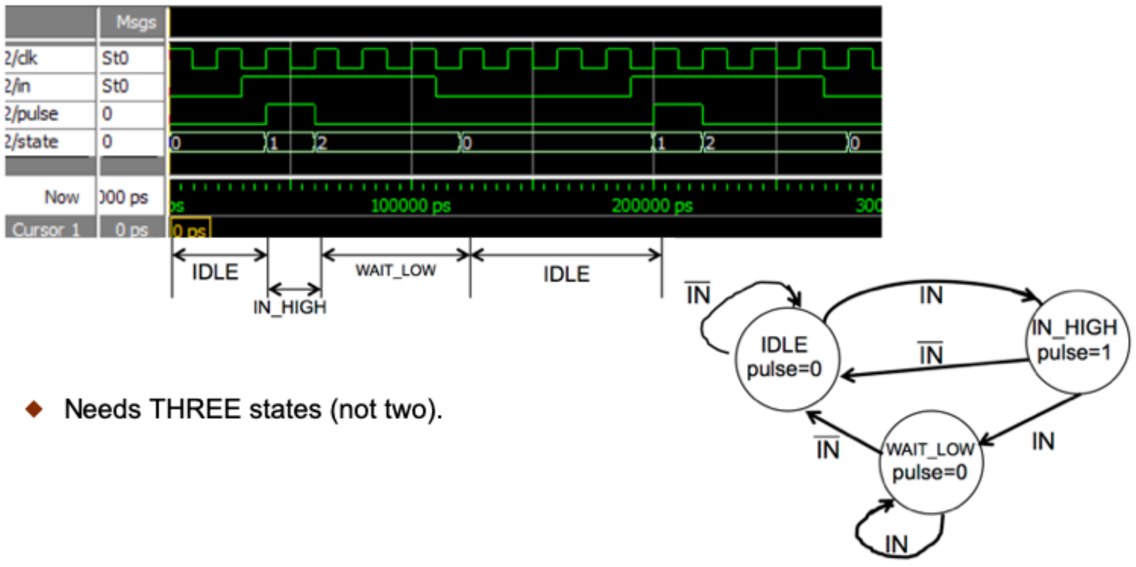




If you enter this Verilog description into Quartus and simulate the circuit, you will see the waveform as shown in this timing diagram as expected. Note that the actual waveform for out is NOT the ideal waveform, but is delayed by one clock cycle.

Example 2 – 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**.

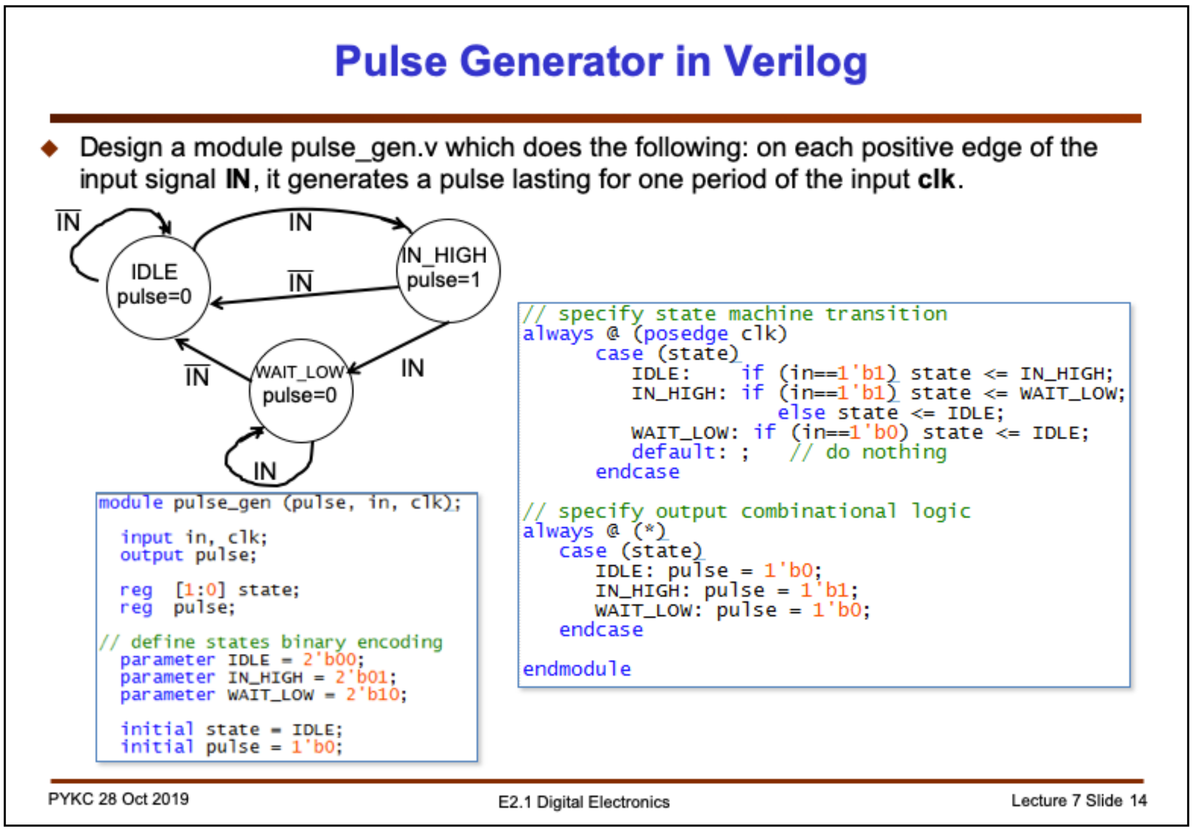


- Needs THREE states (not two).

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.

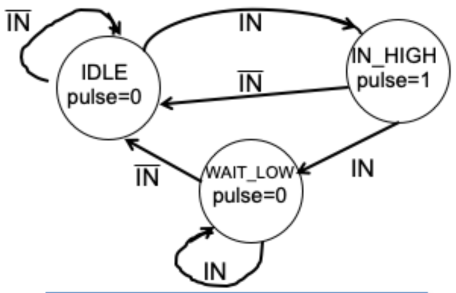
The state diagram for this circuit is shown here. There has to be three state: IDLE (waiting for IN to go high), the IN_HIGH state when a rising edge is detected for IN, and WAIT_LOW state, where we wait for the IN to go low again.

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 Verilog

- 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**.



```verilog
module pulse_gen (pulse, in, clk);

  input in, clk;
  output pulse;

  reg  [1:0] state;
  reg  pulse;

// define states binary encoding
  parameter IDLE = 2'b00;
  parameter IN_HIGH = 2'b01;
  parameter WAIT_LOW = 2'b10;

  initial state = IDLE;
  initial pulse = 1'b0;
```

```verilog
// specify state machine transition
always @ (posedge clk)
      case (state)
         IDLE:    if (in==1'b1) state <= IN_HIGH;
         IN_HIGH: if (in==1'b1) state <= WAIT_LOW;
                        else state <= IDLE;
         WAIT_LOW: if (in==1'b0) state <= IDLE;
         default: ;   // do nothing
      endcase

// specify output combinational logic
always @ (*)
   case (state)
      IDLE: pulse = 1'b0;
      IN_HIGH: pulse = 1'b1;
      WAIT_LOW: pulse = 1'b0;
   endcase

endmodule
```

PYKC 28 Oct 2019 E2.1 Digital Electronics Lecture 7 Slide 14

This FSM has three states: IDEL, IN_HIGH and WAIT_LOW. Mapping the state diagram to Verilog is straight forward.

1.The declaration part is standard. This is followed by the **parameter section**.. Here we use straight forward binary number assignment, and therefore we have two state bits (maximum four states, but only three are used).

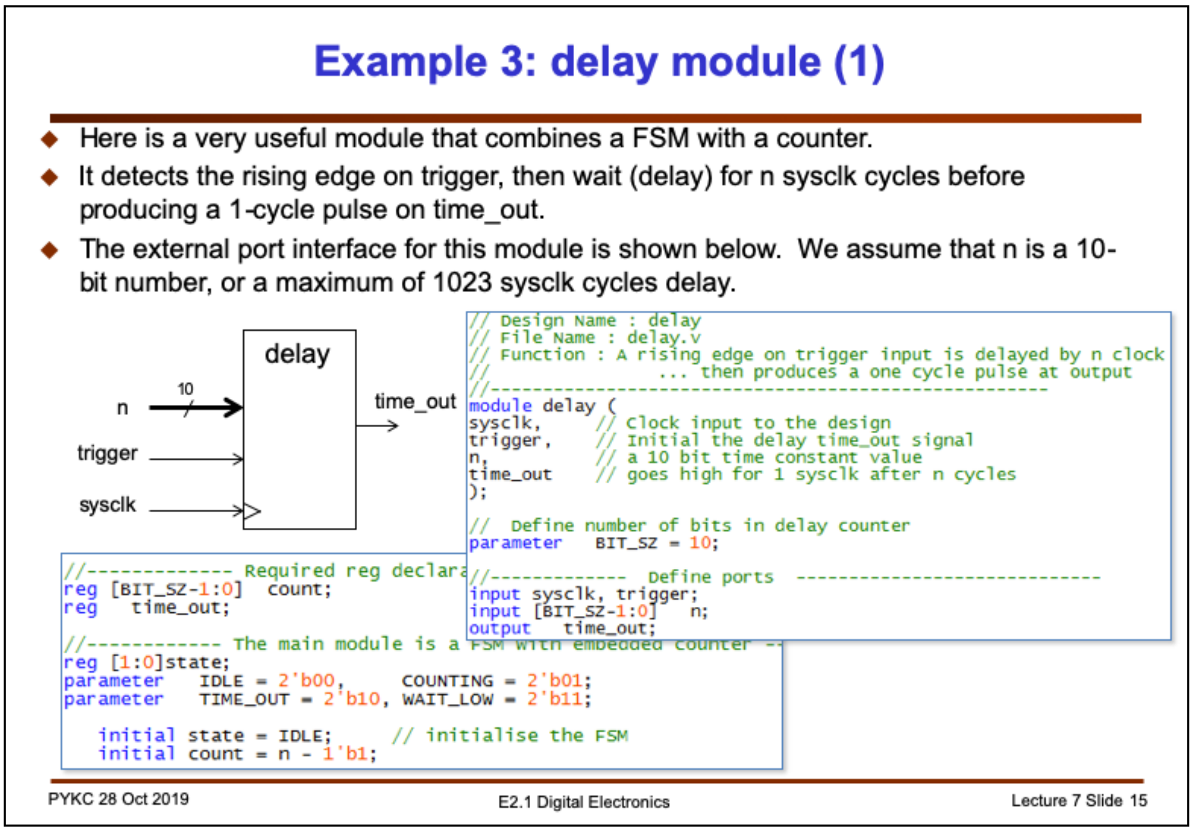
2.The initial section is for initialization. Normally for a FSM design, it is best to include a RESET input signal which, when asserted, will synchronously put the state machine to an initial state. Here we are using a nice feature of FPGAs, which allows the digital circuits to be initialised to any states during CONFIGURATION (i.e. when downloading the bit-stream). When you configure the FPGA, the registers used for state[1:0] will be loaded with the value 2'b00The actual state machine is specified with the always @ block.

3.The first line defines the **default output value** for pulse is 0. This ensures that pulse is always defined.

4.The case statement is the best way to specify a FSM. Each case specifies both the conditions for state transitions and the output. It is important to note that state and output specified for each CASE are the next state and next output. For example, if the FSM is in the IDLE state and in==1'b1 on the next positive edge of clk, the FSM will go to state IN_HIGH and make pulse go high.

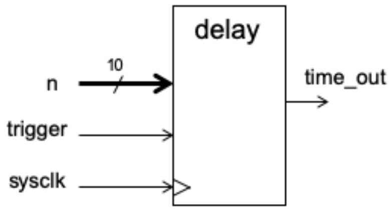
5.The <= assignment specifies that the changes will occur simultaneously when the always @ block is exited.

6.Finally, the default section will catch all unspecified cases. In this case, default section is empty (i.e. by default, do nothing). YOU MUST ALSO INCLUDE THE DEFAULT SECTION IN YOUR FSM DESIGN.



Example 3: 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 sysclk 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 10-bit number, or a maximum of 1023 sysclk cycles delay.



```verilog
// Design Name : delay
// File Name : delay.v
// Function : A rising edge on trigger input is delayed by n clock
//                ... then produces a one cycle pulse at output
//-----------------------------------------------------
module delay (
sysclk,      // Clock input to the design
trigger,     // Initial the delay time_out signal
n,           // a 10 bit time constant value
time_out     // goes high for 1 sysclk after n cycles
);

//  Define number of bits in delay counter
parameter   BIT_SZ = 10;

//-------------  Define ports  -----------------------------
input sysclk, trigger;
input [BIT_SZ-1:0]   n;
output   time_out;
```

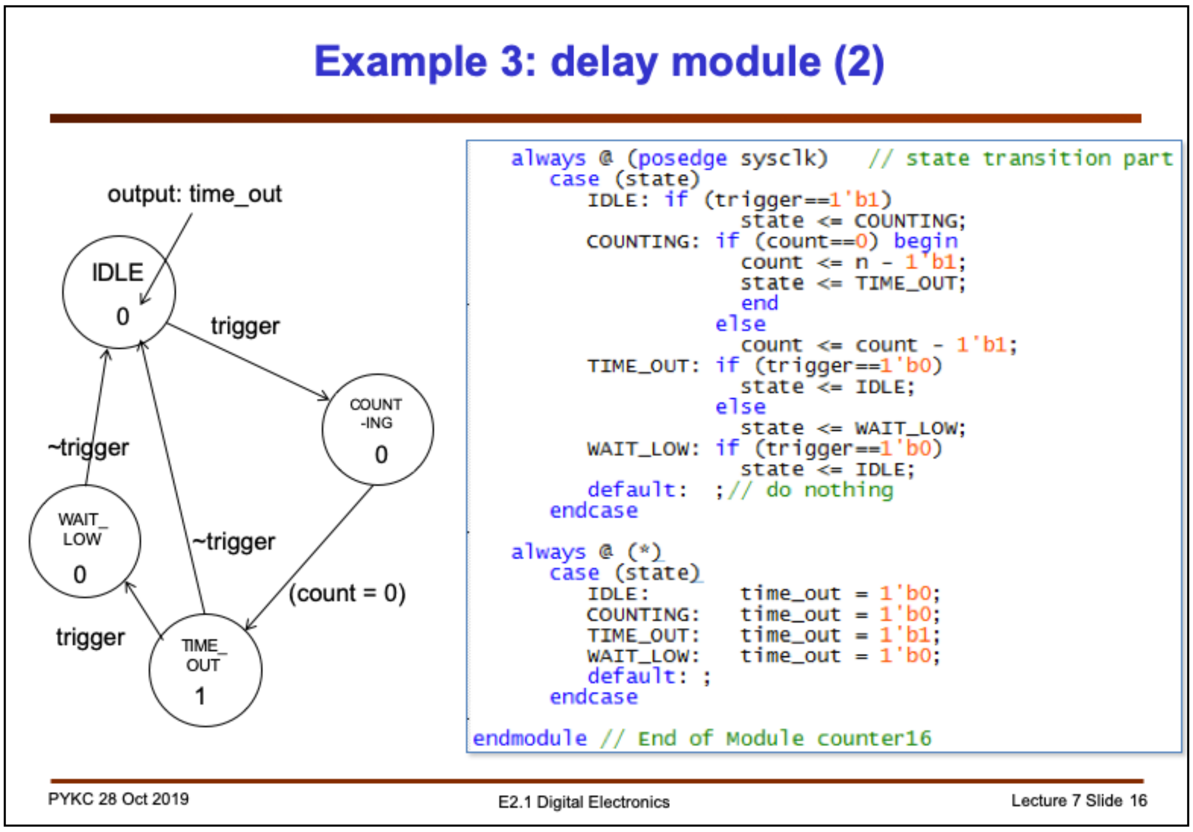
```verilog
//------------- Required reg declara
reg [BIT_SZ-1:0]  count;
reg   time_out;

//------------ The main module is a FSM with embedded counter --
reg [1:0]state;
parameter   IDLE = 2'b00,     COUNTING = 2'b01;
parameter   TIME_OUT = 2'b10, WAIT_LOW = 2'b11;

   initial state = IDLE;     // initialise the FSM
   initial count = n - 1'b1;
```

PYKC 28 Oct 2019 E2.1 Digital Electronics Lecture 7 Slide 15

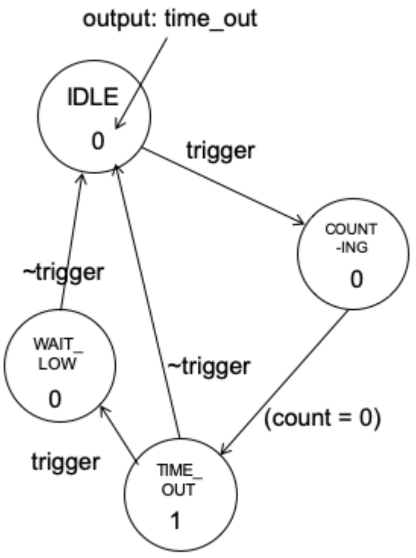
Finally, here is a very useful module that uses a four-state FSM and a counter. It is the combination of the previous example with a down counter embedded inside the FSM.

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

Here we have the port interface and the declaration parts of the Verilog design.



Example 3: delay module (2)



```
   always @ (posedge sysclk)   // state transition part
      case (state)
         IDLE: if (trigger==1'b1)
                     state <= COUNTING;
         COUNTING: if (count==0) begin
                     count <= n - 1'b1;
                     state <= TIME_OUT;
                     end
                   else
                     count <= count - 1'b1;
         TIME_OUT: if (trigger==1'b0)
                     state <= IDLE;
                   else
                     state <= WAIT_LOW;
         WAIT_LOW: if (trigger==1'b0)
                     state <= IDLE;
         default:  ;// do nothing
      endcase

   always @ (*)
      case (state)
         IDLE:       time_out = 1'b0;
         COUNTING:   time_out = 1'b0;
         TIME_OUT:   time_out = 1'b1;
         WAIT_LOW:   time_out = 1'b0;
         default: ;
      endcase

endmodule // End of Module counter16
```

PYKC 28 Oct 2019 E2.1 Digital Electronics Lecture 7 Slide 16

The FSM state diagram is very similar to that for pulse_gen.v. However we have four states instead of three. Go through this yourself and make sure that you understand how this works.