Imperial College
London

Lecture 12
Serial-Peripheral Interface (SPI)

Peter Cheung
Department of Electrical & Electronic Engineering
Imperial College London

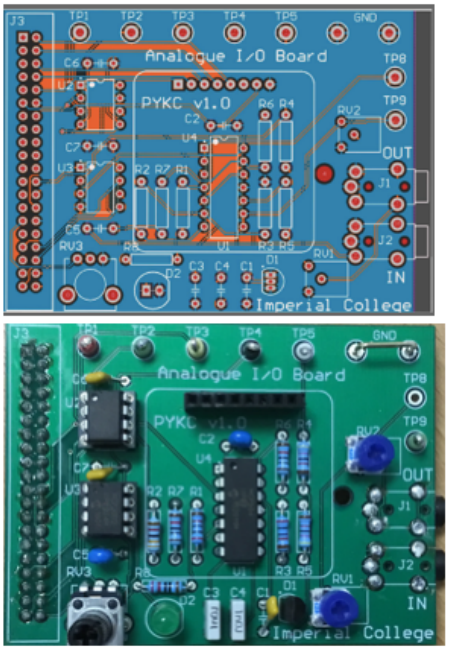
URL: www.ee.imperial.ac.uk/pcheung/teaching/ee2_digital/
E-mail: p.cheung@imperial.ac.uk

Lecture Objectives

- Learn about the DAC used in the Analogue I/O card
- Serial Peripheral Interface used by the DAC – how it works?
- Explore in details the Verilog design of the SPI interface module
- Examine the ADC used in the Analogue I/O card

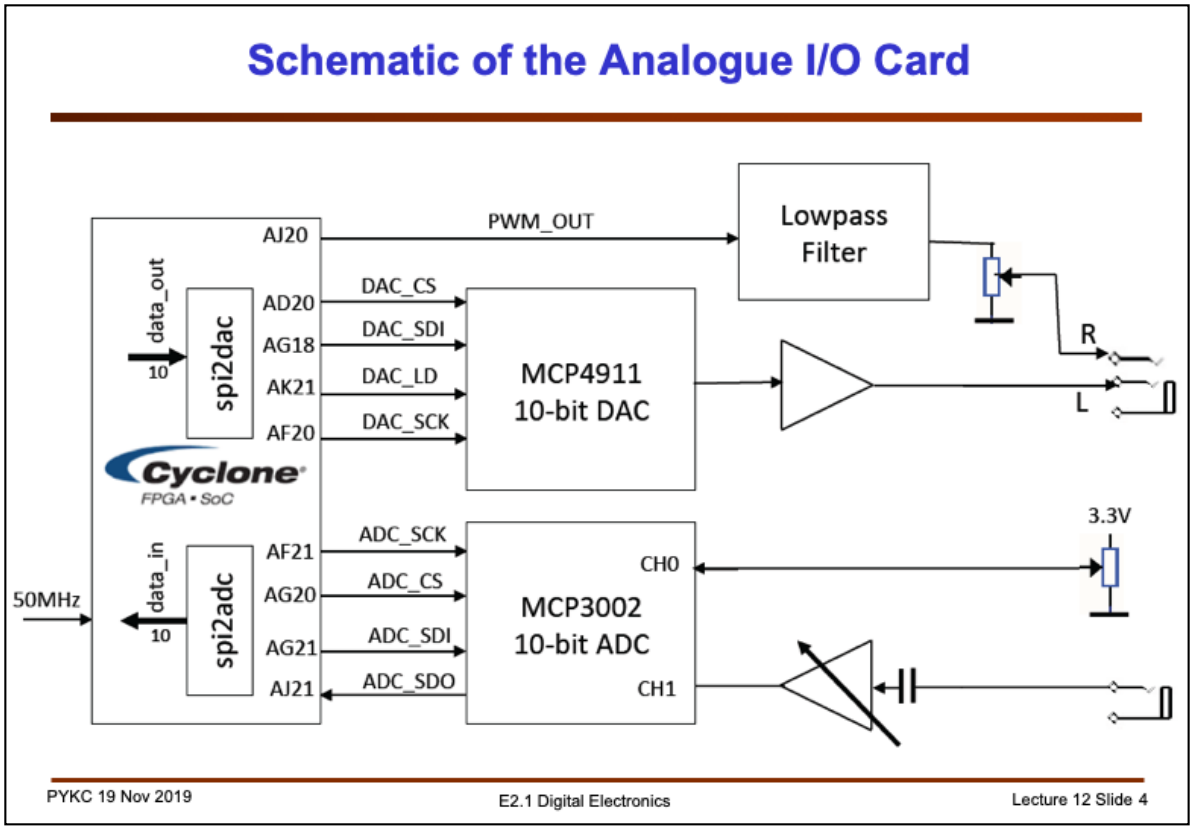
The Analogue I/O Card

- 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



PYKC 19 Nov 2019 E2.1 Digital Electronics Lecture 12 Slide 3

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.



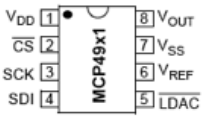
This shows the block diagram of the analogue I/O card used in the VERI experiment. It consists of a DAC (MCP4911) and a ADC (MCP3002), both using Serial Peripheral Interface (SPI). The DAC output is buffered by a unity gain opamp connected to the right channel of a stereo jack socket.

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.

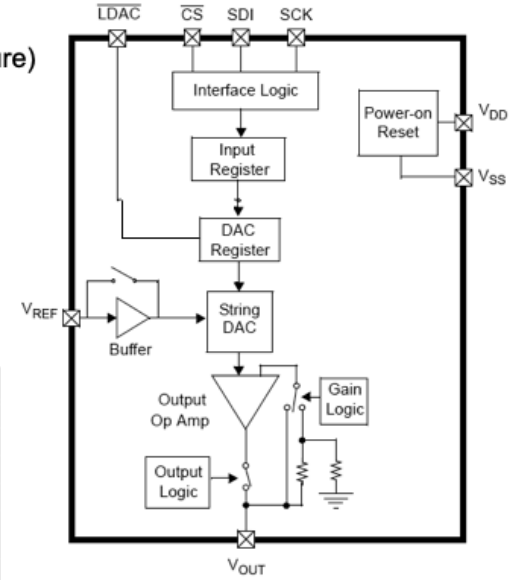
DAC – used in analogue I/O card

- **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



R12.1 p19-21

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.

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

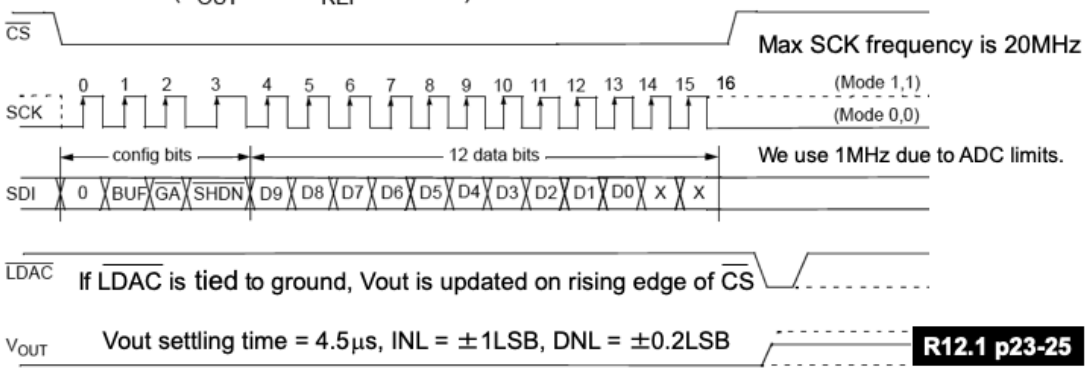
bit 13 **$\overline{GA}$:** Output Gain Selection bit
1 = 1x (V_{OUT} = V_{REF} * D/4096)
0 = 2x (V_{OUT} = 2 * V_{REF} * D/4096)

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)



R12.1 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.

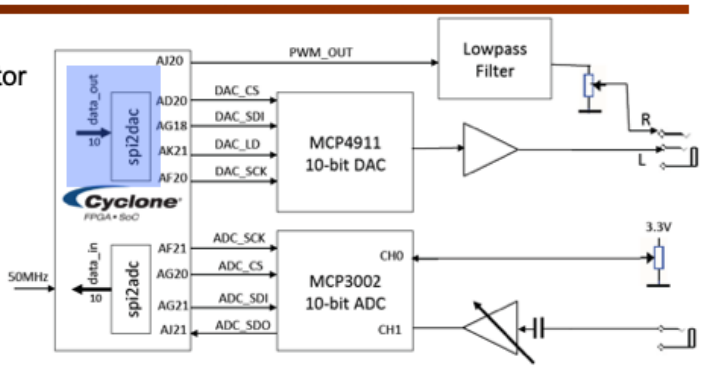
Bit 13 determines the gain of the DAC (x1 or x2). Bit 12 is set to 1 if you are using the DAC, and set to 0 if you want to shutdown the device to conserve power.

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.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 DE1)
- 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.

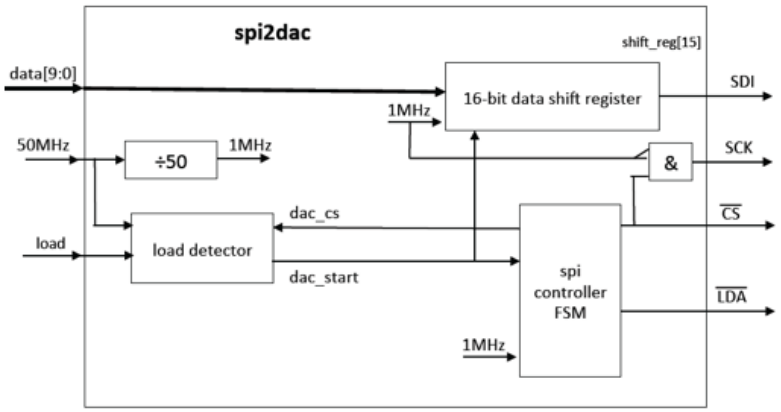
There is one DAC which drives both the small speaker and, much better, drives the ear-phone. (Please bring the ear-phone to the lab.)

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

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.

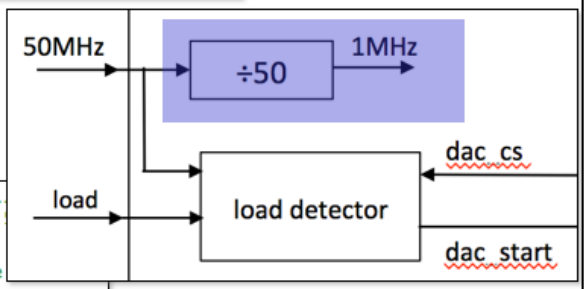
The shift register sends the control bits and the 10-bit data serially to the SDI output.

The spi controller FSM is the main control module designed as a state machine.

We will now consider each sub-module individually.

The 1MHz clock generator

```
parameter    BUF=1'b1;       // 0:no buffer, 1:Vref buffered
parameter    GA_N=1'b1;      // 0:gain = 2x, 1:gain = 1x
parameter    SHDN_N=1'b1;    // 0:power down, 1:dac active

wire [3:0] cmd = {1'b0,BUF,GA_N,SHDN_N};   // wire to VDD or GND
```

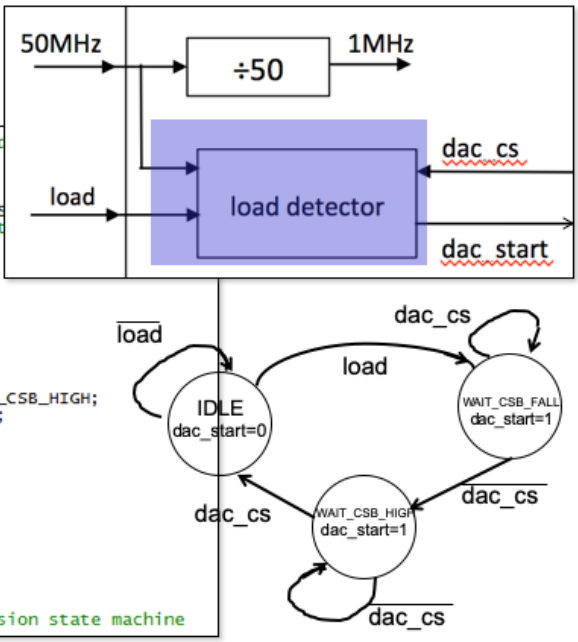



```
// --- internal 1MHz symmetical clock generator ---
reg          clk_1MHz;    // 1Mhz clock derived from
reg [4:0]    ctr;         // internal counter

parameter    TC = 5'd24;  // Terminal count - change
initial begin
   clk_1MHz = 0;          // don't need to reset - don't care
   ctr = 5'b0;            //  ... Initialise when FPGA is confi
end

always @ (posedge sysclk)
  if (ctr==0) begin
     ctr <= TC;
     clk_1MHz <= ~clk_1MHz; // toggle the output clock for sc
   end
  else
     ctr <= ctr - 1'b1;
// ---- end internal 1MHz symmetical clock generator --------
```


This is a straight forward clock divider. The Terminal Count (TC) is set to 24. Divide by 50 is done by toggling the output (clk_1MHz) after 25 clock cycles. Note that I generally prefer to use a down-counter instead of an up-counter. The counter (ctr) is set to 24, it then counts to zero. Output is toggled and the counter (ctr) is reset to the initial value of 24 again.

The load pulse detector

```
// ---- FSM to detect rising edge of load and falling ed
// .... sr_state set on posedge of load
// .... sr_state reset when dac_cs goes high at the end
reg [1:0]   sr_state;
parameter   IDLE  = 2'b00,WAIT_CSB_FALL = 2'b01, WAIT_CS
reg         dac_start;      // set if a DAC write is det

initial begin
   sr_state = IDLE;
   dac_start = 1'b0; // set while sending data to DAC
   end

always @ (posedge sysclk)  // state transition
   case (sr_state)
      IDLE: if (load==1'b1) sr_state <= WAIT_CSB_FALL;
      WAIT_CSB_FALL: if (dac_cs==1'b0) sr_state <= WAIT_CSB_HIGH;
      WAIT_CSB_HIGH: if (dac_cs==1'b1) sr_state <= IDLE;
      default: sr_state <= IDLE;
   endcase

always @ (*)
   case (sr_state)
      IDLE: dac_start = 1'b0;
      WAIT_CSB_FALL: dac_start = 1'b1;
      WAIT_CSB_HIGH: dac_start = 1'b0;
      default: dac_start = 1'b0;
   endcase

//------- End circuit to detect start and end of conversion state machine
```



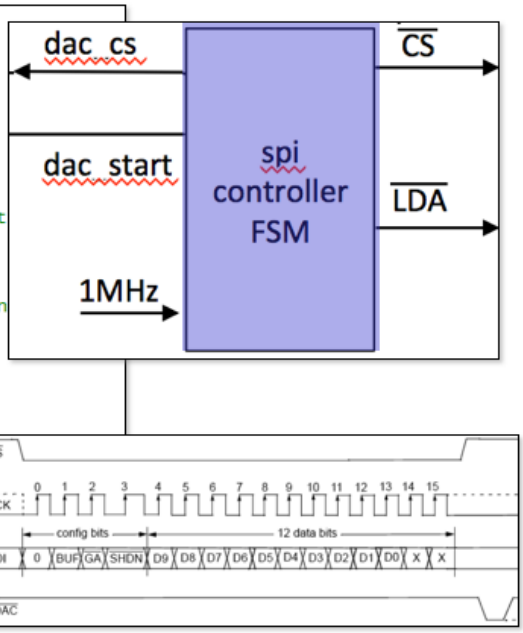
PYKC 19 Nov 2019 E2.1 Digital Electronics Lecture 12 Slide 10

We have TWO signals to detect: the load pulse and the dac_cs signal.

Starting in the IDLE state, when load signal is asserted, we start the DAC cycle by entering the WAIT_CSB_FALL state. In this state, dac_start is asserted, and we wait for DAC_CS to go low from the SPI controller circuit. In this condition, the DAC is in the middle of accepting a new data for conversion. We go to state WAIT_CSB_HIGH TO wait for the conversion to be completed, which is indicated by DAC_CS going high. When that happens, we return to the IDLE state waiting for another 10-bit data to be loaded.

The SPI Controller FSM

```
//------- spi controller FSM
// .... with 17 states (idle, and S1-S16
// .... for the 16 cycles each sending 1-bit to dac)
reg [4:0]   state;

initial  begin
   state = 5'b0; dac_ld = 1'b0; dac_cs = 1'b1;
   end

always @(posedge clk_1MHz)  // FSM state transition
   case (state)
      5'd0: if (dac_start == 1'b1)     // waiting to start
               state <= state + 1'b1;
            else
               state <= 5'b0;
      5'd17:   state <= 5'd0;  // go back to idle state
      default: state <= state + 1'b1;  // default go to n
   endcase

always @ (*)   begin       // FSM output
   dac_cs = 1'b0;  dac_ld = 1'b1;
   case (state)
      5'd0:    dac_cs = 1'b1;
      5'd17:   begin dac_cs = 1'b1; dac_ld = 1'b0; end
      default: begin dac_cs = 1'b0; dac_ld = 1'b1; end
      endcase
   end //always
// --------- END of spi controller FSM
```



The controlling FSM controller is actually simpler than it first appears.

We need a FSM to have 18 states. State 0 is the idle state, waiting for a new data to be sent to the DAC. Here DAC_CS (which is low active) is '1' and we wait for the dac_start to be asserted.

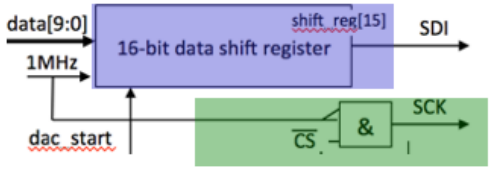
The default value of dac_cs and dac_ld are specified first. By default we always go to the next state, i.e. state value goes up by 1.

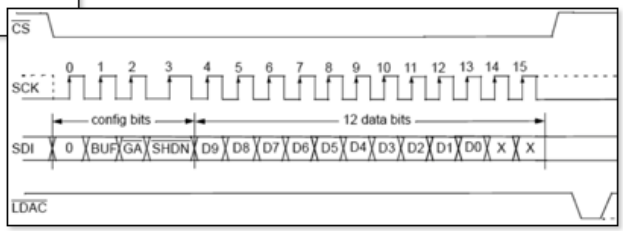
Once the state machine moves to state 1, it just go through to state 16, which corresponds to cycle 0 to 15 in the timing diagram here. At the end of state 16, we de-assert dac_cs (i.e. go high), assert dac_ld (low) and go back to the IDLE state.

The data shift register

```
parameter   BUF=1'b1;       // 0:no buffer, 1:Vref buffered
parameter   GA_N=1'b1;      // 0:gain = 2x, 1:gain = 1x
parameter   SHDN_N=1'b1;    // 0:power down, 1:dac active

wire [3:0] cmd = {1'b0,BUF,GA_N,SHDN_N};  // wire to VDD or GND
```

```
// shift register for output data
reg [15:0] shift_reg;
initial  begin
   shift_reg = 16'b0;
   end

always @(posedge clk_1MHz)
   if((dac_start==1'b1)&&(dac_cs==1'b1))      //
      shift_reg <= {cmd,data_in,2'b00};
   else                                       //
      shift_reg <= {shift_reg[14:0],1'b0};

// Assign outputs to drive SPI interface to DAC
      assign dac_sck = !clk_1MHz&!dac_cs;
      assign dac_sdi = shift_reg[15];
```





Finally, the data and clock output is specified here. SDI is driven through a parallel in, serial out shift register.

We use a number of useful tricks here:

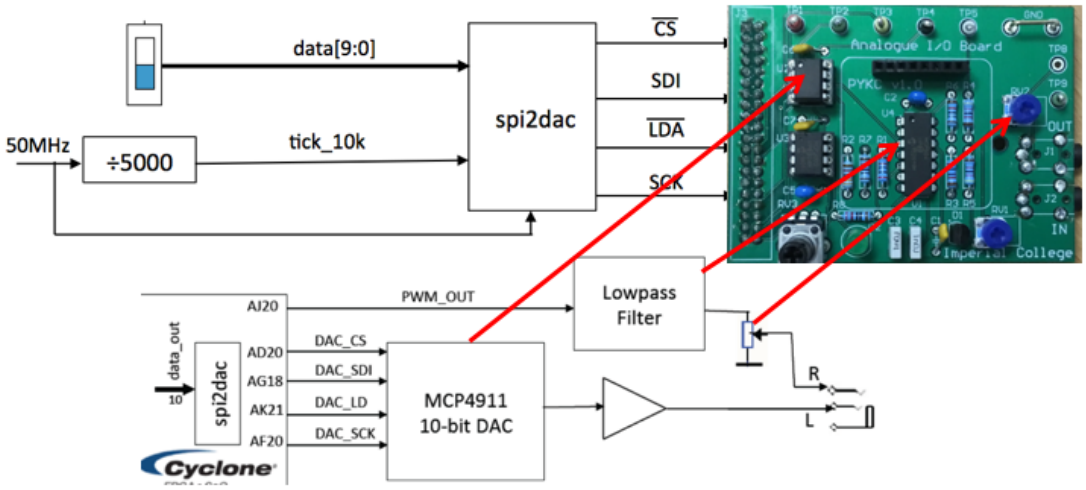
1.cmd is a 4-bit value defining the first four bits of the SDI data values. We use symbolic variable names to make the code easy to read.

2.Shift_reg <= {cmd, data_in, 2'b00} - parallel load the 16-bit value into the shift register.

3.Shift_reg <= {shift_reg[14:0], 1'b0} - perform left shift

The SDI is taken from the MSB of the shift register. The serial clock Is !dac_cs (low active) ANDed with the inverter version of the clock (making the rising edge of the SCK signal in the middle of the data bit).

Part 3 (ex10 & 11) - Testing DAC, SPI and PWM



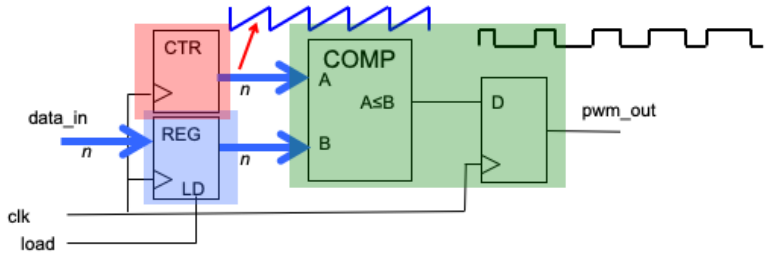
- Use the 10 slider switches to set data value to converter to analogue voltage
- Continuously loading the switch value to DAC at 10KHz rate
- You need: clktick_16, pwm and spi2dac

PYKC 19 Nov 2019 E2.1 Digital Electronics Lecture 12 Slide 13

In the Lab experiment, you will test the spi2dac.v module both with the simulator and on the hardware (with a scope) by inspecting the output signals on the test pins (located at the top of the I/O board).

Ex 10 and 11 are simple, but will give you confidence that the interface module works.

ex 11 - PWM DAC in Verilog (Lecture 9, slide 15)




```
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 19 Nov 2019 E2.1 Digital Electronics Lecture 12 Slide 14

Here is the same slide as that found in Lecture 9. Just a reminder for you on how the PWM module works. It is very simple, but very effective. You should compare the DAC output and PWM output, and see that the two methods are equally effective in producing an analogue voltage.