Imperial College London

MSc in Analogue & Digital IC Design

Lecture 3
Pulse-width Modulator, Finite State Machines & Serial-Peripheral Interface

Peter Cheung
Department of Electrical & Electronic Engineering
Imperial College London

Course webpage: https://github.com/Mastering-Digital-Design/Lab-Module
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.

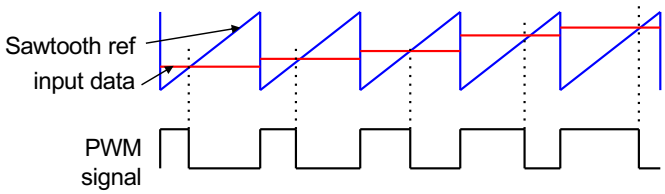
Lecture Objectives

- 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

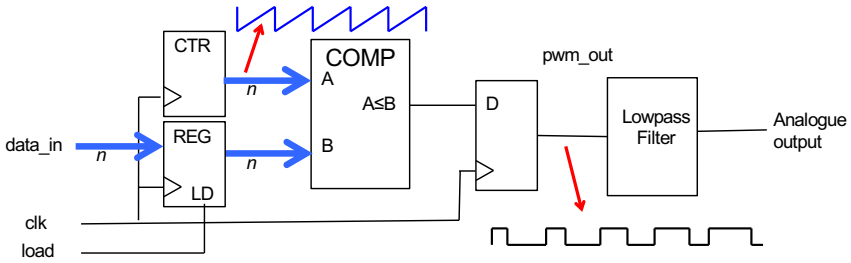
Here again is a list of topics covered in this lecture. The we basically will cover three things: PWM, FSM design and SPI for interfacing. All these are relevant to Part 3 of the Experiment for this week.

Pulse-width Modulated (PWM) DAC

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



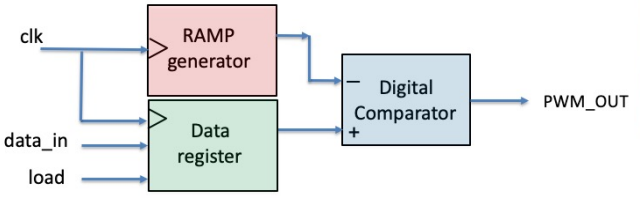
- Sawtooth value generated by a wrap-around counter
- Sample command pulse resets counter, load register and set FF
- When input value is reached by counter, comparator output a pulse to reset FF



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. 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 SystemVerilog

```
module pwm # (parameter WIDTH = 10)(
    input  logic                clk,        // system clock
    input  logic [WIDTH-1:0]    data_in,    // input data for conversion (limited to 10-bit)
    input  logic                load,       // high pulse to load new data
    input  logic [WIDTH-1:0]    max,        // maximum value of data_in
    output logic                pwm_out     // PWM output
);
    logic [WIDTH-1:0]       d;              // internal registe
    logic [WIDTH-1:0]       count;          // internal 10-bit
```




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

initial count = {WIDTH{1'b0}};

always_ff @ (posedge clk) begin
    if (count == max)
        count <= {WIDTH{1'b0}};
    else
        count <= count + 1'b1;

    if (count >= d)
        pwm_out <= 1'b0;
    else
        pwm_out <= 1'b1;
    end
```

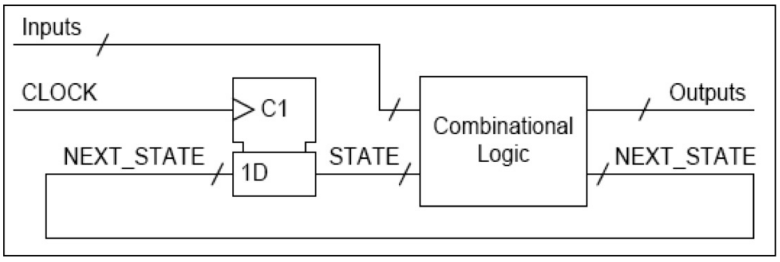
PYKC 1 Nov 2024 MSc Lab – Mastering Digital Design Lecture 3 Slide 4

Implementing a PWM DAC is extremely simple in SystemVerilog. Here is the code with colour coding for the three major blocks.

This module deliberately has reset signal missing. Instead the counter is initialized to zero in the SystemVerilog specification with “initial” keyword. This is possible for FPGA because initialization happens when you send the bitstream (sof file) to the FPGA during programming. If you were designing for an ASIC (Application Specific Integrated Circuit), which is not an FPGA, you must always use an explicit reset signal to reset all states in the chip.

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)

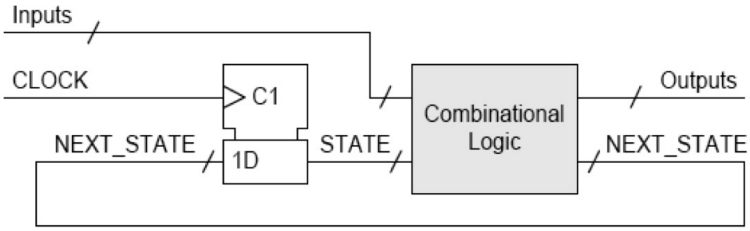
Here is a simplified generic diagram of a finite (or synchronous) state machine (FSM or SSM). A set of D-flipflips are used to store the current state value. The current state together with external inputs are fed to a combinational logic circuit to evaluate two things: the **next state** and the **current 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 1 Nov 2024 MSc Lab – Mastering Digital Design Lecture 3 Slide 6

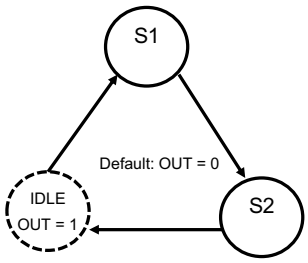
The combinational logic circuit in a FSM performs two separate tasks:

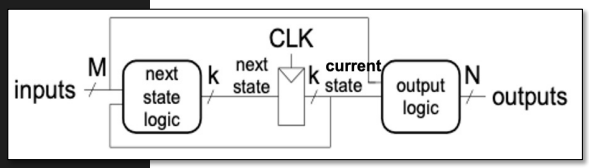
1. It determines what **the output signals** should be. This derived by the current state value STATE and the current inputs. Therefore such output signals could change in the middle of a clock cycle if input signals are NOT synchronized with the CLOCK.
2. It determines what the **next state value**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

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

Example 1: Divide by 3 FSM (Moore)

```
1  module div3FSM (
2      input   logic clk,  // clock signal
3      input   logic rst,  // asynchronous reset
4      output  logic out   // goes high 1 cycle every 3 clk cycles
5  );
6
7      // Define our states
8      typedef enum {IDLE, S1, S2}  my_state;
9      my_state current_state, next_state;
10
11     // state registers
12     always_ff @(posedge clk, posedge rst)
13         if (rst)     current_state <= IDLE;
14         else         current_state <= next_state;
15
16     // next state logic
17     always_comb
18         case (current_state)
19             IDLE:    next_state = S1;
20             S1:      next_state = S2;
21             S2:      next_state = IDLE;
22             default: next_state = IDLE;
23         endcase
24
25     // output logic
26     assign out = (current_state == IDLE);
27 endmodule
```





PYKC 1 Nov 2024 MSc Lab – Mastering Digital Design Lecture 3 Slide 7

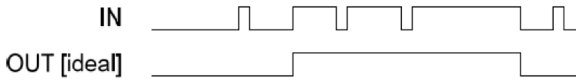
Here is a very simple three states FSM. The intial state is the IDLE state (here shown in dotted lines). The purpose of this machine is to divide clock signal by three – out goes high for one cycle every three.

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

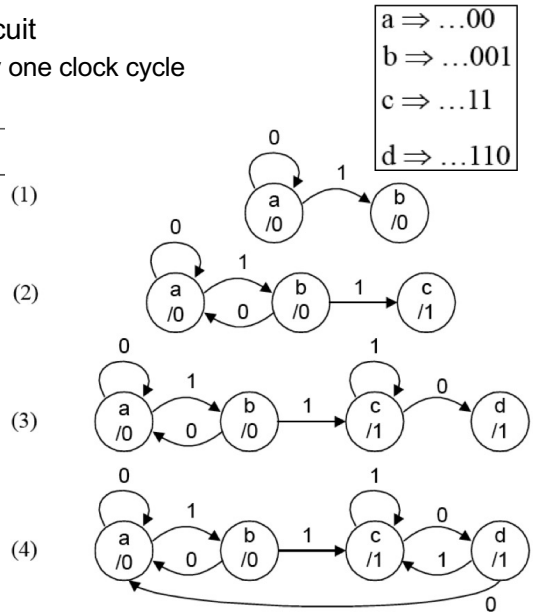
1. Interface declaration – define the input and output signals.
2. State enumeration – specify an enumated type, here we call it my_state, and give all states a state name (e.g. IDLE, S1, … etc), and then declare two state variables: current_state, and next_state.
3. State registers – specify the registers 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 1 Nov 2024 MSc Lab – Mastering Digital Design Lecture 3 Slide 8

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:

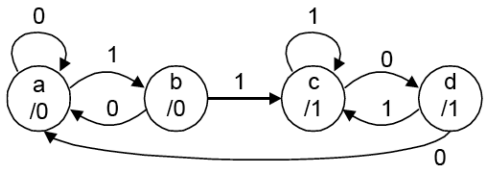
state **a**: IN=0 for >1 clock	state **b**: IN=1 for 1 clock
state **c**: IN=1 for >1 clock	state **d**: IN=0 for 1 clock

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.

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:

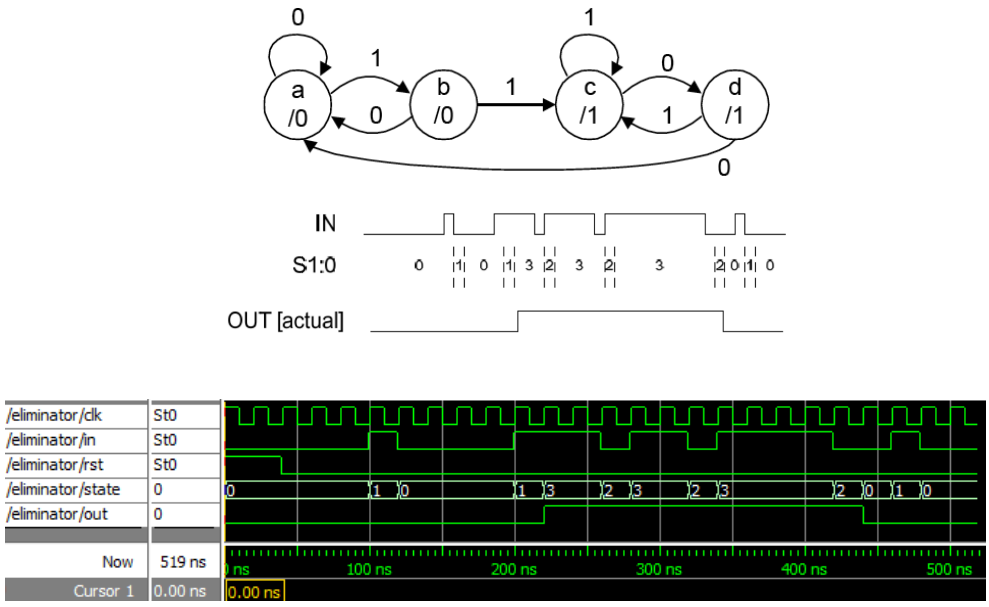
  a = 0001 b = 0010 c = 0100 d = 1000
- Using one-hot encoding would use MORE state registers. For N-states, we would need to use N flipflops.
- The advantage is that the state transition and output logic could be much simpler than using binary encoding. There is no longer need for logic to decode the 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.

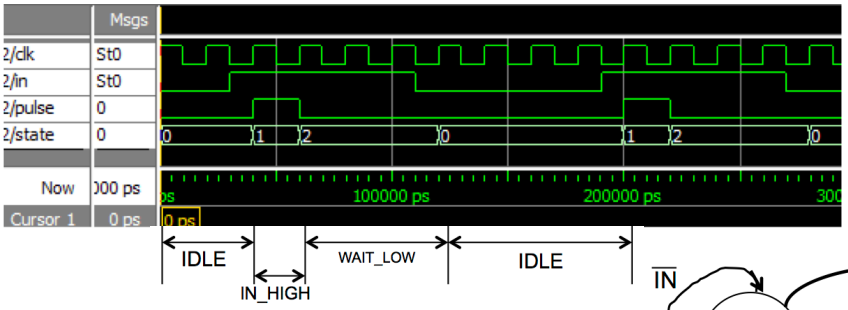
Eliminator simulation in Quartus (RTL)

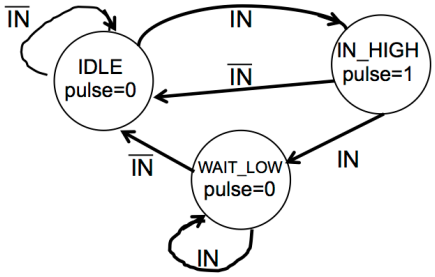


If you enter this Verilog description into Quartus and simulate the circuit, you will see the waveform as shown in this timing diagram as expected. Note that the actual waveform for out is NOT the ideal waveform, but is delayed by one clock cycle.

Example 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).

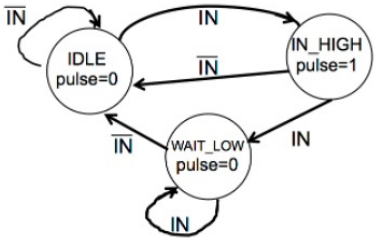
Let us now consider another example, which will appear in the Lab Experiment later. You are required to design a pulse generator circuit that, on the positive edge of the input **IN**, a pulse lasting for one clock period is produced.

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;
```

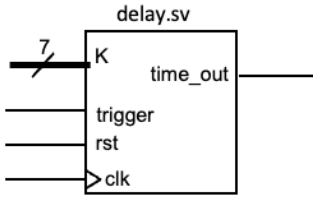
```
// next state logic
always_comb
    case (current_state)
        IDLE:       if (in==1'b1)   next_state = IN_HIGH;
                    else            next_state = current_state;
        IN_HIGH:    if (in==1'b1)   next_state = WAIT_LOW;
                    else            next_state = IDLE;
   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



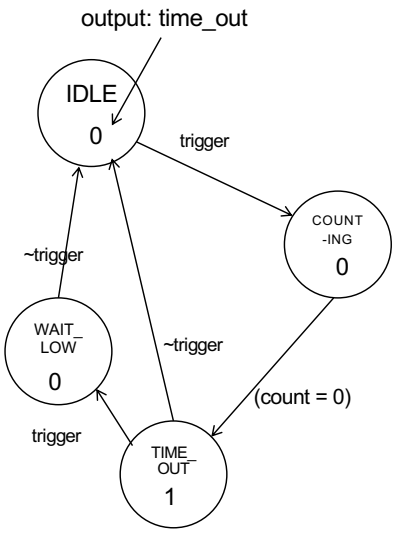

```
module delay #(
    parameter WIDTH = 7     // no of bits in delay counter
)(
    input   logic               clk,        // clock signal
    input   logic               rst,        // reset signal
    input   logic               trigger,    // trigger input signal
    input   logic [WIDTH-1:0] 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

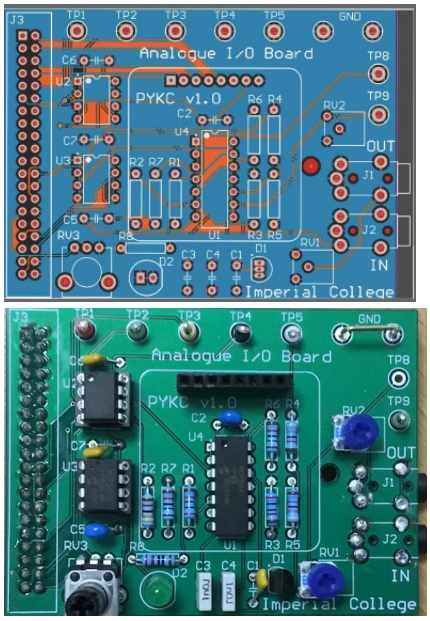

Finally, we need the sequential circuit specifications for state transistion, and for the counter logic itself.

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.

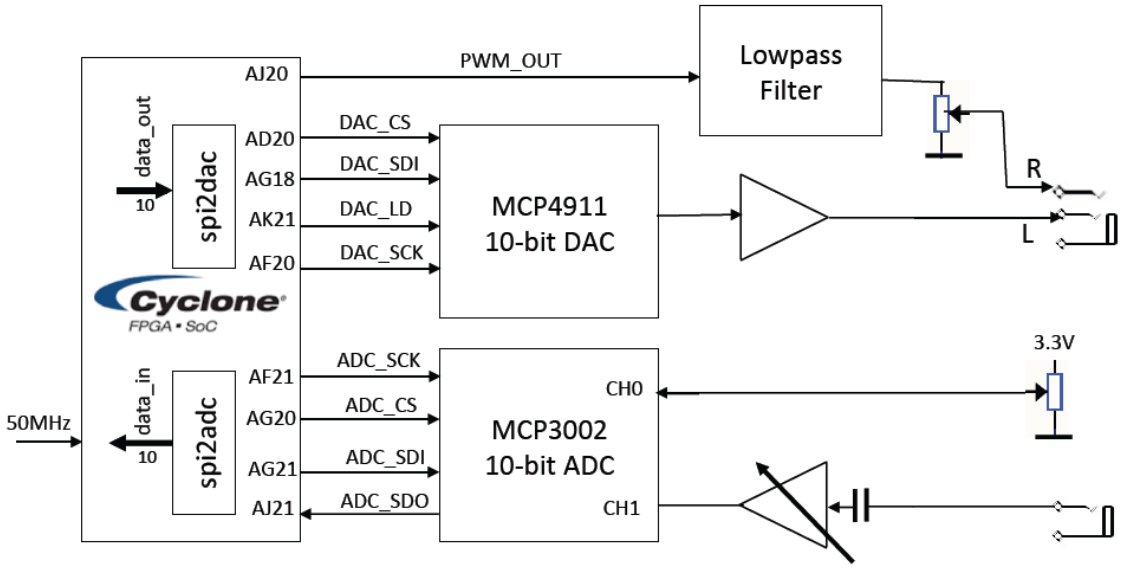
The Analogue I/O Card

- 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.

Schematic of the Analogue I/O Card



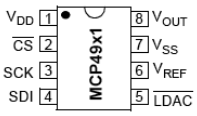
This shows the block diagram of the analogue I/O card used in the VERI experiment. It consists of a DAC (MCP4911) and a ADC (MCP3002), both using Serial Peripheral Interface (SPI). The DAC output is buffered by a unity gain opamp connected to the right channel of a stereo jack socket.

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.

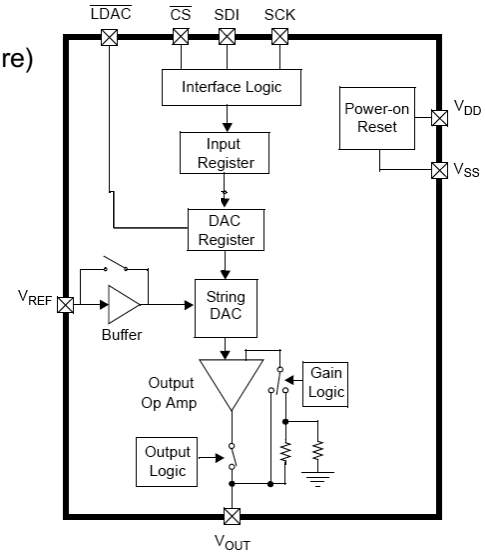
DAC – used in analogue I/O card

- **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 $\overline{\text{LDAC}}$ 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 1 Nov 2024 MSc Lab – Mastering Digital Design Lecture 3 Slide 20

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.

Serial Peripheral Interface for DAC (SPI)

bit 15 0 = Write to DAC register
1 = Ignore this command

bit 14 **BUF:** V_{REF} Input Buffer Control bit
1 = Buffered
0 = Unbuffered

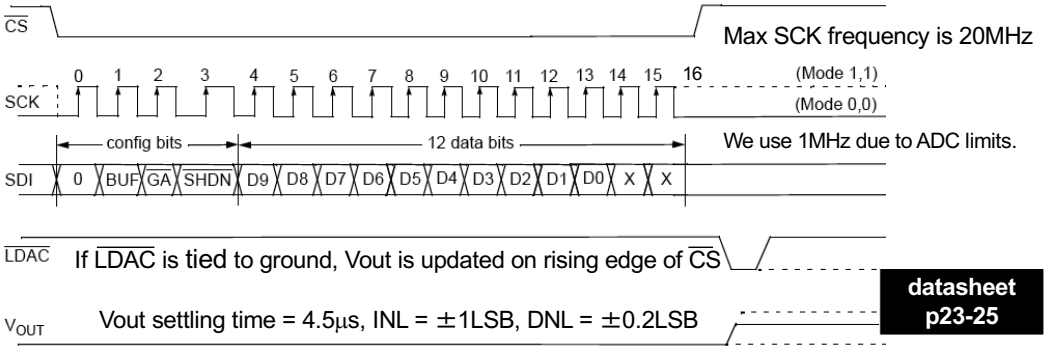
V_{REF} = 1.23V

bit 13 **$\overline{GA}$:** Output Gain Selection bit
1 = 1x (V_{OUT} = V_{REF} * D/4096)
0 = 2x (V_{OUT} = 2 * V_{REF} * D/4096)

bit 12 **$\overline{SHDN}$:** Output Shutdown Control bit
1 = Active mode operation. VOUT is available.
0 = Shutdown the device.

bit 11-0 **D11:D0:** DAC Input Data bits. Bit x is ignored.
bit 11-2 **D9:D0**: DAC input data bit

Vout = V_{REF} * (D[9:0]/1024)



datasheet p23-25

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.

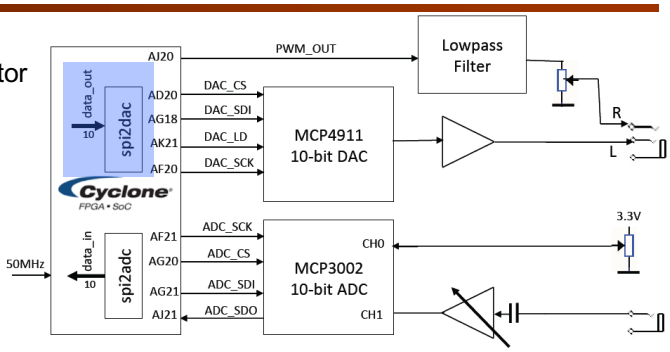
Bit 13 determines the gain of the DAC (x1 or x2). Bit 12 is set to 1 if you are using the DAC, and set to 0 if you want to shutdown the device to conserve power.

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.sv
- Interface circuit to ADC
  - spi2adc.sv



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 DE1-SoC)
- 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.