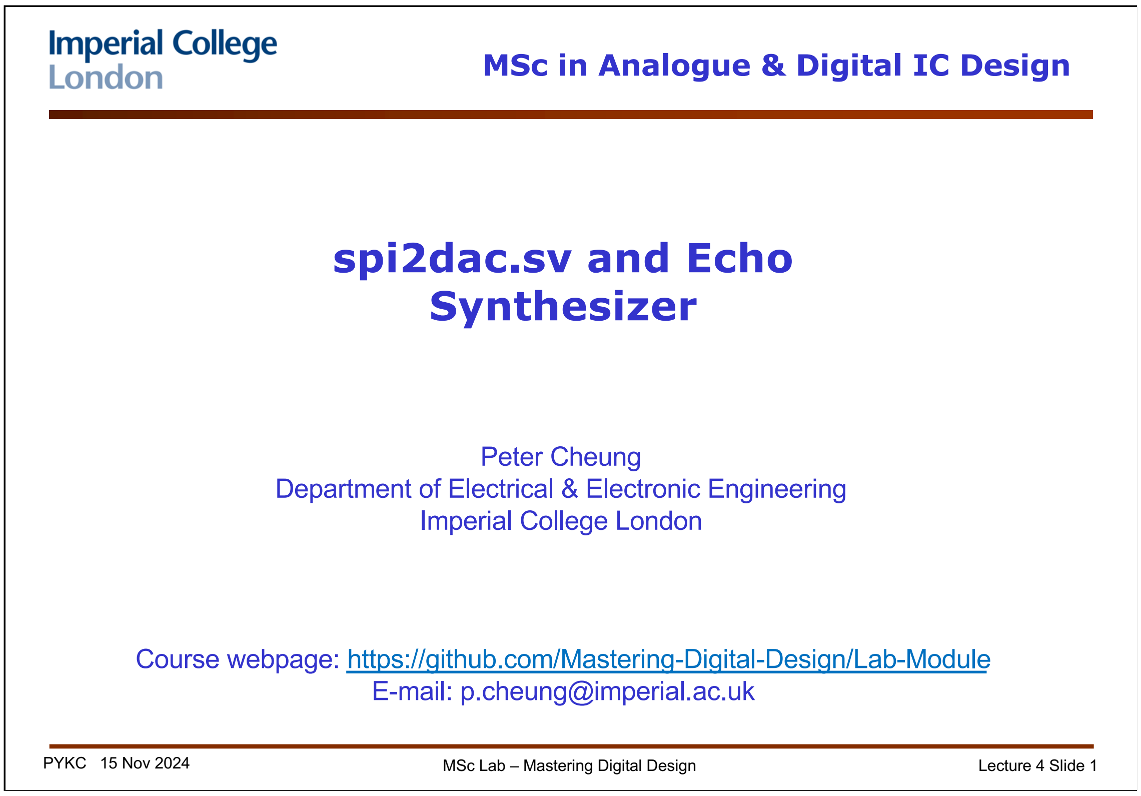


Imperial College London

MSc in Analogue & Digital IC Design

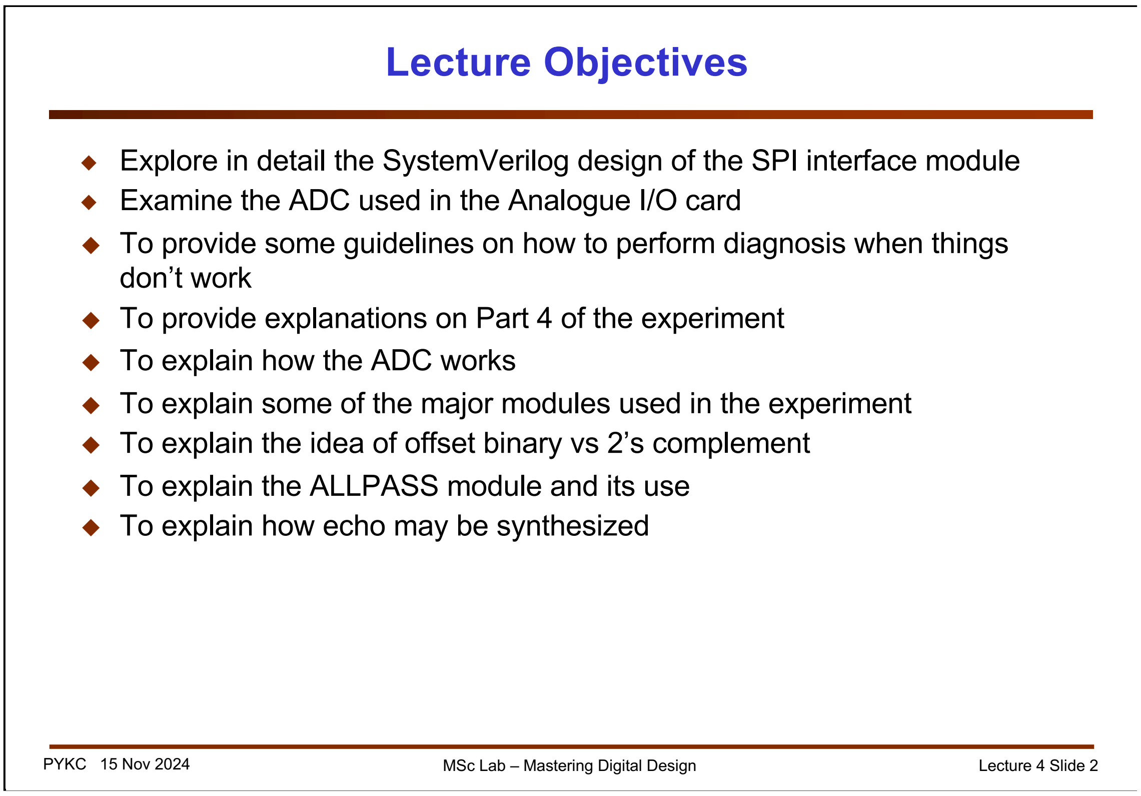
spi2dac.sv and Echo Synthesizer

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.

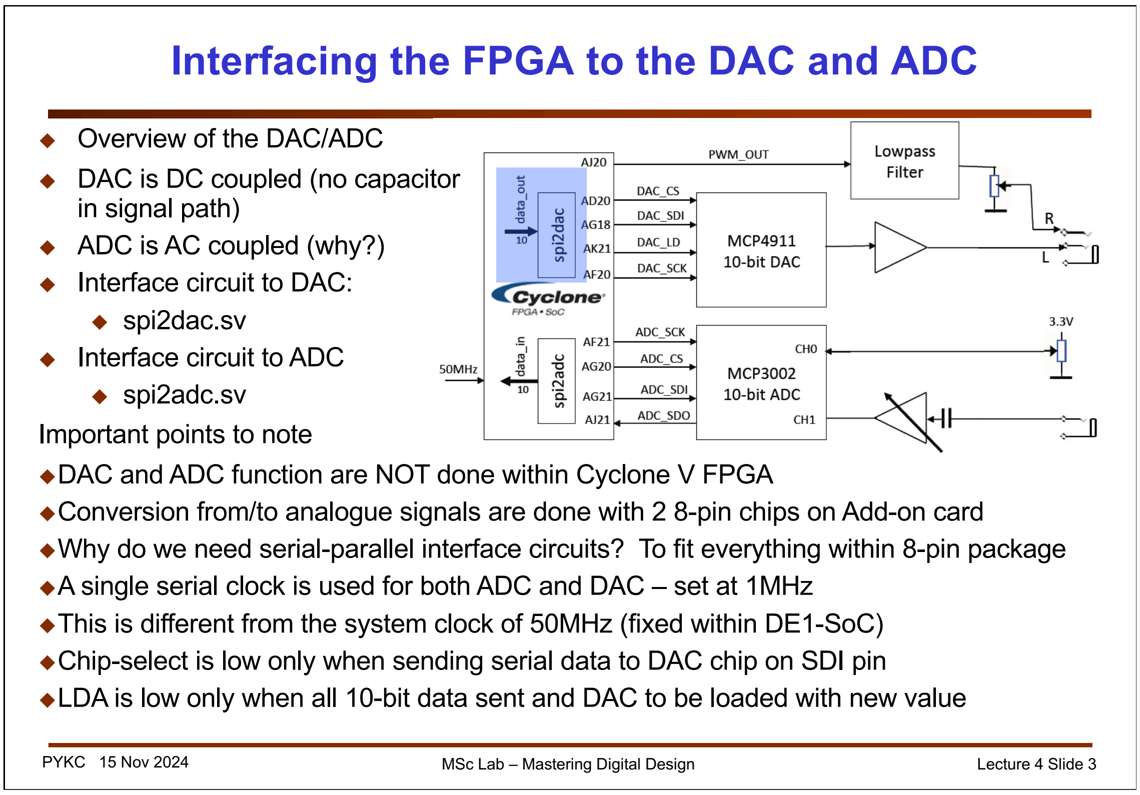
In part 3, you are going to use the FPGA to interface with the external world through a DAC and a ADC on the add-on card. You will also learn about FSM design and PWM module. Finally the DAC and ADC use a serial interface known as SPI. We will take a brief look at this interface standard without going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



Lecture Objectives

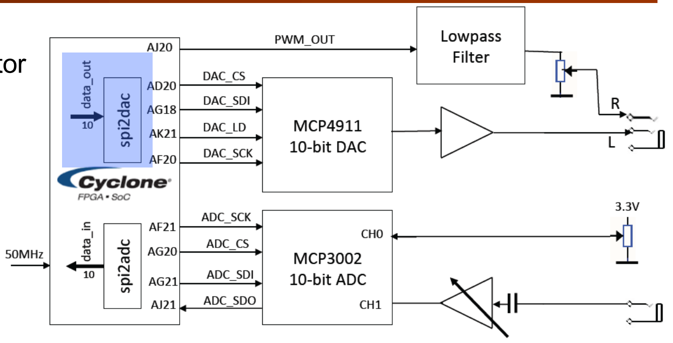
- Explore in detail the SystemVerilog design of the SPI interface module
- Examine the ADC used in the Analogue I/O card
- To provide some guidelines on how to perform diagnosis when things don't work
- To provide explanations on Part 4 of the experiment
- To explain how the ADC works
- To explain some of the major modules used in the experiment
- To explain the idea of offset binary vs 2's complement
- To explain the ALLPASS module and its use
- To explain how echo may be synthesized

This lecture is designed to complement part 3 & 4 of the experiment.



Interfacing the FPGA to the DAC and ADC

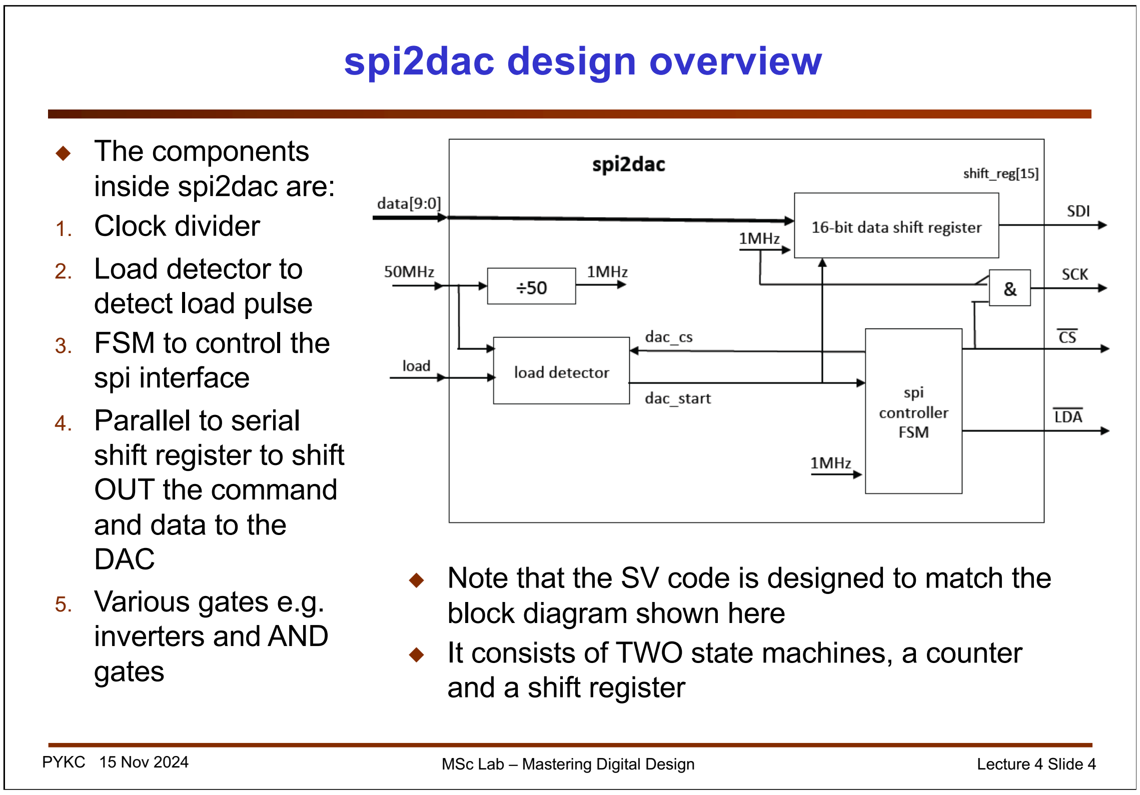
- Overview of the DAC/ADC
- DAC is DC coupled (no capacitor in signal path)
- ADC is AC coupled (why?)
- Interface circuit to DAC:
  - spi2dac.sv
- Interface circuit to ADC
  - spi2adc.sv



Important points to note

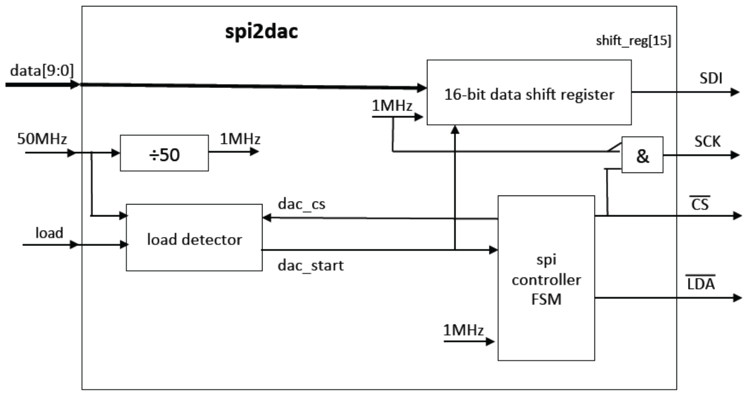
- DAC and ADC function are NOT done within Cyclone V FPGA
- Conversion from/to analogue signals are done with 2 8-pin chips on Add-on card
- Why do we need serial-parallel interface circuits? To fit everything within 8-pin package
- A single serial clock is used for both ADC and DAC – set at 1MHz
- This 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.



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

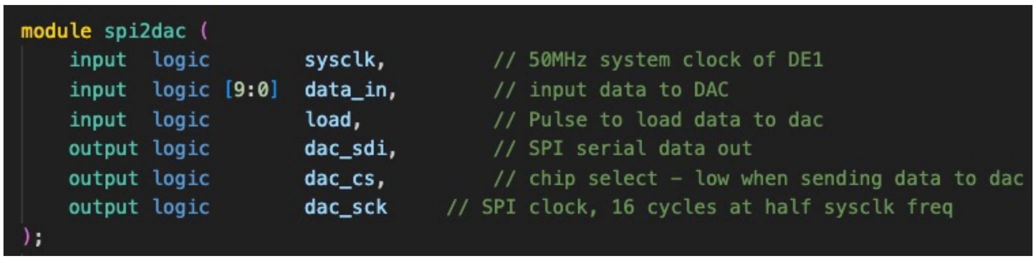


- Note that the SV code is designed to match the block diagram shown here
- It consists of TWO state machines, a counter and a shift register

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.

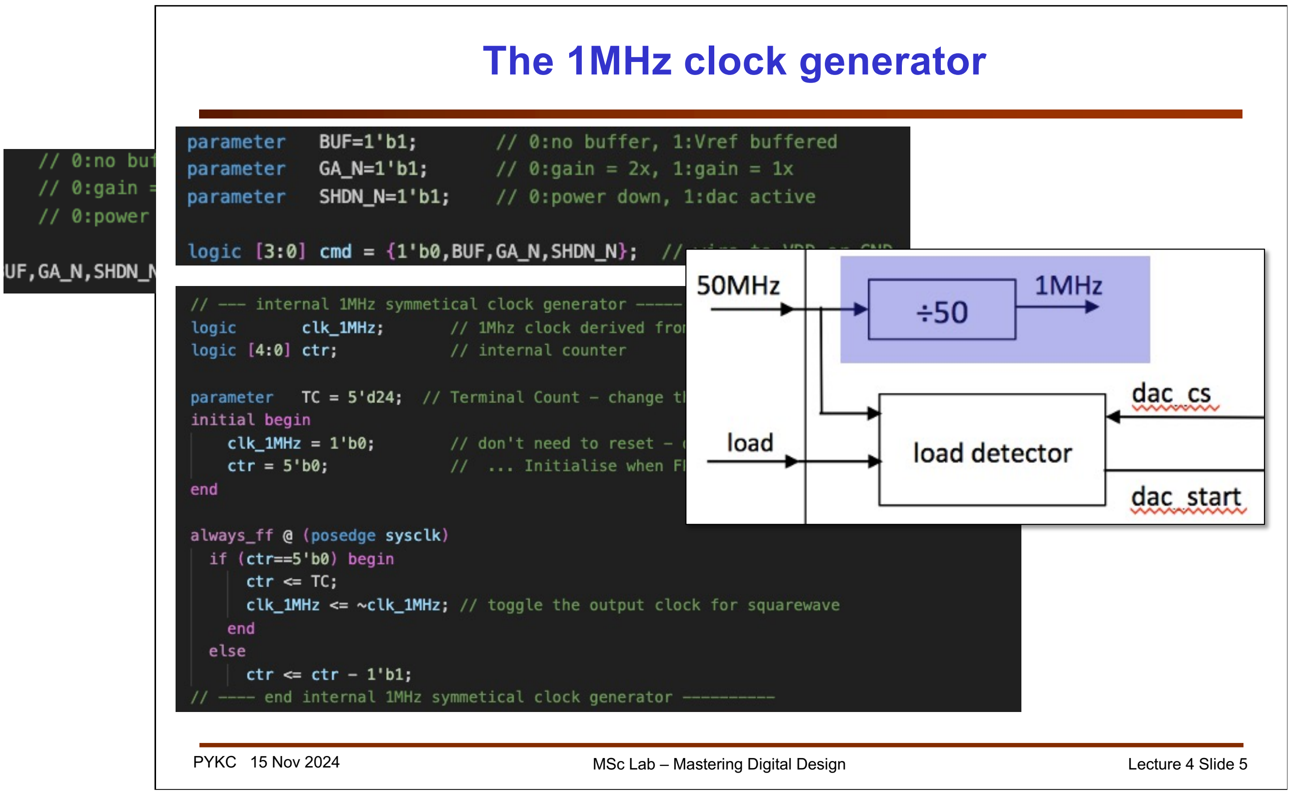
The internal working of sp2dac can be divided into 4 main modules. The divide-by-50 module is straight forward – it produces a 1MHz clock for the finite state machine, and is gated through the AND gate to generate the serial clock signal (at 1MHz).
The load detector module handles the load command and 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.

The spi2dac.sv interface declaration is:



```
module spi2dac (
    input   logic        sysclk,         // 50MHz system clock of DE1
    input   logic [9:0]  data_in,        // input data to DAC
    input   logic        load,           // Pulse to load data to dac
    output  logic        dac_sdi,        // SPI serial data out
    output  logic        dac_cs,         // chip select - low when sending data to dac
    output  logic        dac_sck      // SPI clock, 16 cycles at half sysclk freq
);
```

We will now consider each sub-module individually.

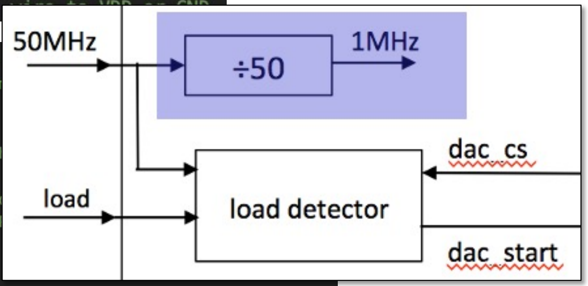


```
  // 0:no bu
  // 0:gain =
  // 0:power

UF,GA_N,SHDN_N
```

The 1MHz clock generator

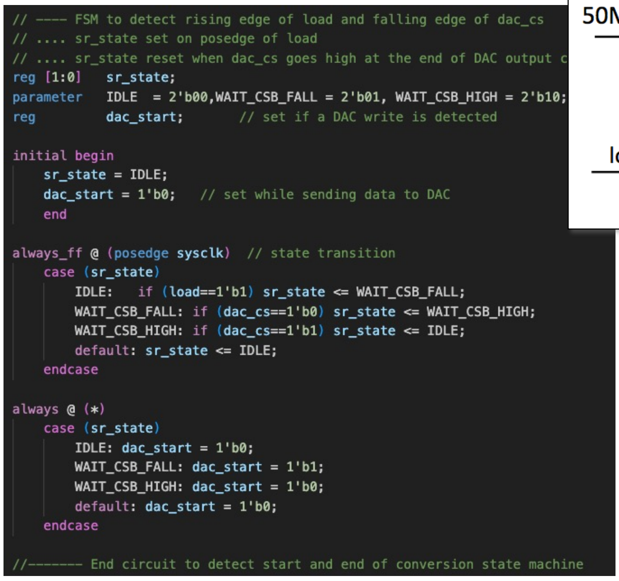
```
parameter   BUF=1'b1;        // 0:no buffer, 1:Vref buffered
parameter   GA_N=1'b1;       // 0:gain = 2x, 1:gain = 1x
parameter   SHDN_N=1'b1;     // 0:power down, 1:dac active

logic [3:0] cmd = {1'b0,BUF,GA_N,SHDN_N};  //
```

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

parameter   TC = 5'd24;  // Terminal Count - change t
initial begin
    clk_1MHz = 1'b0;         // don't need to reset - 
    ctr = 5'b0;              //  ... Initialise when F
end

always_ff @ (posedge sysclk)
  if (ctr==5'b0) begin
      ctr <= TC;
      clk_1MHz <= ~clk_1MHz; // toggle the output clock for squarewave
    end
  else
      ctr <= ctr - 1'b1;
// ---- end internal 1MHz symmetical clock generator ----------
```

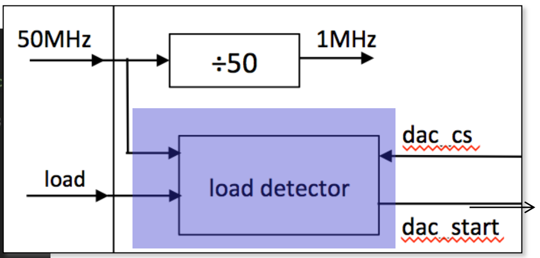


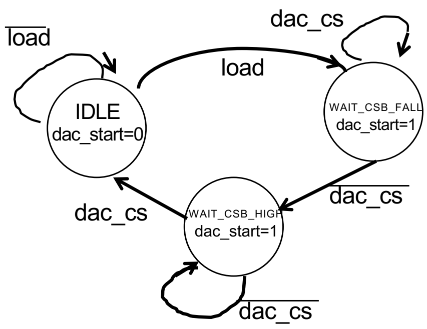
This is a straight forward clock divider. The Terminal Count (TC) is set to 24. Divide by 50 is done by toggling the output (clk_1MHz) after 25 clock cycles. Note that I generally prefer to use a down-counter instead of an up-counter. The counter (ctr) is initialised to 24, it then counts to zero. Output 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 edge of dac_cs
// .... sr_state set on posedge of load
// .... sr_state reset when dac_cs goes high at the end of DAC output c
reg [1:0]   sr_state;
parameter   IDLE  = 2'b00,WAIT_CSB_FALL = 2'b01, WAIT_CSB_HIGH = 2'b10;
reg         dac_start;       // set if a DAC write is detected

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

always_ff @ (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
```



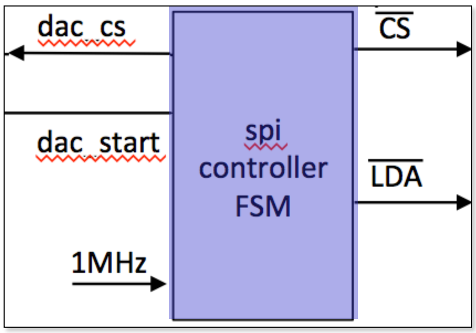


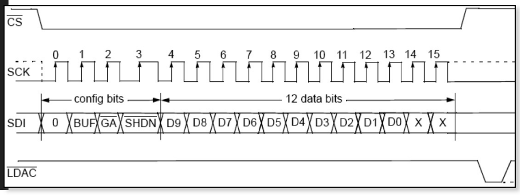
We have TWO signals to detect: the load pulse and the dac_cs signal.

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_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 next state
    endcase

always @ (*)    begin            // FSM output
    dac_cs = 1'b0;
    case (state)
        5'd0:    dac_cs = 1'b1;
        5'd17:   dac_cs = 1'b1;
        default: dac_cs = 1'b0;
        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.

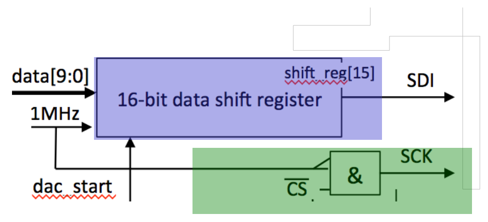
The default value of dac_cs and dac_ld are specified first. By default we always go to the next state, i.e. state value goes up by 1.

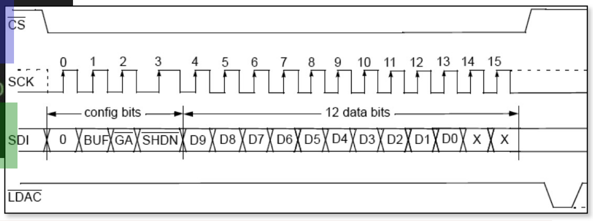
Once the state machine moves to state 1, it just go through to state 16, which corresponds to cycle 0 to 15 in the timing diagram here. At the end of state 16, we de-assert dac_cs (i.e. go high), assert dac_ld (low) and go back to the IDLE state.

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

logic [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 D
        assign dac_sck = !clk_1MHz&!dac_cs;
        assign dac_sdi = shift_reg[15];
```

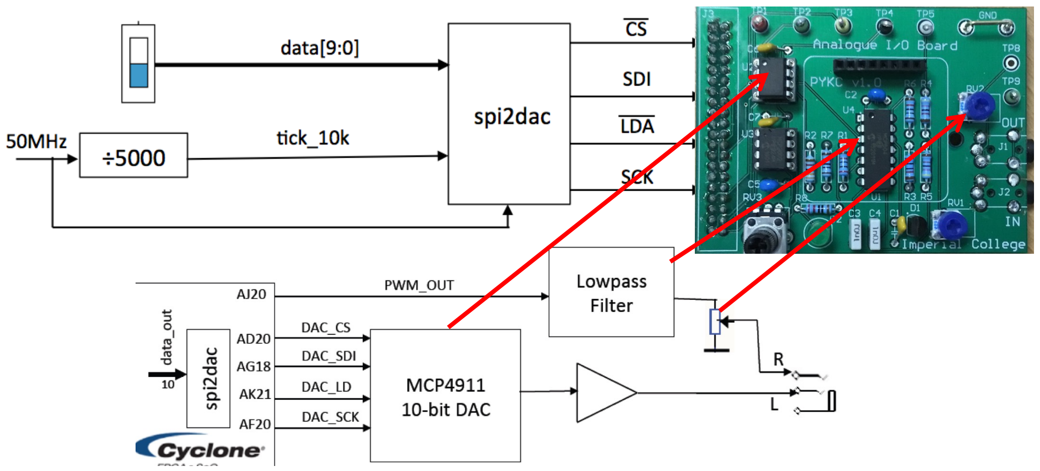




Finally, the data and clock output is specified here. SDI is driven through a parallel in, serial out shift register.
We use a number of useful tricks here:
1.cmd is a 4-bit value defining the first four bits of the SDI data values. We use symbolic variable names to make the code easy to read.
2.Shift_reg <= {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).

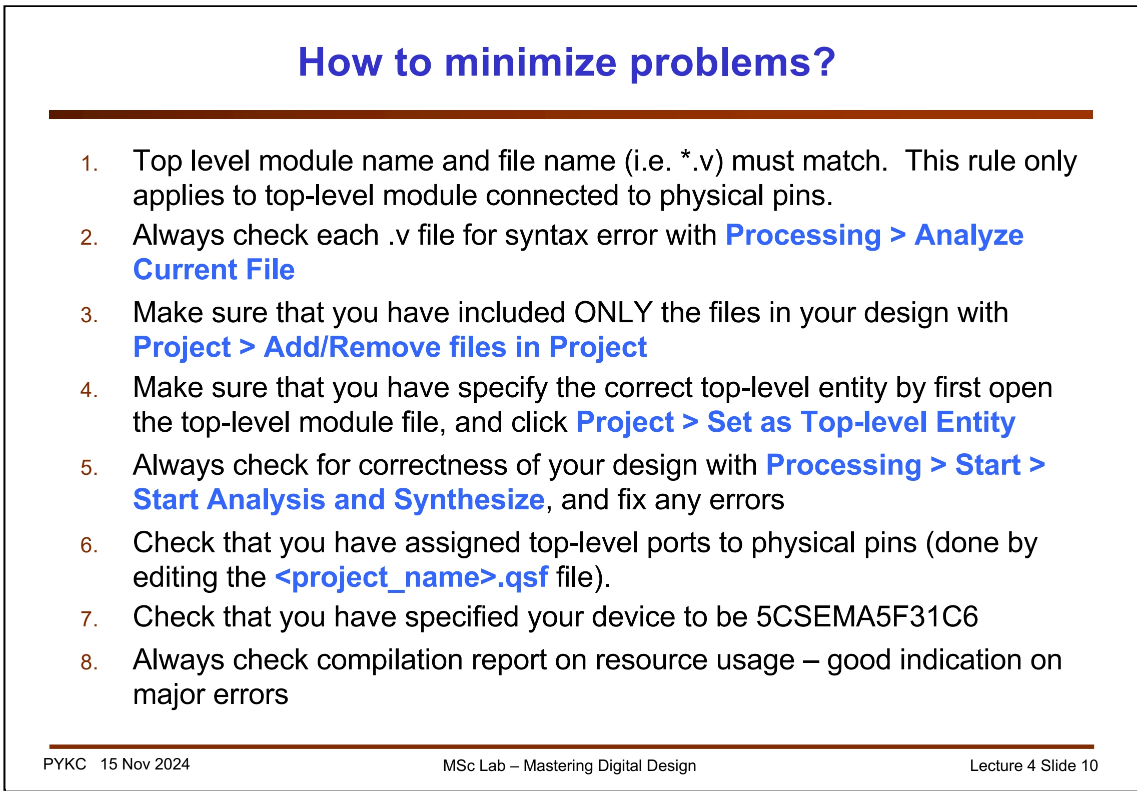
Part 3 (ex10 & 11) - Testing DAC, SPI and PWM



- 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

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.

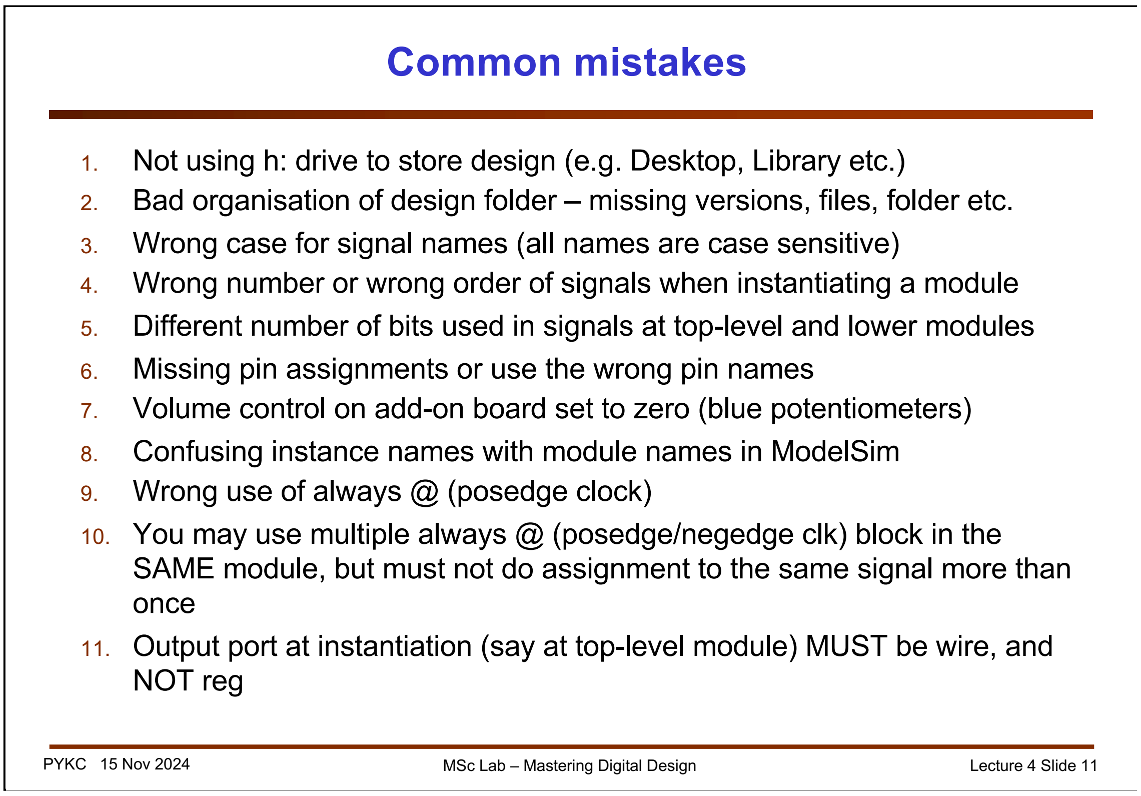


How to minimize problems?

1. Top level module name and file name (i.e. *.v) must match. This rule only applies to top-level module connected to physical pins.
2. Always check each .v file for syntax error with **Processing > Analyze Current File**
3. Make sure that you have included ONLY the files in your design with **Project > Add/Remove files in Project**
4. Make sure that you have specify the correct top-level entity by first open the top-level module file, and click **Project > Set as Top-level Entity**
5. Always check for correctness of your design with **Processing > Start > Start Analysis and Synthesize**, and fix any errors
6. Check that you have assigned top-level ports to physical pins (done by editing the **<project_name>.qsf** file).
7. Check that you have specified your device to be 5CSEMA5F31C6
8. Always check compilation report on resource usage – good indication on major errors

PYKC 15 Nov 2024 MSc Lab – Mastering Digital Design Lecture 4 Slide 10

This slide is self explanatory. These are some steps you should take in order to minimize problems that you may encounter.



Common mistakes

1. Not using h: drive to store design (e.g. Desktop, Library etc.)
2. Bad organisation of design folder – missing versions, files, folder etc.
3. Wrong case for signal names (all names are case sensitive)
4. Wrong number or wrong order of signals when instantiating a module
5. Different number of bits used in signals at top-level and lower modules
6. Missing pin assignments or use the wrong pin names
7. Volume control on add-on board set to zero (blue potentiometers)
8. Confusing instance names with module names in ModelSim
9. Wrong use of always @ (posedge clock)
10. You may use multiple always @ (posedge/negedge clk) block in the SAME module, but must not do assignment to the same signal more than once
11. Output port at instantiation (say at top-level module) MUST be wire, and NOT reg

PYKC 15 Nov 2024 MSc Lab – Mastering Digital Design Lecture 4 Slide 11

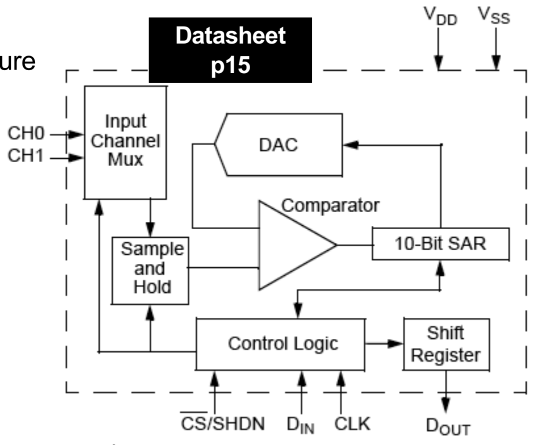
Here is a list of common mistakes students had in the lab.

ADC – used in add-on card

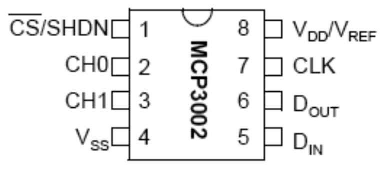
- **Microchip MCP3002** 10-bit ADC
- Uses **successive approximation** architecture
- Serial Peripheral Interface (SPI)

- Analog inputs programmable as single-ended or pseudo-differential pairs
- On-chip sample and hold
- SPI serial interface (modes 0,0 and 1,1)
- Single supply operation: 2.7V - 5.5V
- 200 ksps max sampling rate at V_{DD} = 5V
- 75 ksps max sampling rate at V_{DD} = 2.7V

Datasheet p15



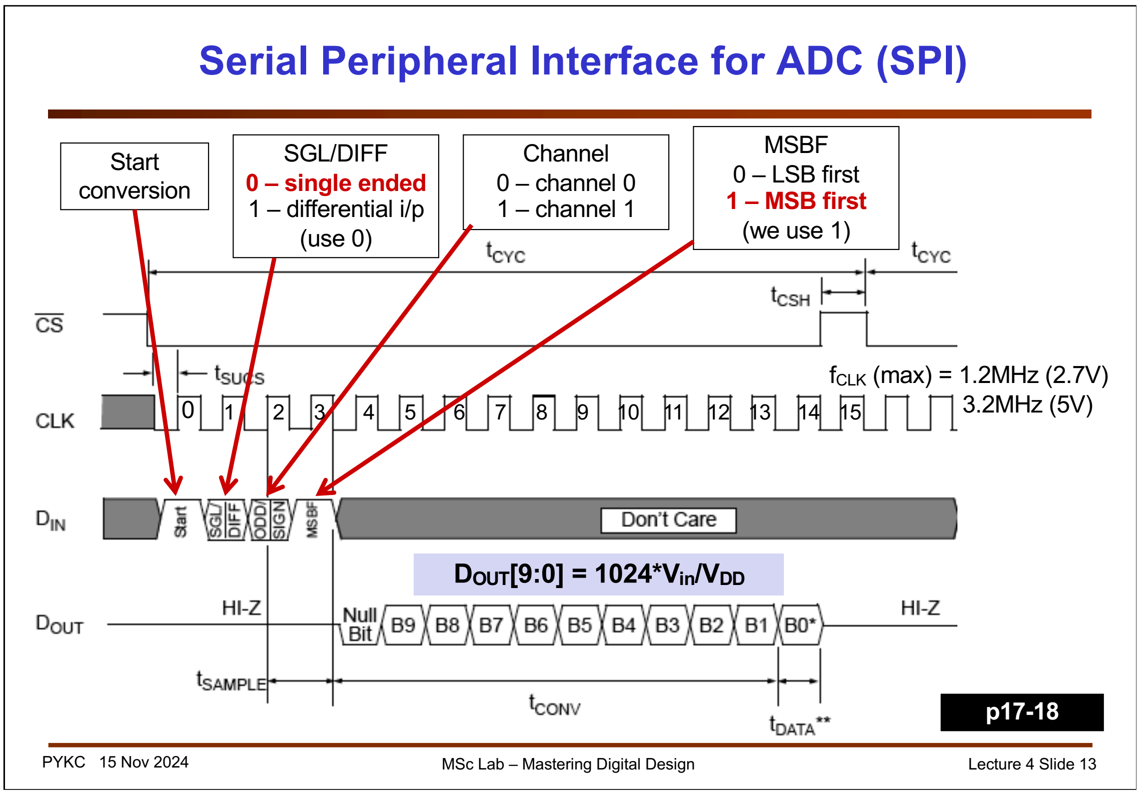
Symbol	Description
$\overline{CS}$/SHDN	Chip Select/Shutdown Input
CH0	Channel 0 Analog Input
CH1	Channel 1 Analog Input
V_{SS}	Ground
D_{IN}	Serial Data In
D_{OUT}	Serial Data Out
CLK	Serial Clock
V_{DD}/V_{REF}	+2.7V to 5.5V Power Supply and Reference Voltage Input



PYKC 15 Nov 2024 MSc Lab – Mastering Digital Design Lecture 4 Slide 12

This shows the ADC block diagram. Again the digital interface obeys the SPI protocol, with Chip Select (CS), Serial Clock (CLK), Serial Data in (Din) and Serial Data Out (Dout) signals.

This ADC uses a 10-bit DAC internally, and the successive approximate algorithm (SAR) as described in our earlier lecture on ADCs.

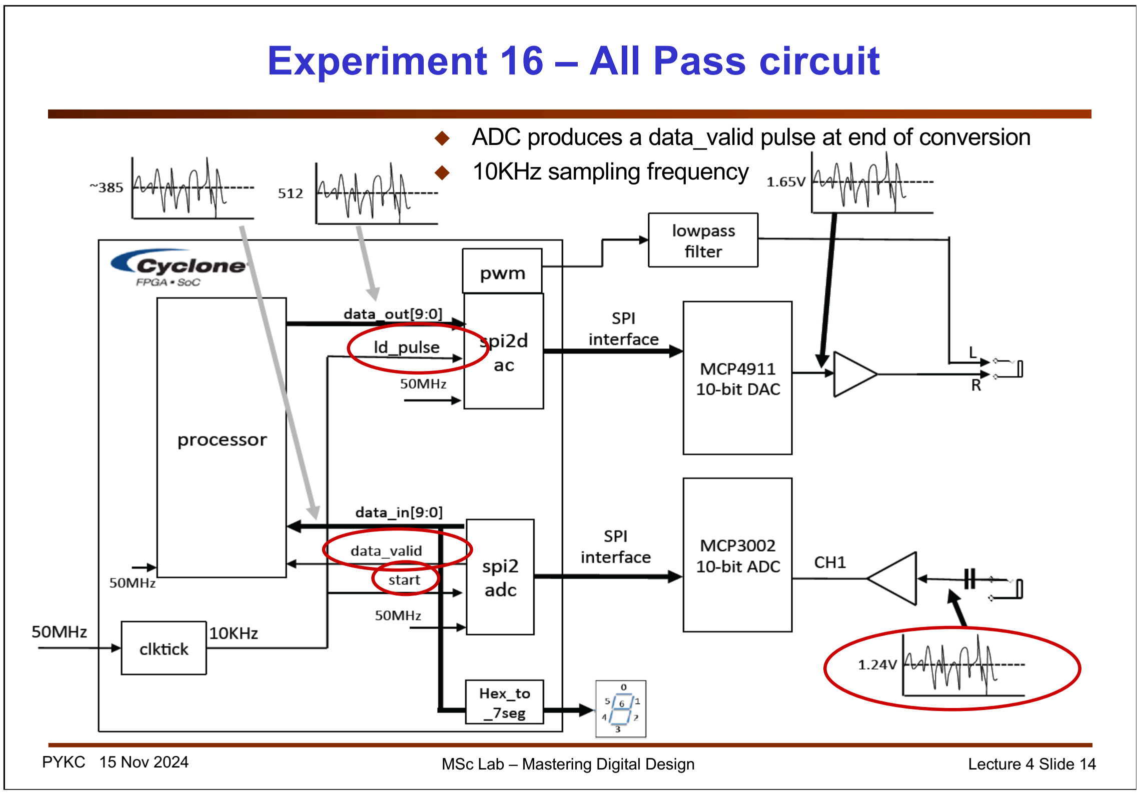


Serial Peripheral Interface for ADC (SPI)

The control of the ADC is slightly more complicated than that for the DAC. Nevertheless, the idea is similar. The transfer cycle is again 16 states, going from state 0 to state 15.

Conversion is started with Chip Select going low, and Din bit 15 = ‘1’. The next bit to Din specifies whether the analogue signal is single ended or differential. (We use single-ended for our experiment.)
The next bit selects channel 0 or 1, followed by specifying data to be returned least-significant bit first or most-significant bit first. We use MSB first.

After these four “setup” bits are sent to the ADC, it returns 11 bits to Dout. First bit is always 0. Then the next 10 bits are the converted data MSB first.

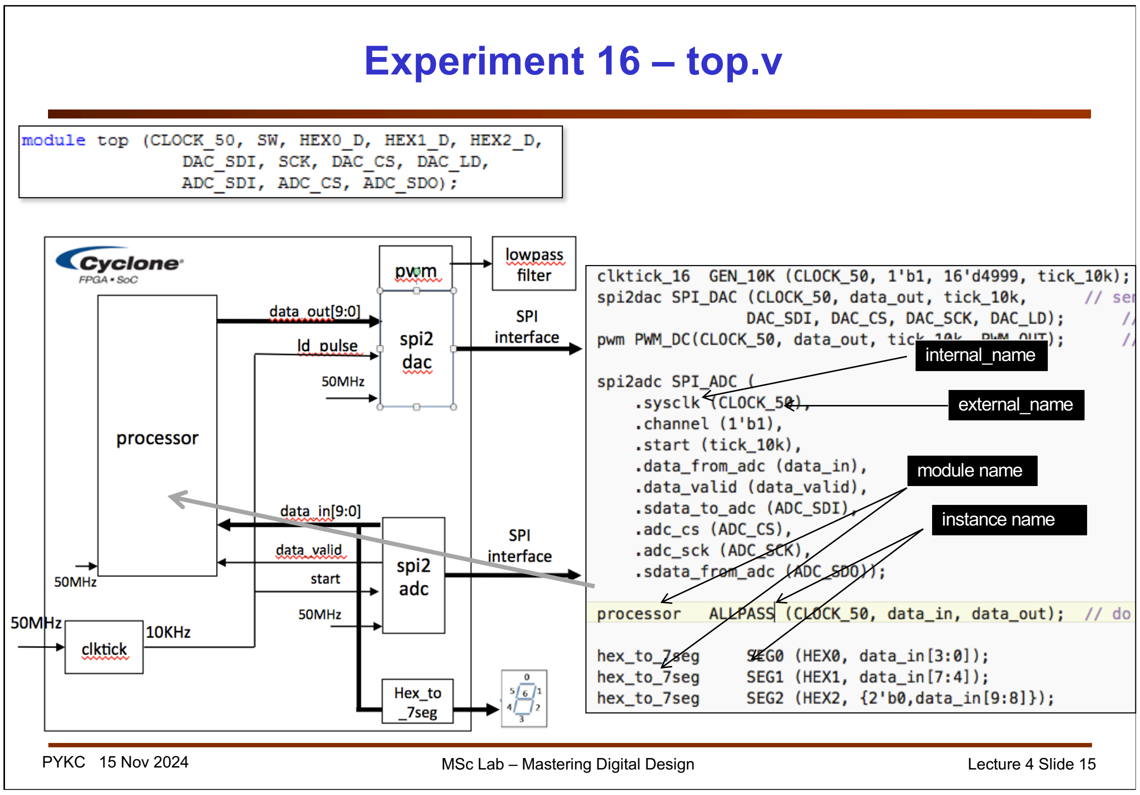


Experiment 16 – All Pass circuit

- ADC produces a data_valid pulse at end of conversion
- 10KHz sampling frequency

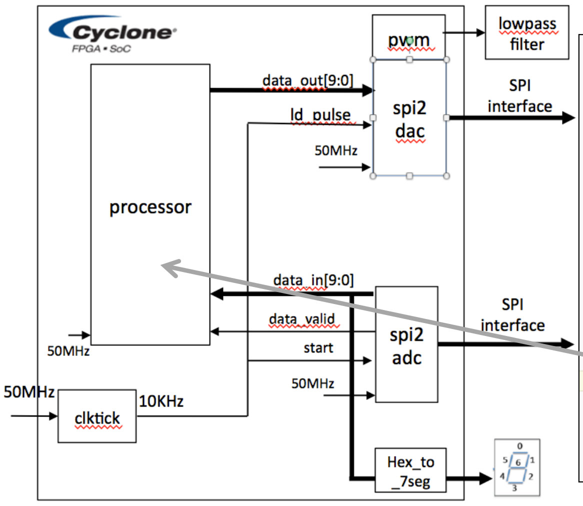
PYKC 15 Nov 2024 MSc Lab – Mastering Digital Design Lecture 4 Slide 14

This is the block diagram of the basic framework used for Part 4 of VERI. The two main modules spi2dac.v and spi2adc.v provide interfacing to the DAC and ADC respectively. The control circuit is simple – a clock tick circuit generating a 10 KHz sampling clock The 7 segment displays can be used to monitor the ADC converted data.



Experiment 16 – top.v

```
module top (CLOCK_50, SW, HEX0_D, HEX1_D, HEX2_D,
            DAC_SDI, SCK, DAC_CS, DAC_LD,
            ADC_SDI, ADC_CS, ADC_SDO);
```

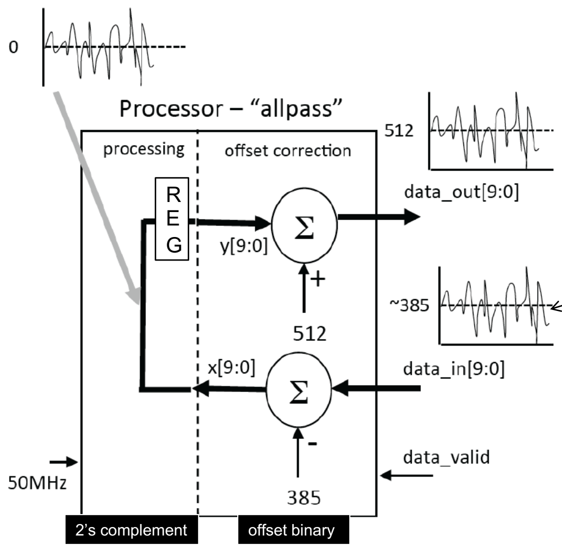


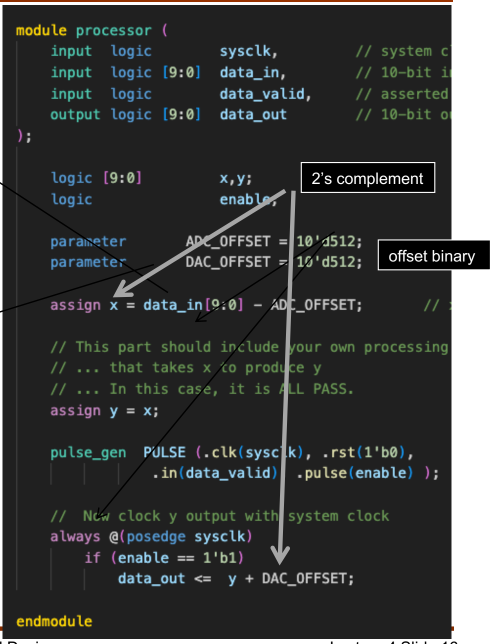
```
clktick_16  GEN_10K (CLOCK_50, 1'b1, 16'd4999, tick_10k);
spi2dac SPI_DAC (CLOCK_50, data_out, tick_10k,      // se
                 DAC_SDI, DAC_CS, DAC_SCK, DAC_LD);     //
pwm PWM_DC(CLOCK_50, data_out, tick_10k, PWM_OUT);      //

spi2adc SPI_ADC (
    .sysclk (CLOCK_50),
    .channel (1'b1),
    .start (tick_10k),
    .data_from_adc (data_in),
    .data_valid (data_valid),
    .sdata_to_adc (ADC_SDI),
    .adc_cs (ADC_CS),
    .adc_sck (ADC_SCK),
    .sdata_from_adc (ADC_SDO));

processor   ALLPASS (CLOCK_50, data_in, data_out);  // do

hex_to_7seg     SEG0 (HEX0, data_in[3:0]);
hex_to_7seg     SEG1 (HEX1, data_in[7:4]);
hex_to_7seg     SEG2 (HEX2, {2'b0,data_in[9:8]});
```

Here is the top level specification connecting all the modules to the FPGA. Here the spi2adc instantiation is done in a verbose, but secure way. Many mistakes happen because the order of signals in the top level is different from that in the module level. Therefore we can associate internal name EXPLICITLY to external name with the syntax: .<internal_name> (external name> as shown above. For example, inside spi2adc, the signal sysclk is connected to the top level CLOCK_50 signal. Now, the order of the signals as used here is irrelevant.

This shows a "processor" module, which in this experiment does an ALL PASS function. That is, it takes a sample from the ADC and immediately send this sample back out to DAC. Therefore everything is simply passed from input to output.

Experiment 16 – allpass.v (offset correction)



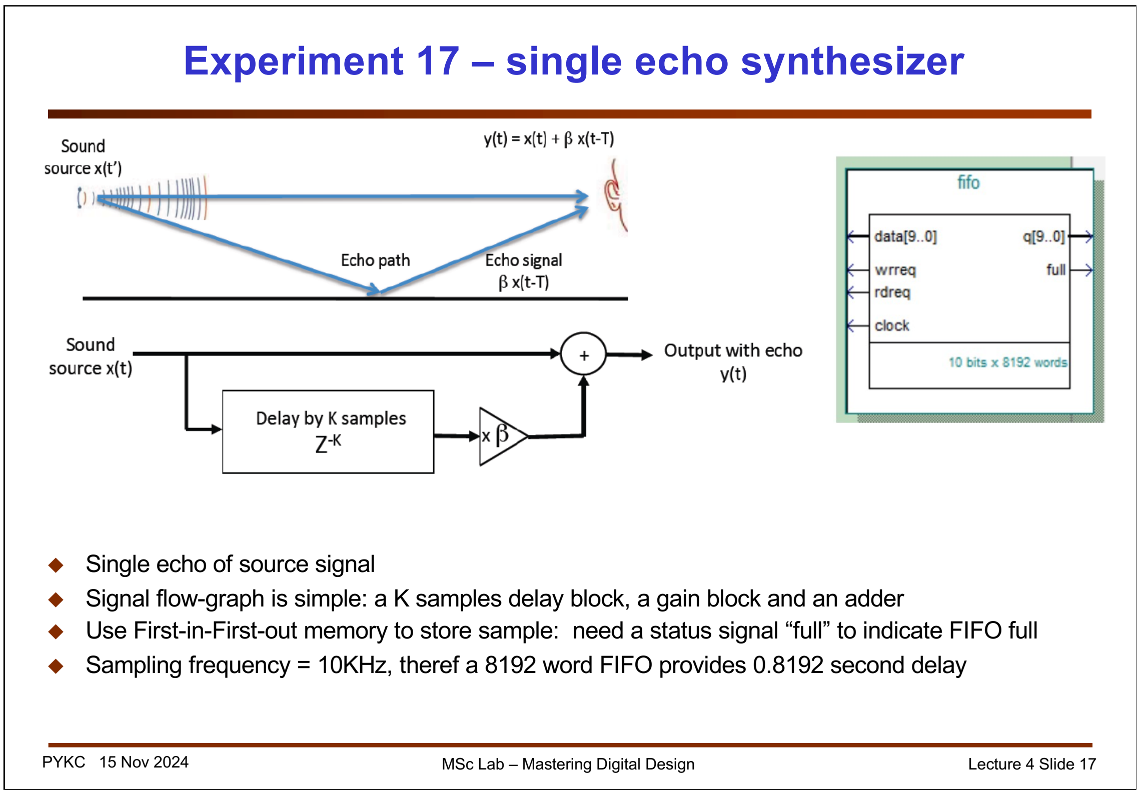


```
module processor (
    input  logic        sysclk,        // system c
    input  logic [9:0]  data_in,       // 10-bit i
    input  logic        data_valid,    // asserted
    output logic [9:0]  data_out       // 10-bit o
);

    logic [9:0]         x,y;
    logic               enable;

    parameter       ADC_OFFSET = 10'd512;
    parameter       DAC_OFFSET = 10'd512;

    assign x = data_in[9:0] - ADC_OFFSET;       // 

    // This part should include your own processing
    // ... that takes x to produce y
    // ... In this case, it is ALL PASS.
    assign y = x;

    pulse_gen  PULSE (.clk(sysclk), .rst(1'b0),
                      .in(data_valid), .pulse(enable) );

    //  Now clock y output with system clock
    always @(posedge sysclk)
        if (enable == 1'b1)
            data_out <=  y + DAC_OFFSET;

endmodule
```

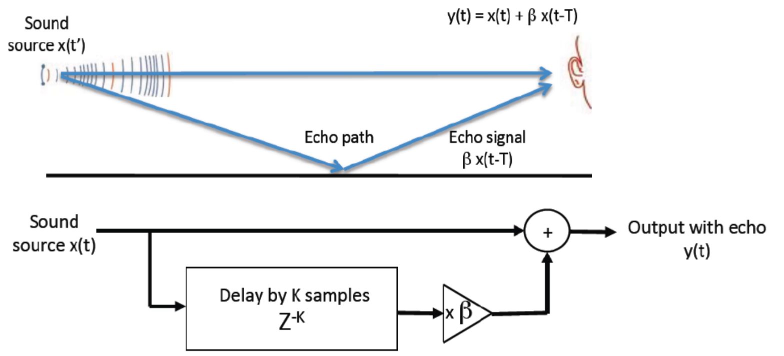
The ALL PASS module is slightly more complex than it may appear. Data_in[9:0] is used to represent the analogue signal input (which is bipolar) as offset binary. There is an offset of around 385 if the input is connect to zero (no signal). The output data_out[9:0] also has an offset. To get Vout = 0V, you need to send the binary number 512.

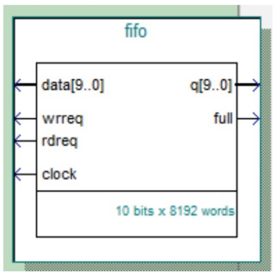
If you are to process the signal using normal arithmetic operators such as +, - and *, you need to use 2’s complement number system. Therefor the ADC data is first offset correct by subtracting the offset 385 from the converted data to yield x[9:0]. The actual processing step is simply the store this data in a register in 2’s complement form. Then the output y[9:0] is again converted back to offset binary for the DAC to output. This is done by adding 512 to y[9:0].

If allpass.v andex16_top.v are both correctly specified, you can send in the ADC a record speech signal via the 2.5mm cable, and hear the same speech using your earphone.



Experiment 17 – single echo synthesizer





- Single echo of source signal
- Signal flow-graph is simple: a K samples delay block, a gain block and an adder
- Use First-in-First-out memory to store sample: need a status signal "full" to indicate FIFO full
- Sampling frequency = 10KHz, theref a 8192 word FIFO provides 0.8192 second delay

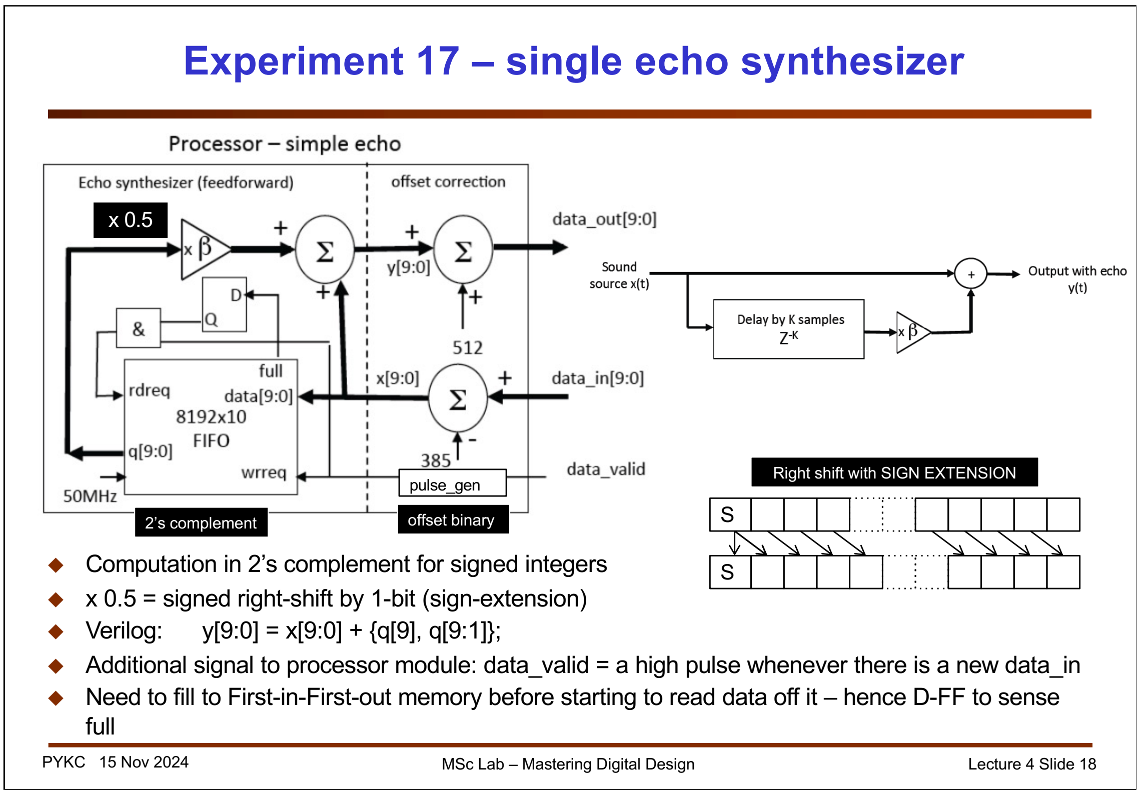
PYKC 15 Nov 2024 MSc Lab – Mastering Digital Design Lecture 4 Slide 17

The final compulsory exercise is to create an echo synthesizer. The basic idea is simple: an echo is recreated when the listener receive the source signal via a direct path AND a delayed echo path as shown.

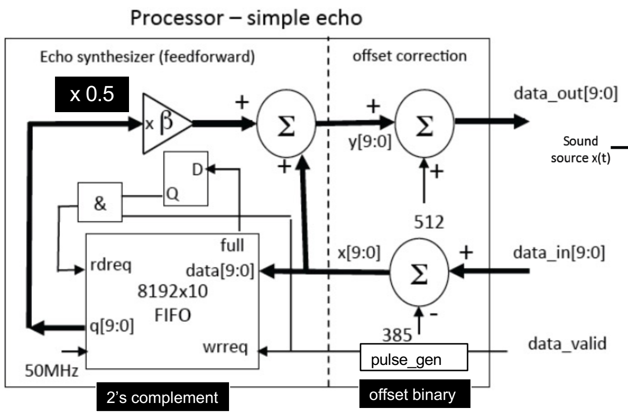
In order for this to work, we need a delay component in the FPGA system. The easiest way to achieve this is to use a first-in-first-out (FIFO). I will explain exactly what a FIFO is in a later lecture. For now it is sufficient for you to know that a FIFO block has data[9:0] as input, and q[9:0] as output. The first sample that goes in is the sample the first sample that comes out. There is a write request signal wrreq which is asserted when you want to write a word into the FIFO. Similar a rdreq signal is asserted when you want to read a word out from the FIFO. There is a synchronising clock signal.

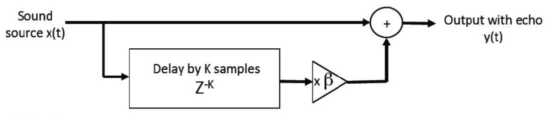
Finally if the FIFO is full (in this case storing 8192 samples already), then the full signal goes high.

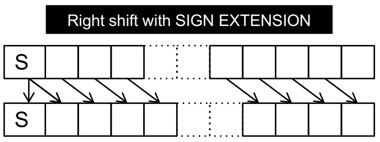
This FIFO will provide 0.8192 second delay if the sampling clock is 10KHz.



Experiment 17 – single echo synthesizer





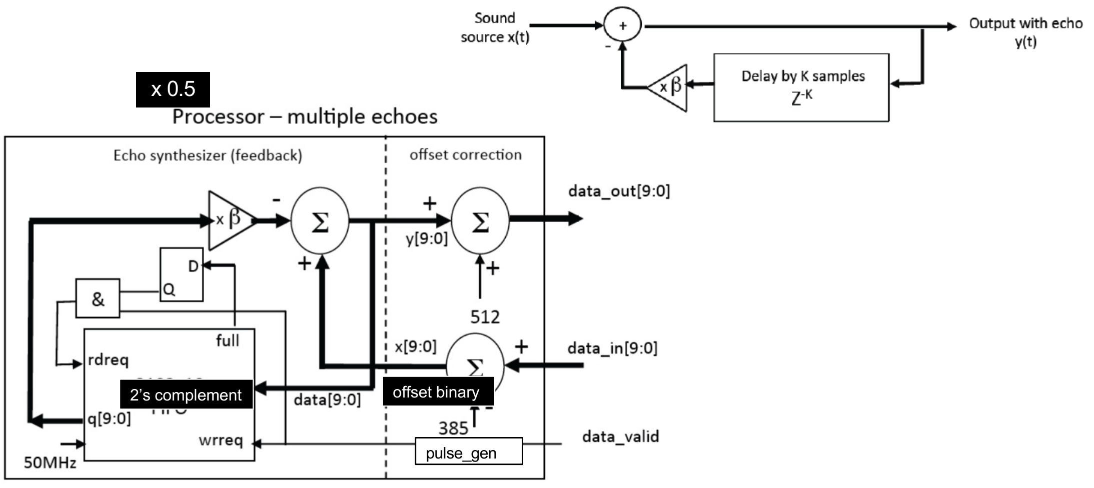


- Computation in 2's complement for signed integers
- x 0.5 = signed right-shift by 1-bit (sign-extension)
- Verilog: y[9:0] = x[9:0] + {q[9], q[9:1]};
- Additional signal to processor module: data_valid = a high pulse whenever there is a new data_in
- Need to fill to First-in-First-out memory before starting to read data off it – hence D-FF to sense full

PYKC 15 Nov 2024 MSc Lab – Mastering Digital Design Lecture 4 Slide 18

Here is the block diagram of the processor module for a single echo synthesier. The FIFO control circuit is quite simple, the D-FF and the AND gate ensure that during initial operation, the FIFO is only written to until it is completely filled. Initially, DFF is '0' because full is '0'. The AND gate block sthe data_valid pulse from the ADC. Therefore for the first 9192 conversions, the FIFO is only written to, and nothing is taken off it. When the FIFO is full, Q of DFF goes high, and from now on, every data written into the FIFO, another data value 8192 samples earlier (ie. Z^-8192) is taken off the FIFO as the echo signal. This is then scaled by a constant 0.5 (which is an arithmetic right shift with sign extension).

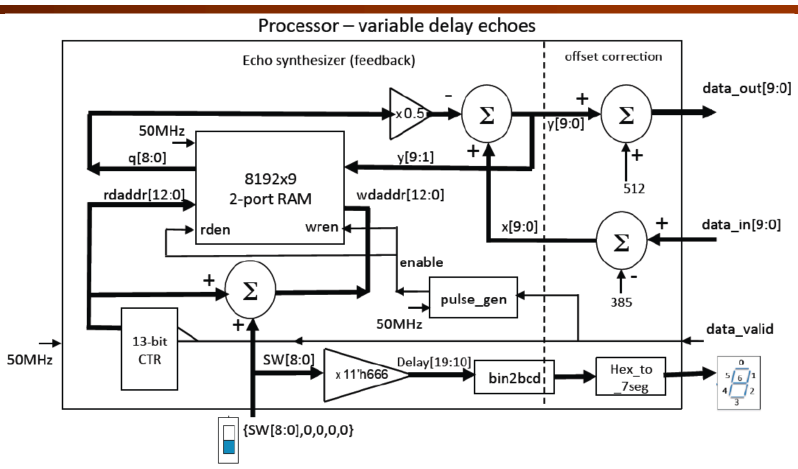
Experiment 18 – multiple echoes synthesizer



- Instead of feedforward only, this uses a feedback loop
- To avoid instability, you must SUBTRACT delayed echo signal instead of add
- FIFO now stores y[9:0] output, and NOT input

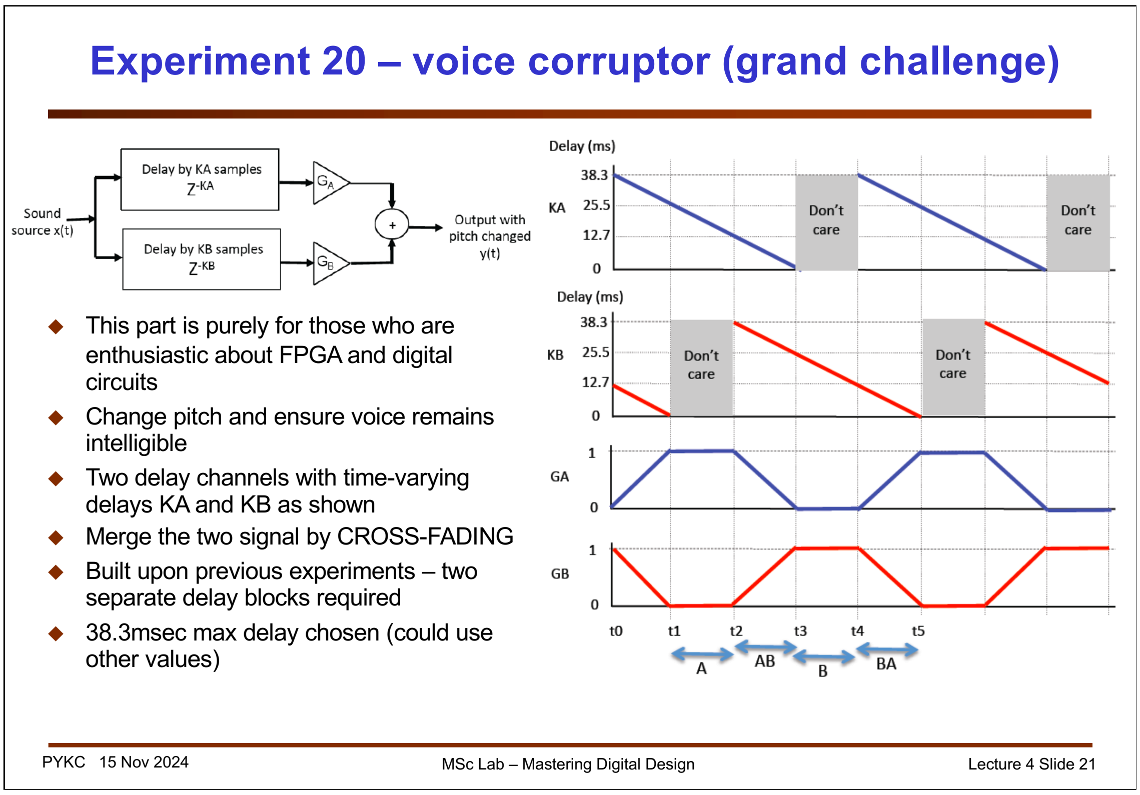
A slight modification create a mult-echo synthesizer. Here we put the delay element in a feedback path. Note that you MUST perform a subtract instead of an add, otherwise the system has positive feedback and will become unstable.

Experiment 19 – variable delay echoes

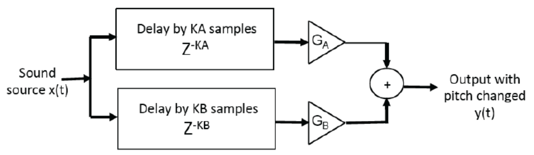


- Entirely optional – do this only if you have time and is truly interested (but at least test my solution)
- Use 2-port RAM instead of FIFO for delay block
- RAM – only 9-bit wide (10-bit not a option), so store most-significant 9 bits y[9:1]
- Write_address = Read_address + delay value from SW[8:0] (SW[9] already used)
- Compute delay in millisecond and display as decimal value

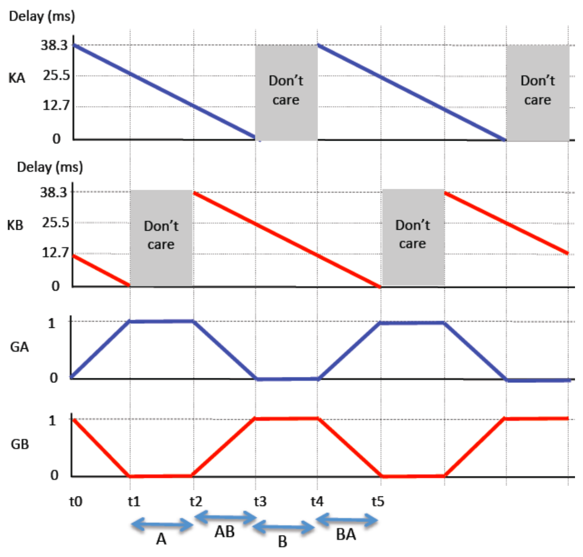
This exercise is optional. Instead of generating the FIFO block using the Megawizard tool in Quartus, you can produce a 2-port RAM and implement your own FIFO. Here we use a 13-bit counter and an adder to produce the read and write addresses. Instead of using the fixed 8192 sample delay, by offseting the counter value with a value from SW[8:0], you can adjust the delay of the echo. There are extra modules here to show the amount of delay as a BCD number on the display.



Experiment 20 – voice corruptor (grand challenge)



Sound source x(t)
Delay by KA samples Z^{-KA}
Delay by KB samples Z^{-KB}
G_A
G_B
+
Output with pitch changed y(t)



- This part is purely for those who are enthusiastic about FPGA and digital circuits
- Change pitch and ensure voice remains intelligible
- Two delay channels with time-varying delays KA and KB as shown
- Merge the two signal by CROSS-FADING
- Built upon previous experiments – two separate delay blocks required
- 38.3msec max delay chosen (could use other values)

PYKC 15 Nov 2024 MSc Lab – Mastering Digital Design Lecture 4 Slide 21

Finally, with minor modifications to the processing module, using TWO delay components and a variable gain (with time), it is possible to produce a pitch changer. There will not be enough time during the experiment for you to do this part. However, it may be a suitable Christmas project.