Imperial College London

MSc in Analogue & Digital IC Design

Pulse-width Modulator, Finite State Machines & Serial-Peripheral Interface

Peter Cheung
Department of Electrical & Electronic Engineering
Imperial College London

Course webpage: www.ee.ic.ac.uk/pcheung/teaching/MSc_Experiment/
E-mail: p.cheung@imperial.ac.uk

I hope you have completed Part 2 of the Experiment and is ready for Part 3.

In part 3, you are going to use the FPGA to interface with the external world through a DAC and a ADC on the add-on card. You will also learn about FSM design and PWM module. Finally the DAC and ADC use a serial interface known as SPI. We will take a brief look at this interface standard without going into details of how to write Verilog to specify the SPI module design.

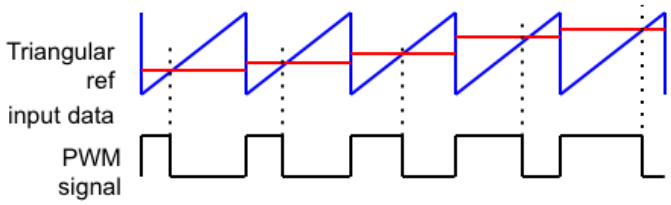
Lecture Objectives

- PWM module and how it works
- Basic about Finite State Machine (FSM)
- How to specify a FSM in Verilog
- The analogue interface add-on card
- Serial Peripheral Interface (SPI) for the DAC and ADC

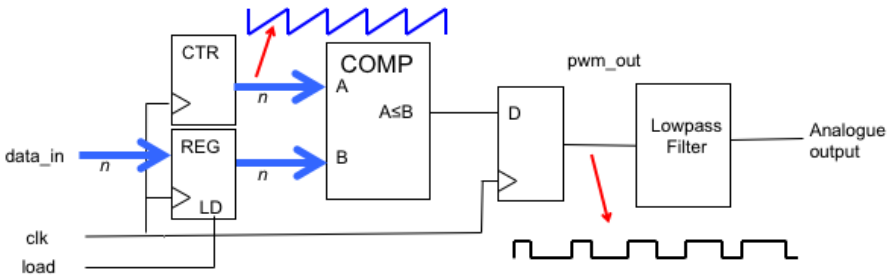
Here again is a list of topics covered in this lecture. The we basically will cover three things: PWM, FSM design and SPI for interfacing. All these are relevant to Part 3 of the Experiment for this week.

Pulse-width Modulated (PWM) DAC

- Simple idea: PWM signal is generated by comparing a triangular reference signal with the input data value



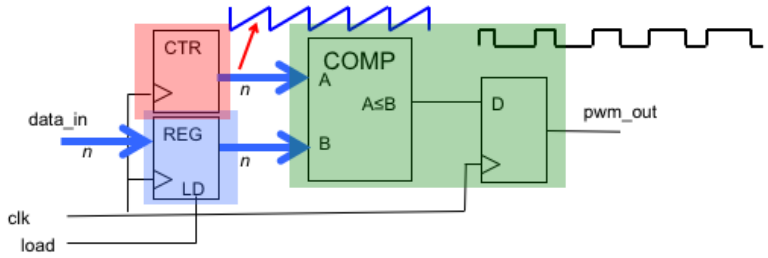
- Triangular value generated by a wrap-around counter
- Sample command pulse resets counter, load register and set FF
- When input value is reached by counter, comparator output a pulse to reset FF



PYKC 26 Oct 2017 MSc Lab – Mastering Digital Design Lecture 3 Slide 3

Instead of using analogue resistor network, it is possible to build a simple DAC using only digital components.
Here is a circuit schematic for a pulse-width modulated DAC. Here the counter is used to produce a count value A that ramps up linearly in a sawtooth manner. The digital value we want to convert to analogue value is data_in, which is stored as B in the input register. A digital comparator circuit compares this input data with the counter value (which is ramping up). While A is less than B, the output of the comparator is high. As soon as A exceeds B, the output goes low. In this way, the pulse width is proportional to the value of B (or data_in) in a linear manner. Passing this PWM signal through a lowpass filter will give an analogue output which is linearly related to data_in.

PWM DAC in Verilog (ex11)




```
module pwm (clk, data_in, load, pwm_out);

   input          clk;        // system clock
   input [9:0]    data_in;    // input data for conve
   input          load;       // high pulse to load r
   output         pwm_out;    // PWM output

   reg [9:0]      d;          // internal register
   reg [9:0]      count;      // internal 10-bit cour
   reg            pwm_out;

   always @ (posedge clk)
      if (load == 1'b1) d <= data_in;
```

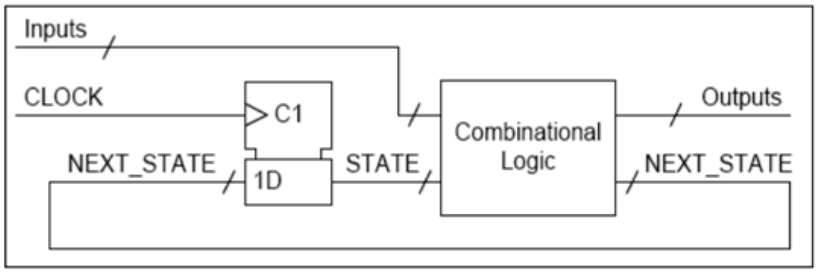
```
initial count = 10'b0;

always @ (posedge clk) begin
   count <= count + 1'b1;
   if (count > d)
      pwm_out <= 1'b0;
   else
      pwm_out <= 1'b1;
   end

module
```

PYKC 26 Oct 2017 MSc Lab – Mastering Digital Design Lecture 3 Slide 4

This is how the PWM module works. It is very simple, but very effective. You should compare the DAC output and PWM output in Part 3 of the experiment, and see that the two methods are equally effective in producing an analogue voltage.

Synchronous State Machines

- **Synchronous State Machine (also called Finite State Machine FSM)**
  = Register + Logic



  - The ***state*** is defined by the register contents
  - Register has *n* flipflops ⇒ 2^n 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

Rules:

- Never mess around with the clock signal
- Never use ***asynchronous*** SET/RESET inputs to register (*asynchronous* = independent of CLOCK)

PYKC 26 Oct 2017 MSc Lab – Mastering Digital Design Lecture 3 Slide 5

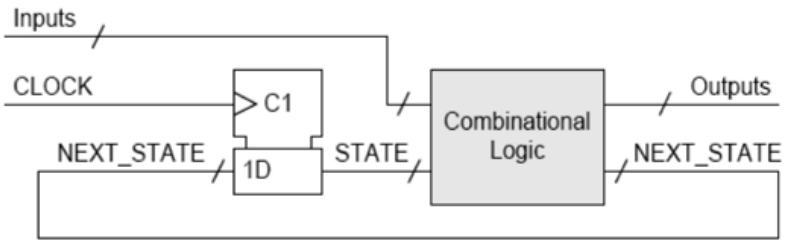
Here is a simplified generic diagram of a finite (or synchronous) state machine (FSM or SSM). A set of D-flipflips are used to store the current state value. The current state together with external inputs are fed to a combinational logic circuit to evaluate two things: the **next state** and the **current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

With an n-bit register and using binary state encoding (i.e. coding states as binary number), 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. Therefore all output signals are synchronized.

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 SSM is used in conjunction with the rest of the system.
2.Do not use asynchronous SET or RESET signals. Doing so would make the rest of the system NOT synchronous to the CLOCK signal.

Combinational Logic Block



- The combinational logic outputs specify two things:
  - **the output signals during the current state**
    These may change during the state if the inputs change
  - **which state to go to at the next CLOCK**
    This too may change during a state but the only thing that matters is its value just before CLOCK
- ***combinational*** logic has no internal feedback loops ⇒ no memory
  - combinational logic outputs are entirely determined by the **current STATE** and the **current Inputs**

PYKC 26 Oct 2017 MSc Lab – Mastering Digital Design Lecture 3 Slide 6

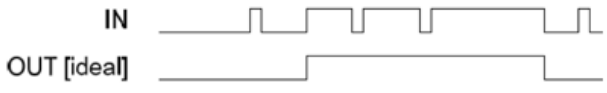
The combinational logic circuit in a FSM performs two separate tasks:

1. It determines what **the output signals** should be. This derived by the current state value STATE and the current inputs. Therefore such output signals could change in the middle of a clock cycle if input signals are NOT synchronized with the CLOCK.
2. It determines what the **next state value** should be, i.e. the state transition of the FSM.

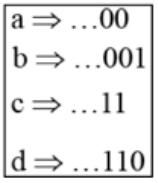
The combinational logic block (by definition) contains no memory (or register) circuit.

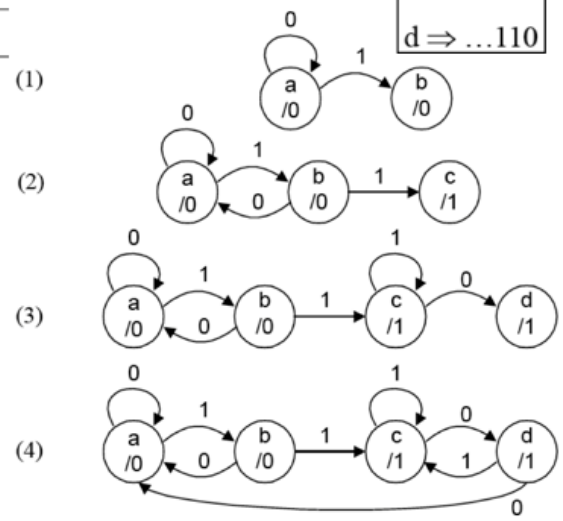
Example 1: Design a Noise Pulse Eliminator (1)

- **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.





PYKC 26 Oct 2017 MSc Lab – Mastering Digital Design Lecture 3 Slide 7

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.

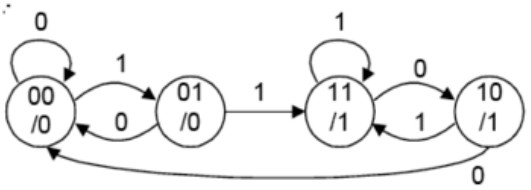
Each state represents a particular history that we need to distinguish from the others:

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 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**:



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 26 Oct 2017 MSc Lab – Mastering Digital Design Lecture 3 Slide 9

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.

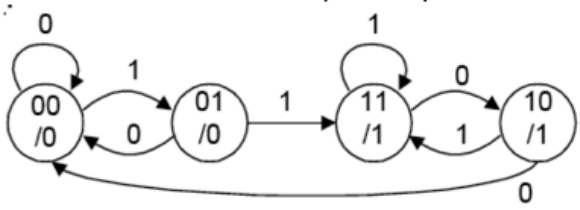
Note that you could assign ANY binary number to any state – and the implemented FSM will work. However, different state encoding will result in different implementations, affecting the complexity of the digital logic.

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

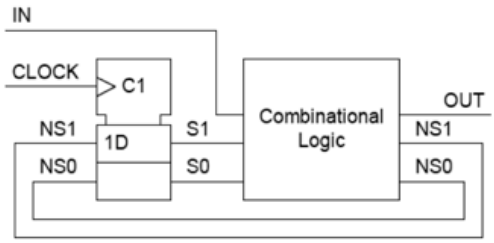
PYKC 26 Oct 2017 MSc Lab – Mastering Digital Design Lecture 3 Slide 10

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)

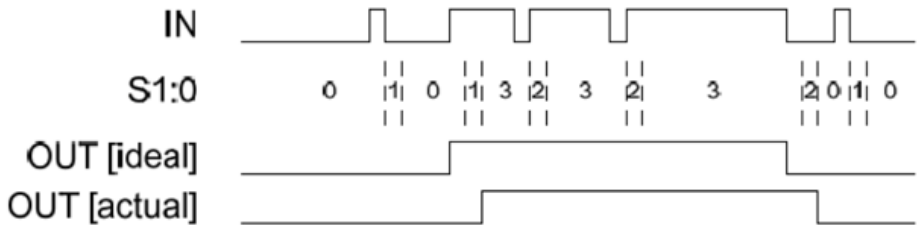
- From this we can derive Boolean expressions for the combinational logic block:

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



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



PYKC 26 Oct 2017 MSc Lab – Mastering Digital Design Lecture 3 Slide 11

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.

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').

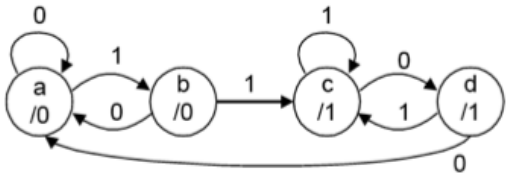
Using one-hot encoding matches the FPGA architecture well. Each FPGA logic element contains a combinational logic module and one or more registers. Therefore FPGA is a register-rich architecture.

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

```
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;
```

Declarations




```
// 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
```

State transitions

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
```

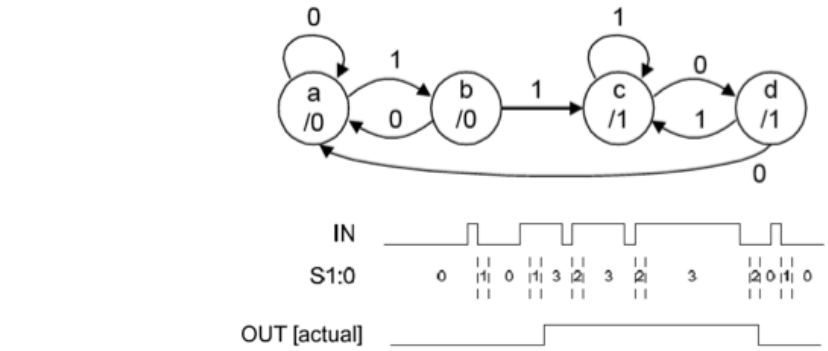

Instead of manually designing a state machine, we usually rely on Verilog specification and synthesis CAD tools such as Altera's Quartus software.
Here we use an EXPLICIT reset signal **rst** to put the state machine in a known state. We also use **one-hot** instead of binary encoding of the states. This is specified in the **parameter block**.

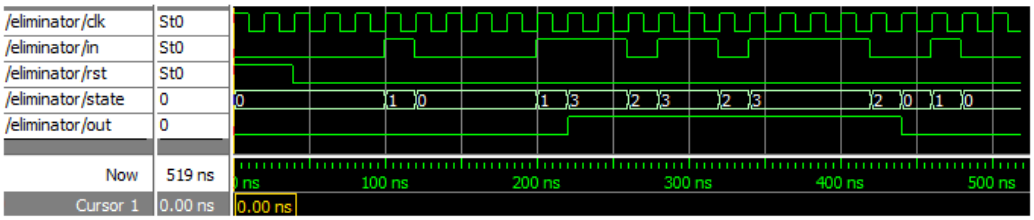
Using parameter block to give a name to each of the states has many benefits: the Verilog design is much easier to read; you can change state assignment values without needing to change any codes. In general, parameter block allows you to use **symbols** (names) to replace **numbers**. This makes the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



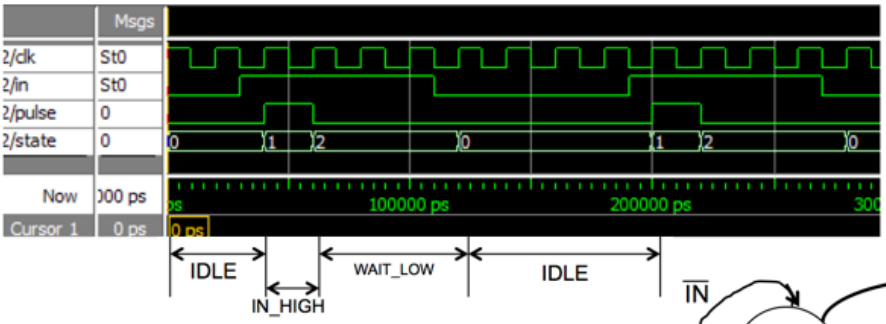


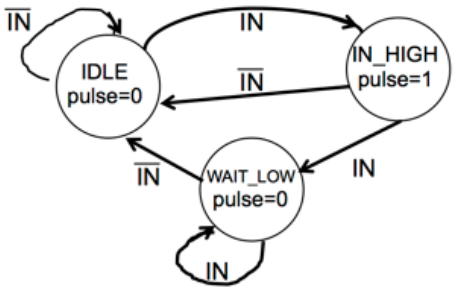
PYKC 26 Oct 2017 MSc Lab – Mastering Digital Design Lecture 3 Slide 14

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).

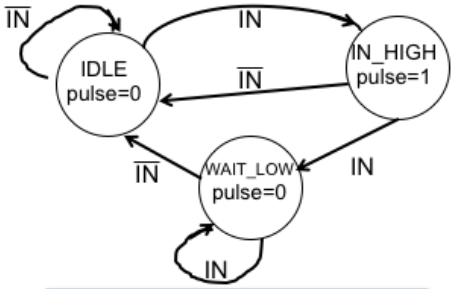
Let us now consider another example, which will appear in the Lab Experiment later. You are required to design a pulse generator circuit that, on the positive edge of the input **IN**, a pulse lasting for one clock period is produced.

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**.




```
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;
```

```
// 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 26 Oct 2017 MSc Lab – Mastering Digital Design Lecture 3 Slide 16

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.

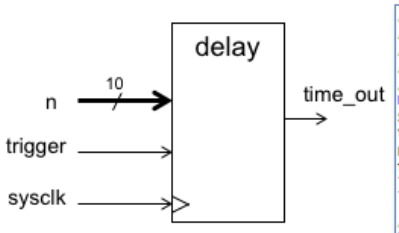
4.The case statement is the best way to specify a FSM. Each case specifies both the conditions for state transitions and the output. It is important to note that state and output specified for each CASE are the next state and next output. For example, if the FSM is in the IDLE state and in==1'b1 on the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



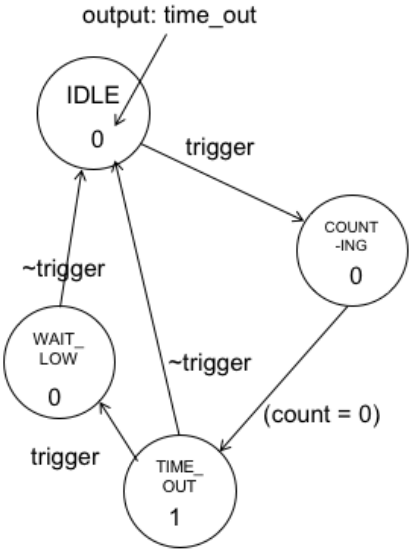

```
// Design Name : delay
// File Name : delay.v
// Function : A rising edge on trigger input is delayed by n clock
//                  ... then produces a one cycle pulse at output
//-----------------------------------------------------
module delay (
sysclk,      // Clock input to the design
trigger,     // 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;
```

```
//------------- 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 26 Oct 2017 MSc Lab – Mastering Digital Design Lecture 3 Slide 17

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)



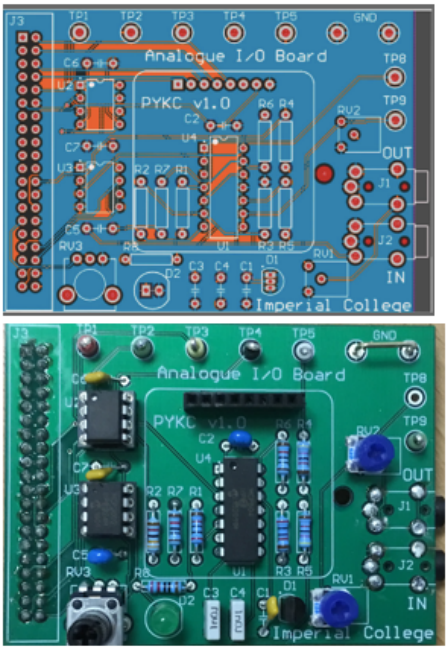

```
always @ (posedge sysclk)   // state transition part
   case (state)
      IDLE: if (trigger==1'b1)
               state <= COUNTING;
      COUNTING: if (count==0) begin
                  count <= n - 1'b1;
                  state <= TIME_OUT;
                  end
                else
                  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 26 Oct 2017 MSc Lab – Mastering Digital Design Lecture 3 Slide 18

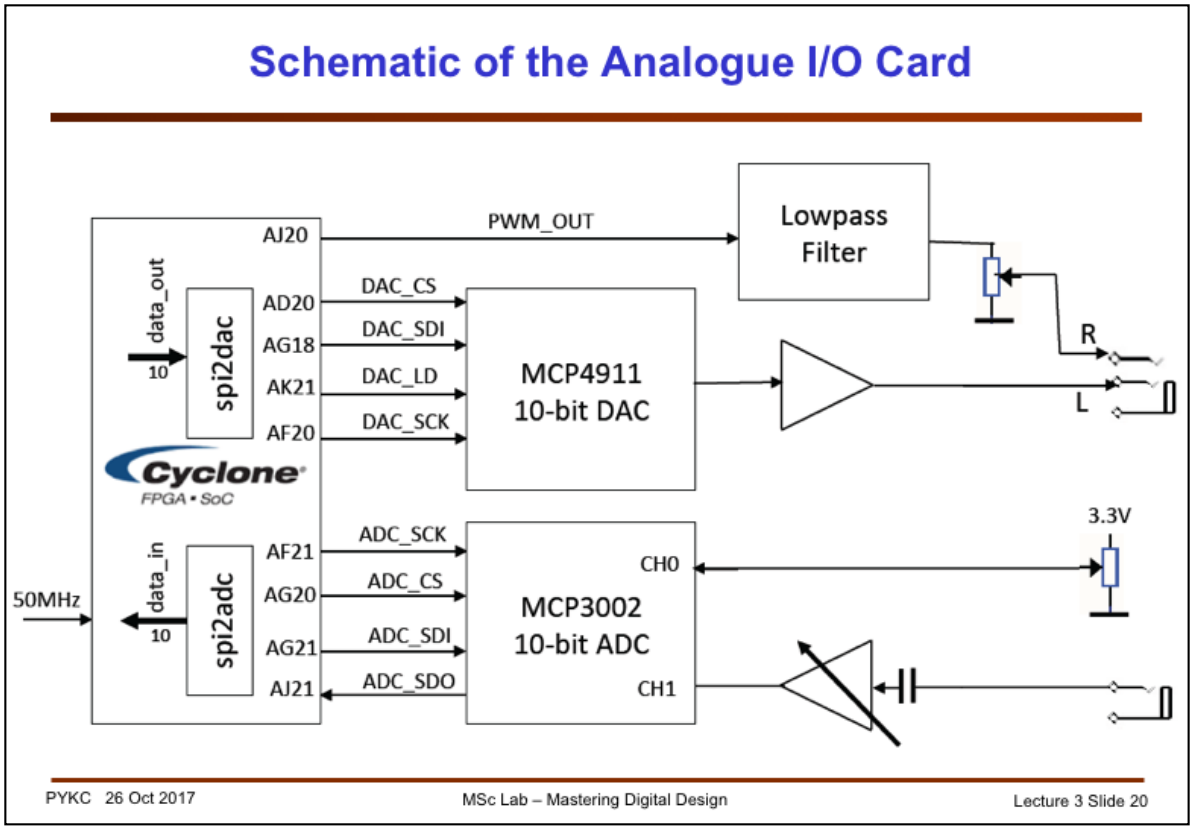
The FSM state diagram is very similar to that for pulse_gen.v. However we have four states instead of three. Go through this yourself and make sure that you understand how this works.

The Analogue I/O Card

- Provides analogue inputs and outputs
- Contains 2 channels ADC, one for a dc voltage set by a potentiometer & another from a socket
- Has 1 DAC to connected to the right channel, and a digital output to the left channel of a headphone socket
- Includes low-pass filter and operational amplifiers
- Will be using this board for Experiment: VERI part 3 and 4



I also provide a purpose-built ADC/DAC board to support the lab experiment. This analogue I/O board in only needed for Part 3 and 4 of VERI. However I will now be examining the digital serial interface for these converter chips.



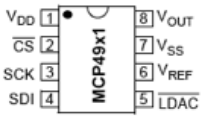
This shows the block diagram of the analogue I/O card used in the VERI experiment. It consists of a DAC (MCP4911) and a ADC (MCP3002), both using Serial Peripheral Interface (SPI). The DAC output is buffered by a unity gain opamp connected to the right channel of a stereo jack socket.

The ADC has two input channels, one from a potentiometer providing a dc voltage (CH0) and another from the 3.5mm jack socket (CH1).

Finally, there is a 2nd order low-pass active filter, the input of which is driven directly from a digital output pin of the Cyclone FPGA. This is intended to provide filtering of a pulse-width modulated DAC output from the FPGA.

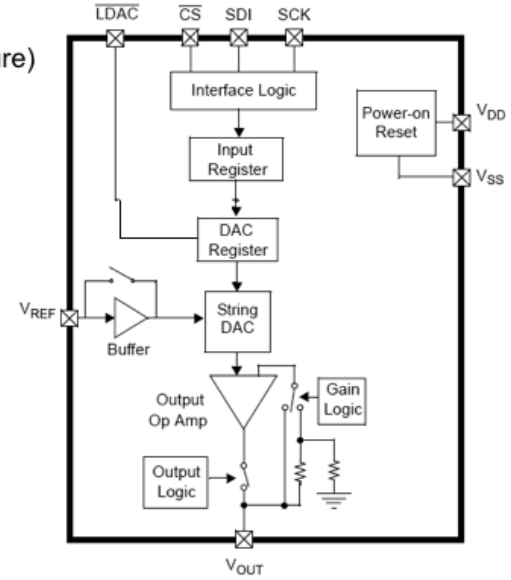
DAC – used in analogue I/O card

- **Microchip MCP4911** 10-bit DAC
- Uses **resistor string** architecture (earlier lecture)
- Serial Peripheral Interface (SPI)



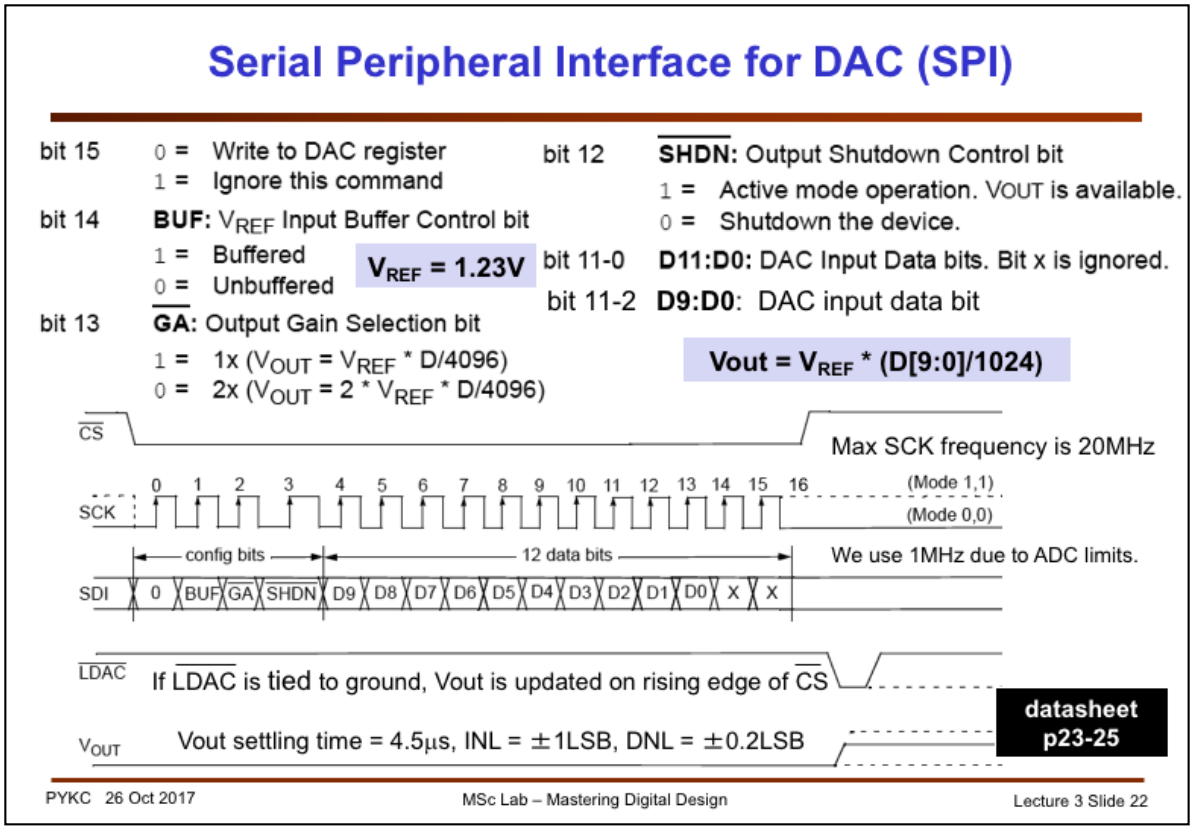
- Rail-to-Rail Output
- SPI Interface with 20 MHz Clock Support
- Simultaneous Latching of the DAC Output with $\overline{LDAC}$ Pin
- Fast Settling Time of 4.5 µs
- Selectable Unity or 2x Gain Output
- External Voltage Reference Input
- External Multiplier Mode

Symbol	Description
V_{DD}	Supply Voltage Input (2.7V to 5.5V)
$\overline{CS}$	Chip Select Input
SCK	Serial Clock Input
SDI	Serial Data Input
$\overline{LDAC}$	DAC Output Synchronization Input. This pin is used to transfer the input register (DAC settings) to the output register (V_{OUT})
V_{REF}	Voltage Reference Input
V_{SS}	Ground reference point for all circuitry on the device
V_{OUT}	DAC Analog Output



PYKC 26 Oct 2017 MSc Lab – Mastering Digital Design Lecture 3 Slide 21

The DAC used with the I/O card is 10-bit, and it uses the Serial Peripheral interface. Its functional block diagram is shown here. The SPI interface has four signals, which should be drive by either the microcontroller or the FPGA. The DAC itself uses a resistor string architecture (i.e. just a bunch of 1024 series resistors of identical values). It has a selectable gain of 1X or 2X.



To send a value to the DAC to output (i.e. produce the analogue output Vout), a 16-bit value is sent to the DAC chip in a serial manner. The Chip Select (SC) signal going low indicate that this is the start of the data. This establishes the beginning of the data frame. First data bit (bit 15) is always 0. Bit 14 determines whether the reference voltage (Vreg) is buffered or not buffered (via an internal opamp). For our design, Vref is around 3.3V.

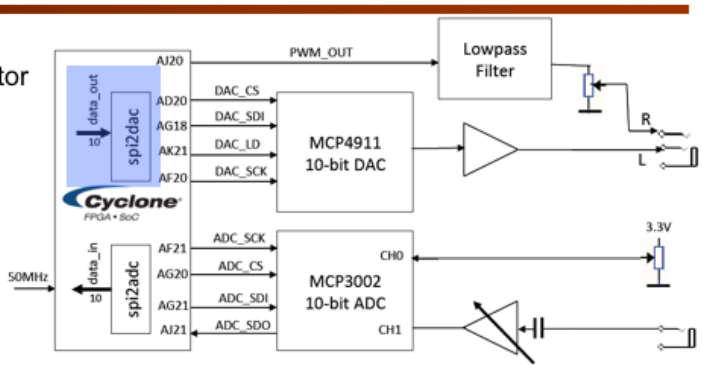
Bit 13 determines the gain of the DAC (x1 or x2). Bit 12 is set to 1 if you are using the DAC, and set to 0 if you want to shutdown the device to conserve power.

Bit 11 to 2 contains the 10-bit data D[9:0] to convert into analogue voltage Vout, MSB first. Bit 1 and 0 are don't cares.

The LDAC (low active) signal can be connected to ground or used a low active strobe signal to transfer the data to the DAC register (i.e. tell the DAC to update Vout). If LDAC is low, DAC update happens on rising edge of CS_bar.

Interfacing the FPGA to the DAC and ADC

- Overview of the DAC/ADC
- DAC is DC coupled (no capacitor in signal path)
- ADC is AC coupled (why?)
- Interface circuit to DAC:
  - spi2dac.v
- Interface circuit to ADC
  - spi2adc.v



Important points to note

- DAC and ADC function are NOT done within Cyclone V FPGA
- Conversion from/to analogue signals are done with 2 8-pin chips on Add-on card
- Why do we need serial-parallel interface circuits? To fit everything within 8-pin package
- A single serial clock is used for both ADC and DAC – set at 1MHz
- This is different from the system clock of 50MHz (fixed within DE0)
- Chip-select is low only when sending serial data to DAC chip on SDI pin
- LDA is low only when all 10-bit data sent and DAC to be loaded with new value

This is a simplified diagram showing how the Cyclone V FPGA is interfaced to the two data converters. There are two ADC channels and in our experiment, we are mostly using channel 1 via the 3.5mm jack socket. You will be supplying speech signals from the desktop computer.

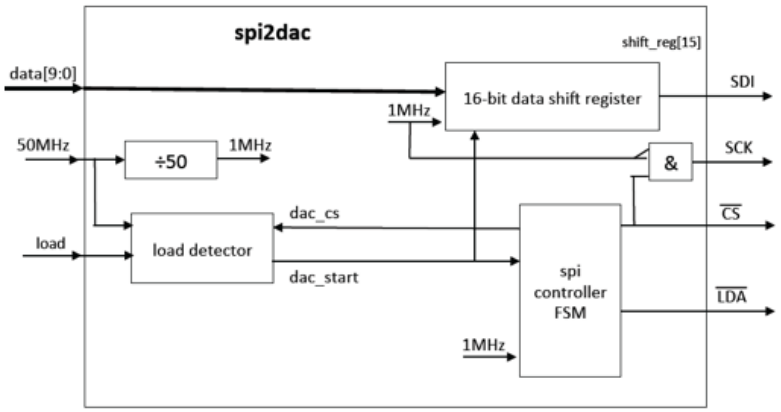
There is one DAC which drives both the small speaker and, much better, drives the ear-phone. (Please bring the ear-phone to the lab.)

The interface between the FPGA chip and the converters is through the SPI bus. You are given the Verilog design for these two interface modules: spi2dac.v and spi2adc.v. In the rest of this lecture, I will be going through the design of the spi2dac module.

spi2dac design overview

- The components inside spi2dac are:

1. Clock divider
2. Load detector to detect load pulse
3. FSM to control the spi interface
4. Parallel to serial shift register to shift OUT the command and data to the DAC
5. Various gates e.g. inverters and AND gates



- Note that the Verilog code is designed to match the block diagram shown here
- It consists of TWO state machines, a counter and a shift register

PYKC 26 Oct 2017 MSc Lab – Mastering Digital Design Lecture 3 Slide 24

In order to use the DAC, you have to include the interface module “spi2dac” in your design. This module has a schematic shown above. It takes two inputs (in addition to the 50MHz clock signal): data[9:0] is the 10-bit digital data to be converted by the DAC, and a load signal which is a high pulse to trigger the spi2dac module to send the 10-bit data to the DAC.

The internal working of sp2dac can be divided into 4 main modules. The divide-by-50 module is straight forward – it produces a 1MHz clock for the finite state machine, and is gated through the AND gate to generate the serial clock signal (at 1MHz).
The load detector module handles the load command and produces control signals to the SPI state machine and the shift register.
The shift register sends the control bits and the 10-bit data serially to the SDI output.
The spi controller FSM is the main control module designed as a state machine.

We will consider each sub-module individually in next week’s lecture.