spi2dac.sv and Echo Synthesizer

Peter Cheung
Department of Electrical & Electronic Engineering
Imperial College London

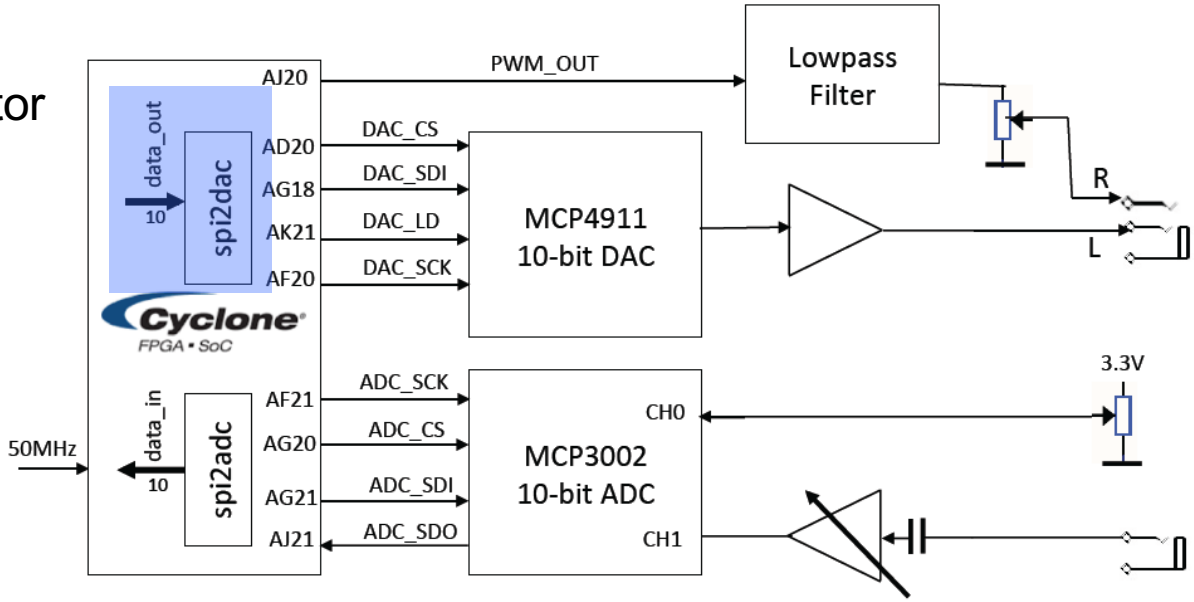
Course webpage: https://github.com/Mastering-Digital-Design/Lab-Module
E-mail: p.cheung@imperial.ac.uk

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

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



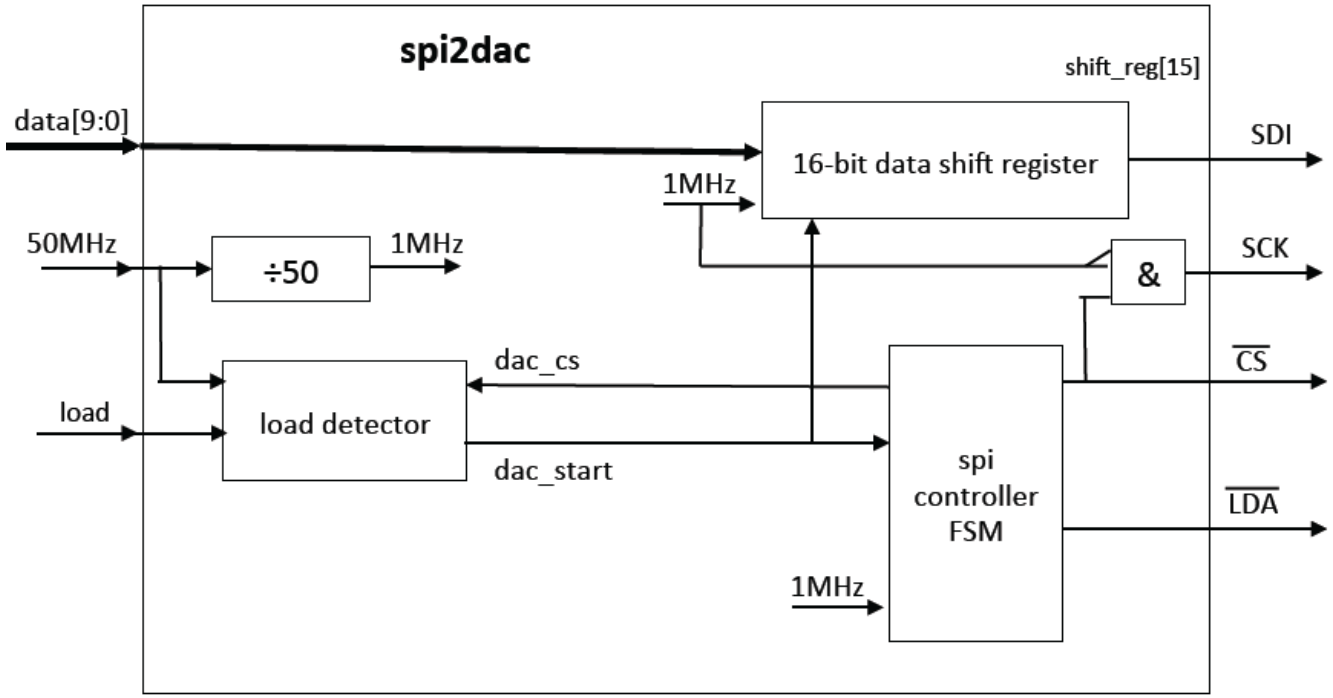
Important points to note

- 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

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

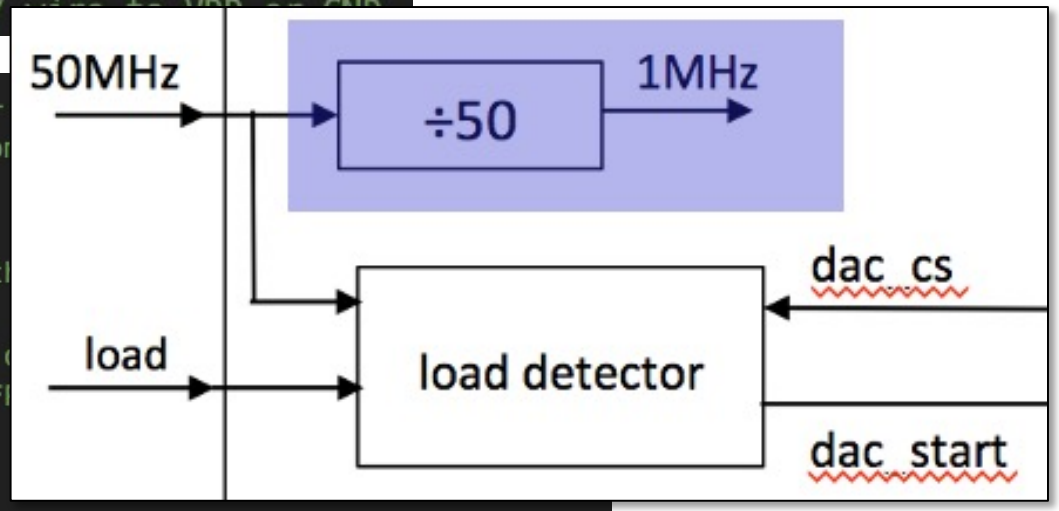


- Note that the SV code is designed to match the block diagram shown here
- It consists of TWO state machines, a counter and a shift register

The 1MHz clock generator

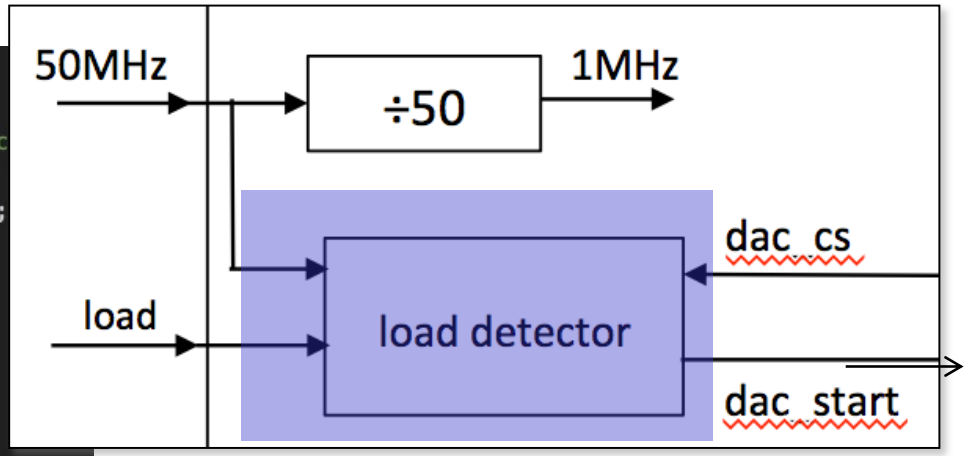
```
parameter    BUF=1'b1;        // 0:no buffer, 1:Vref buffered
parameter    GA_N=1'b1;       // 0:gain = 2x, 1:gain = 1x
parameter    SHDN_N=1'b1;     // 0:power down, 1:dac active

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

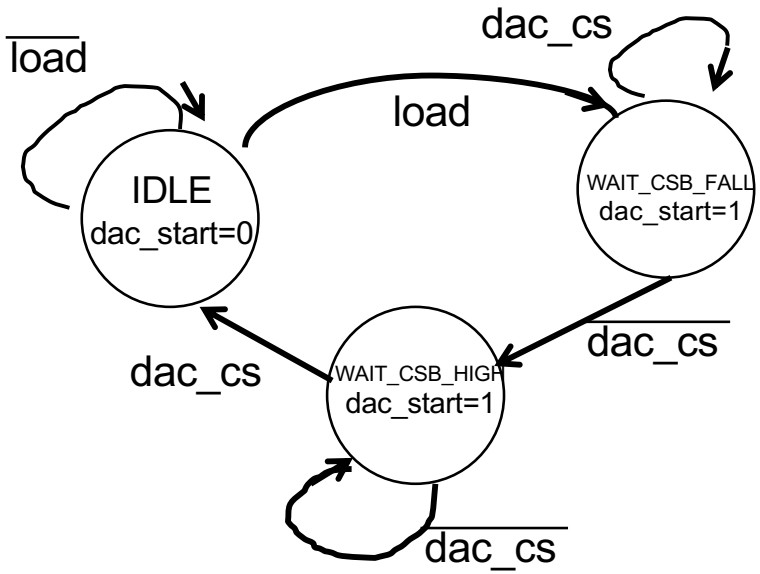
```
// --- internal 1MHz symmetical clock generator -----
logic       clk_1MHz;        // 1Mhz clock derived fro
logic [4:0] ctr;             // internal counter

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 ----------
```



The load pulse detector

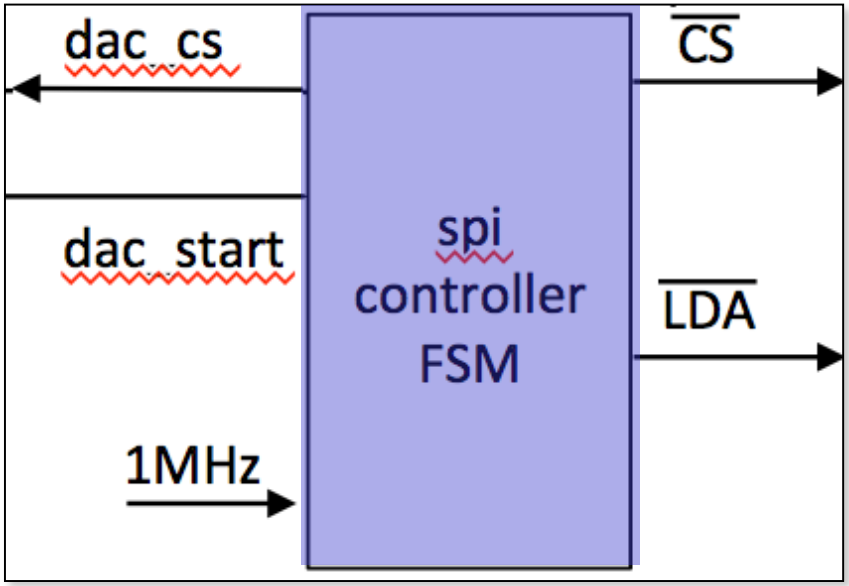
```
// ---- FSM to detect rising edge of load and falling edge of dac_cs
// .... sr_state set on posedge of load
// .... sr_state reset when dac_cs goes high at the end of DAC output c
reg [1:0]   sr_state;
parameter   IDLE  = 2'b00,WAIT_CSB_FALL = 2'b01, WAIT_CSB_HIGH = 2'b10;
reg         dac_start;       // 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
```

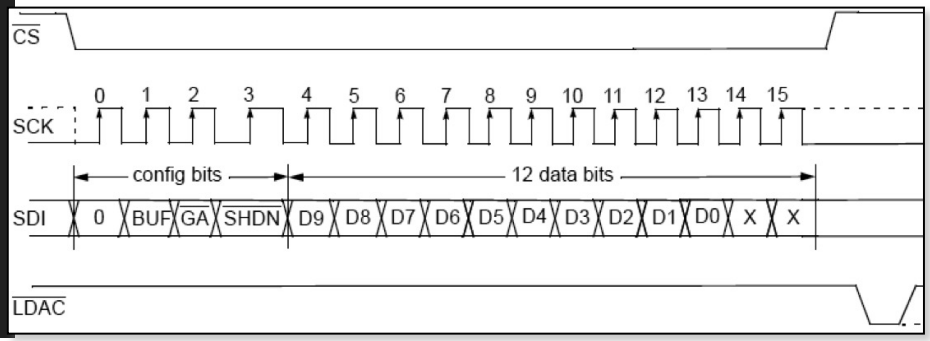




The SPI Controller FSM

```
//------- 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
```

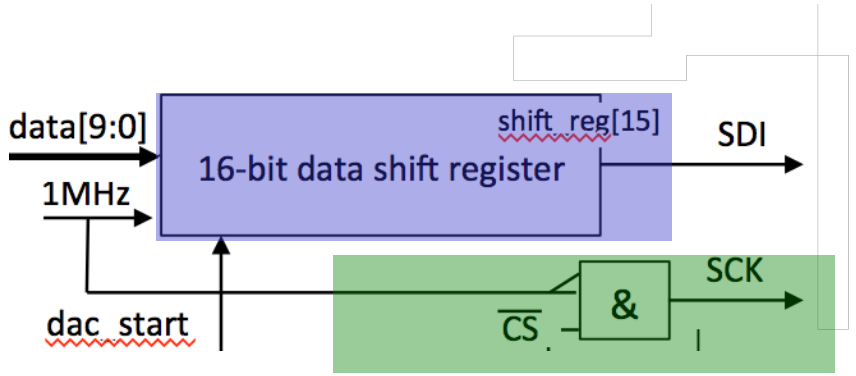


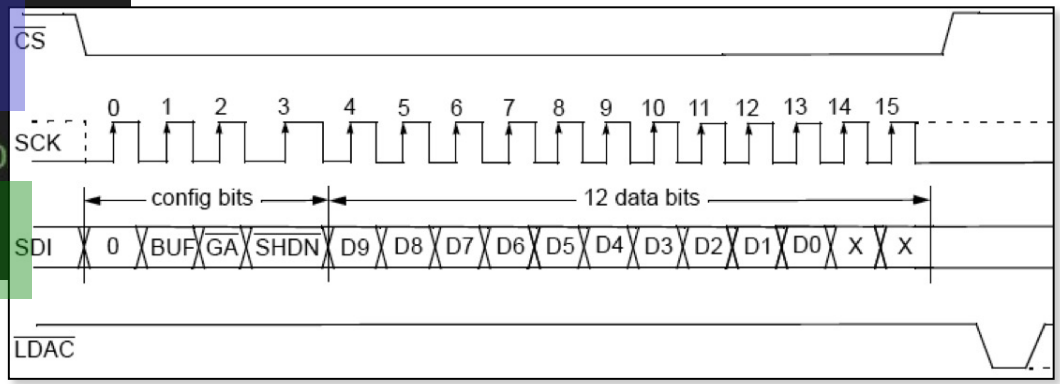


The data shift register

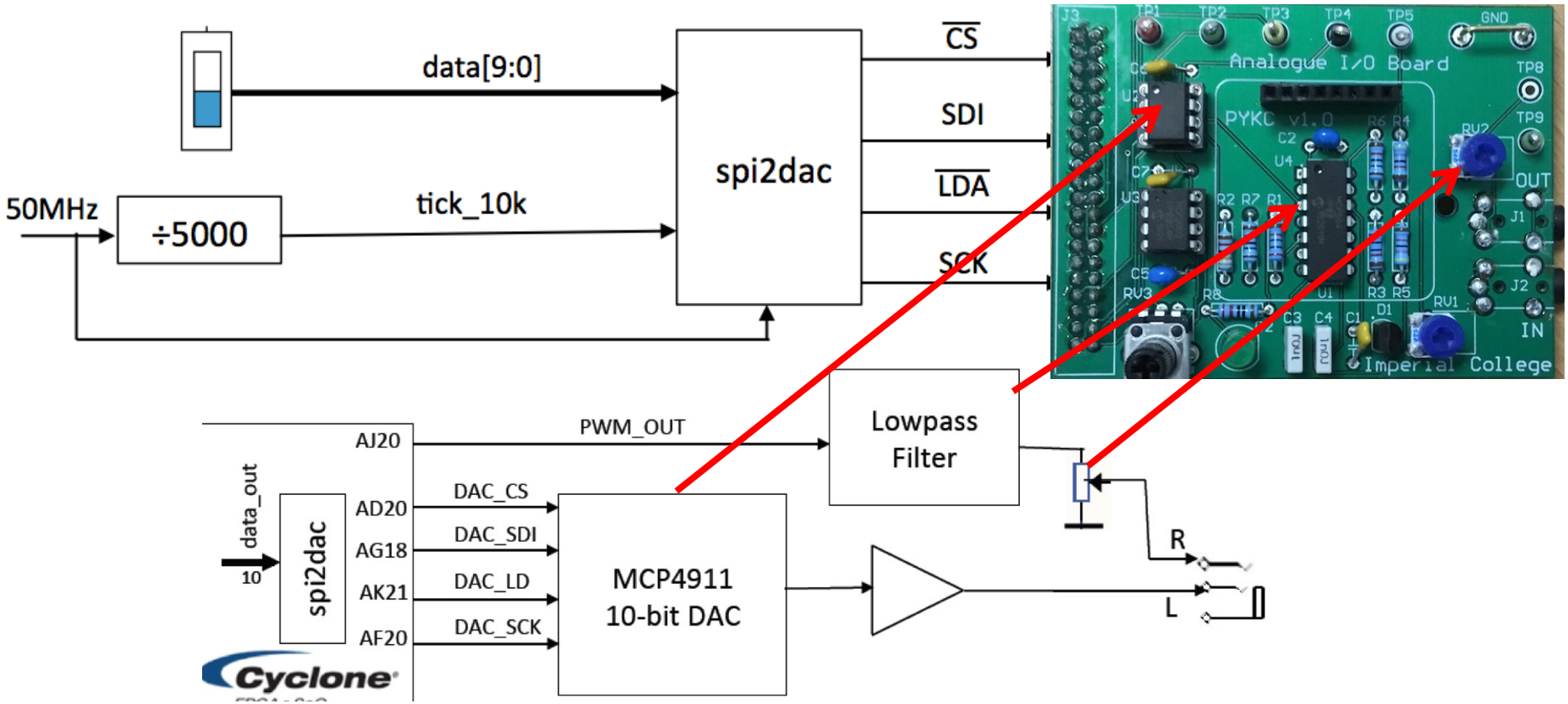
```
parameter    BUF=1'b1;       // 0:no buffer, 1:Vref buffered
parameter    GA_N=1'b1;      // 0:gain = 2x, 1:gain = 1x
parameter    SHDN_N=1'b1;    // 0:power down, 1:dac active

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];
```





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

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

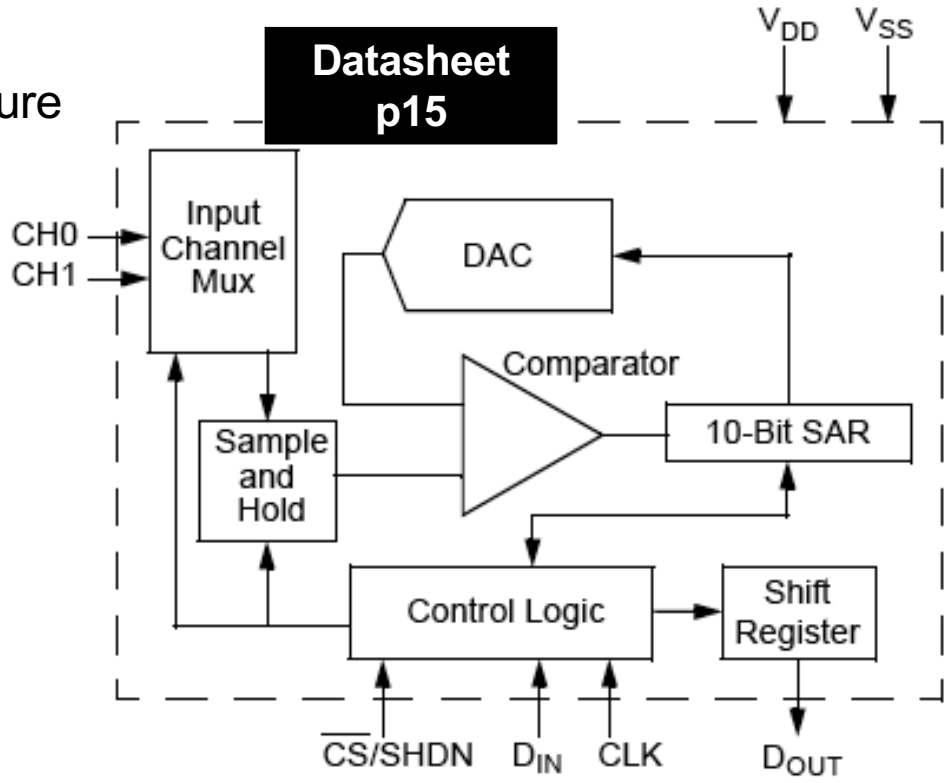
Common mistakes

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

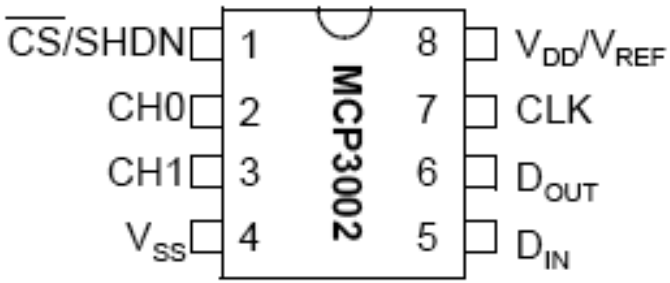
ADC – used in add-on card

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

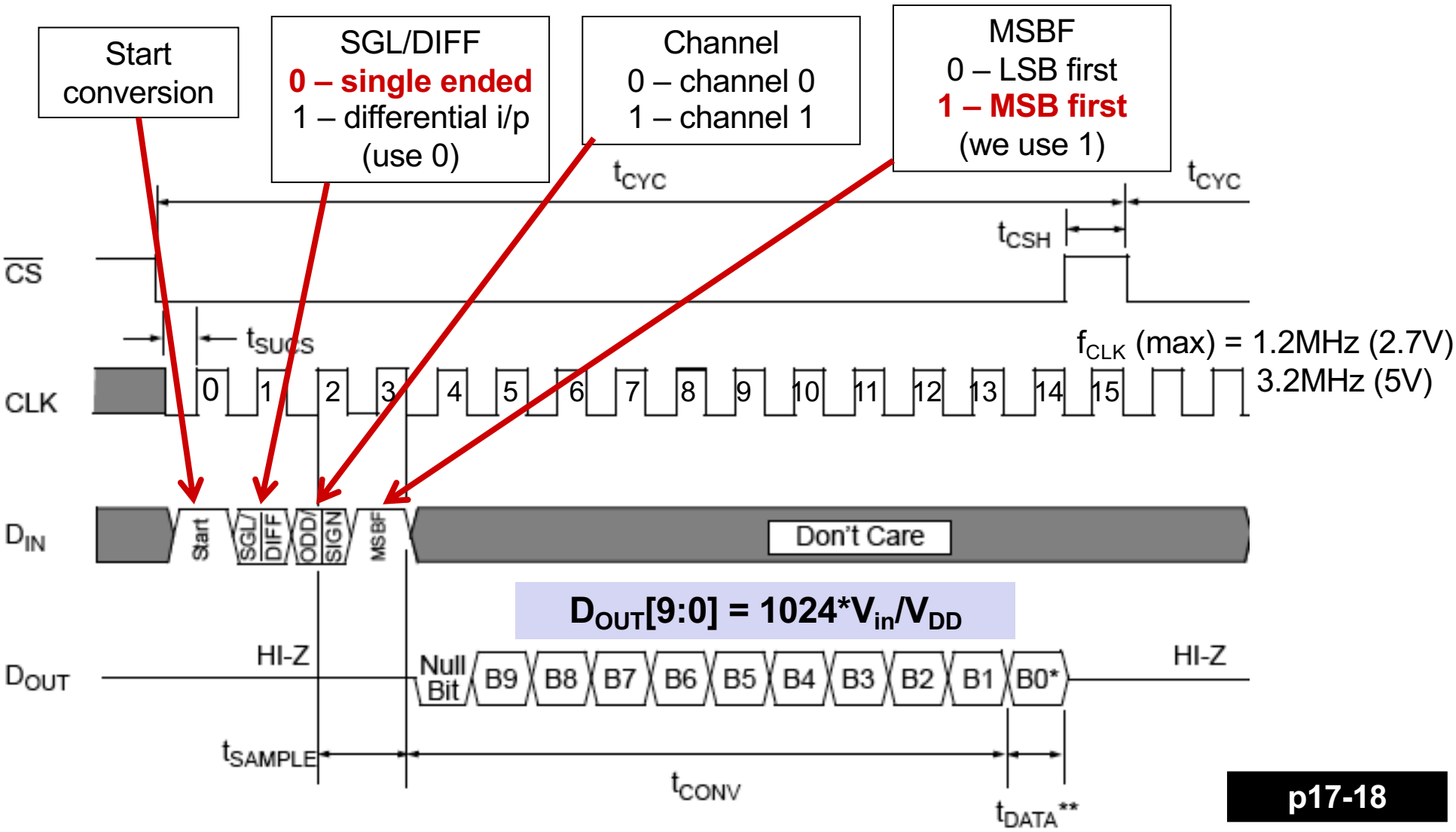
- Analog inputs programmable as single-ended or pseudo-differential pairs
- On-chip sample and hold
- SPI serial interface (modes 0,0 and 1,1)
- Single supply operation: 2.7V - 5.5V
- 200 ksps max sampling rate at V_{DD} = 5V
- 75 ksps max sampling rate at V_{DD} = 2.7V



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



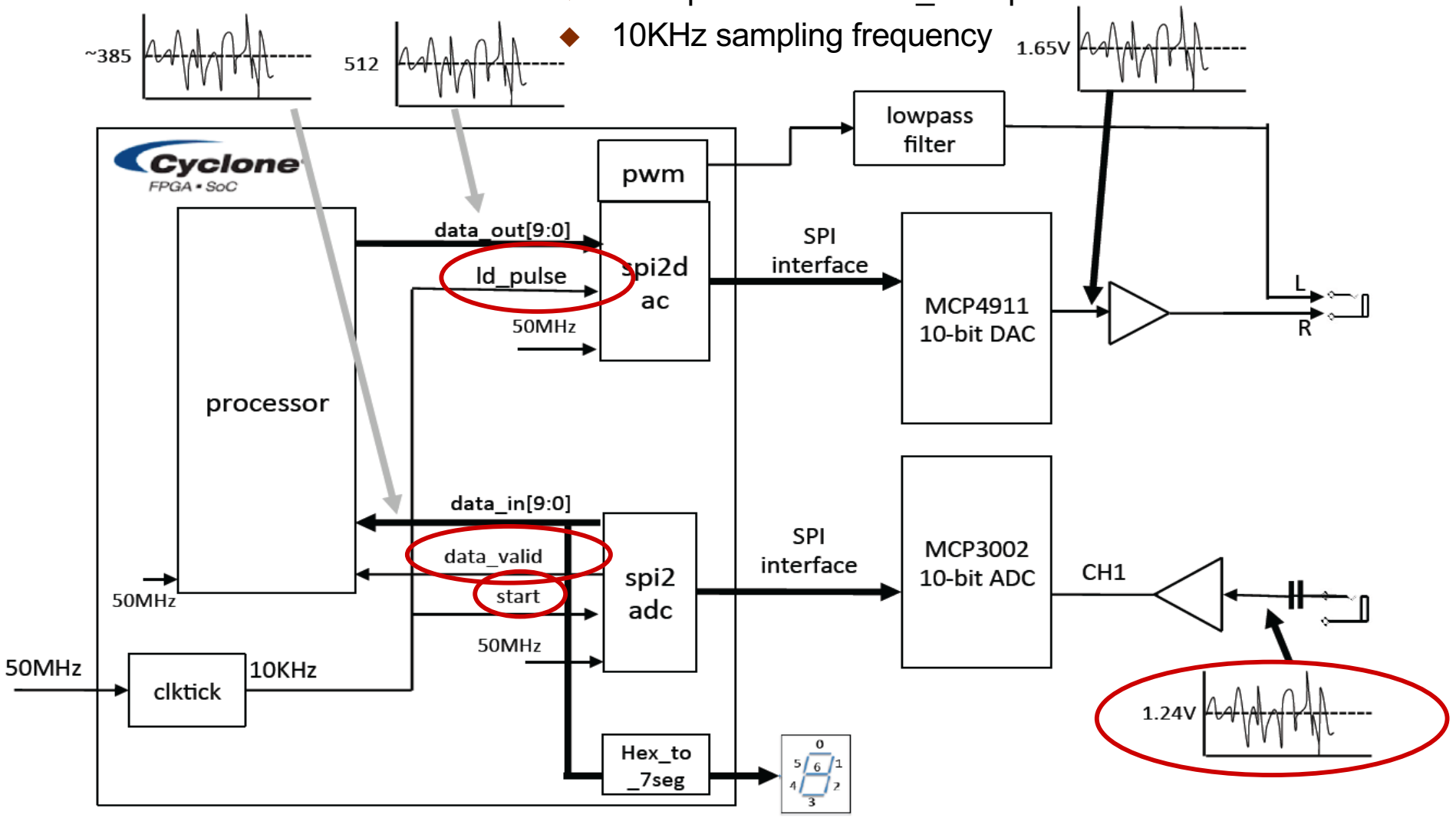
Serial Peripheral Interface for ADC (SPI)



p17-18

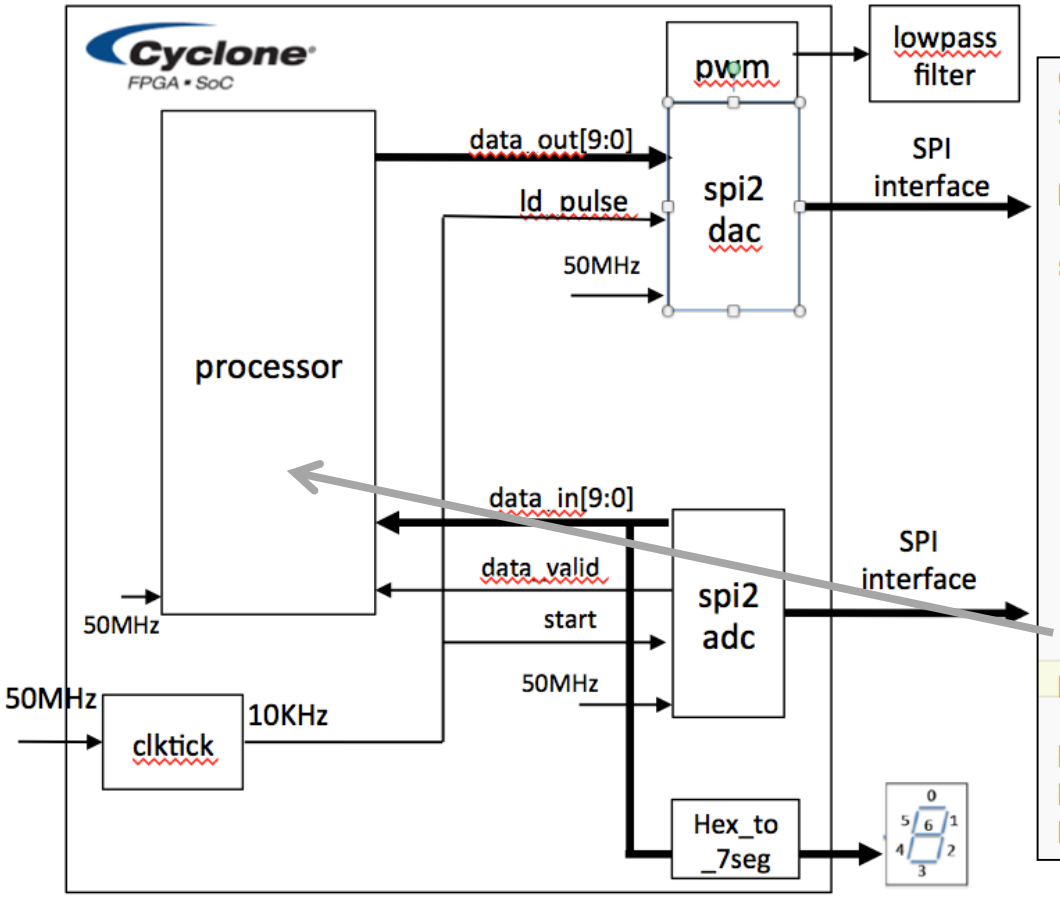
Experiment 16 – All Pass circuit

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



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,      // ser
                 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]});
```

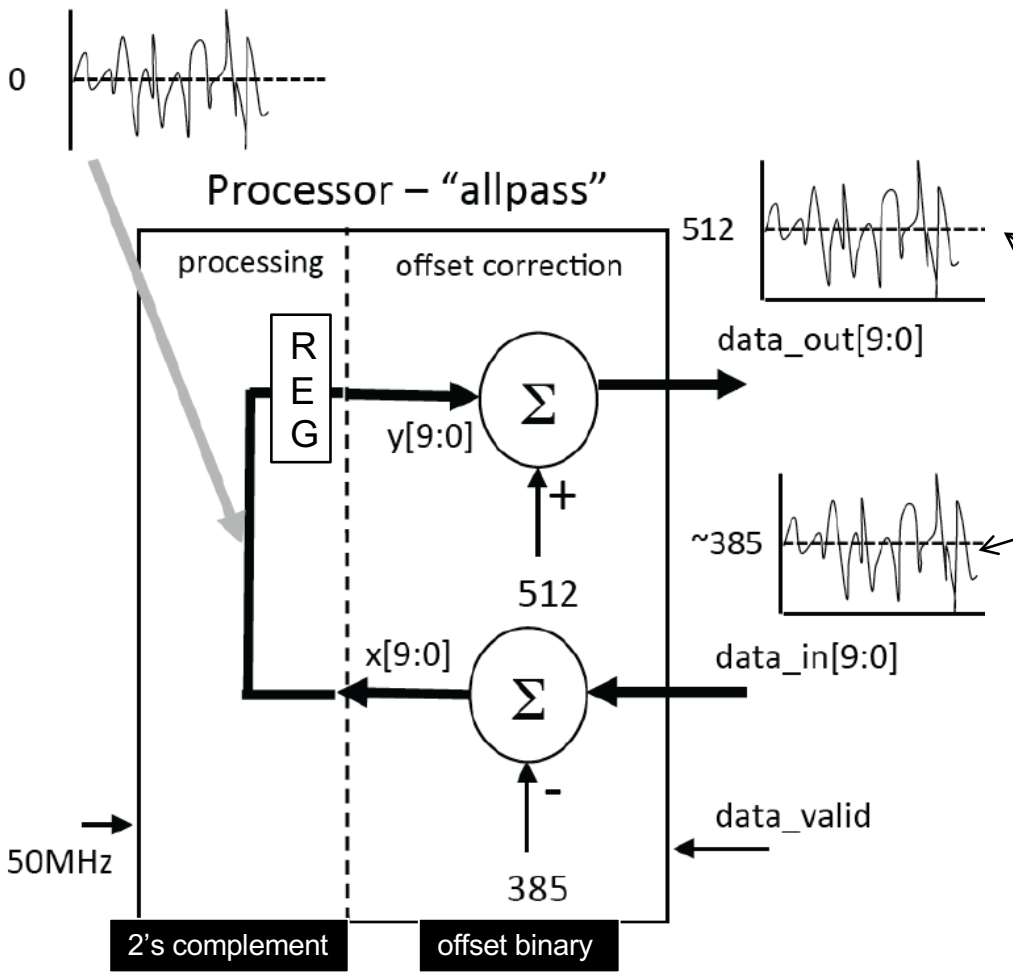
internal_name

external_name

module name

instance name

Experiment 16 – allpass.v (offset correction)

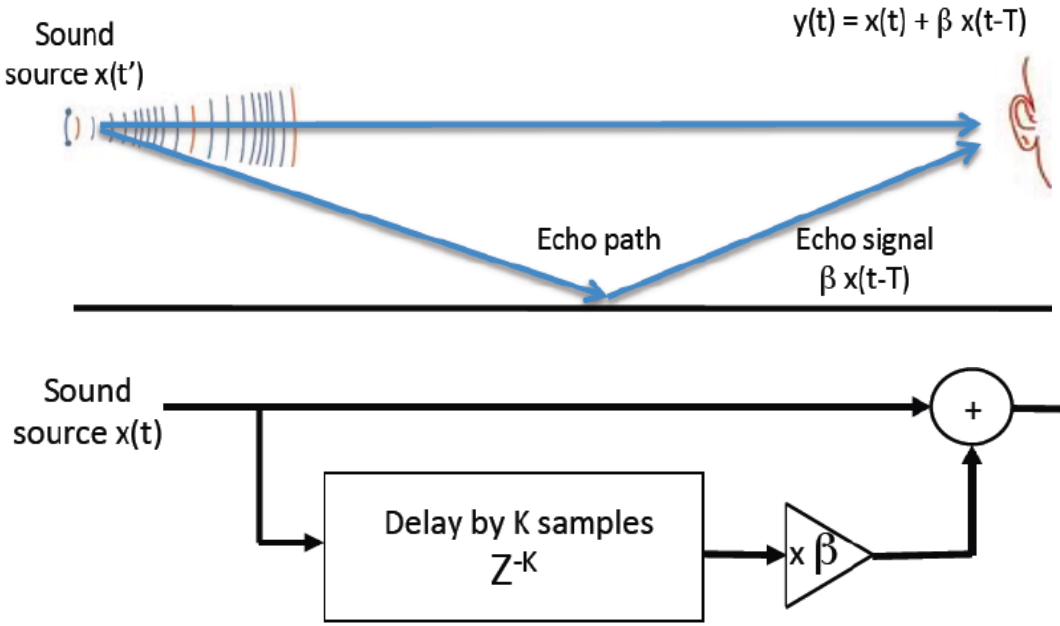


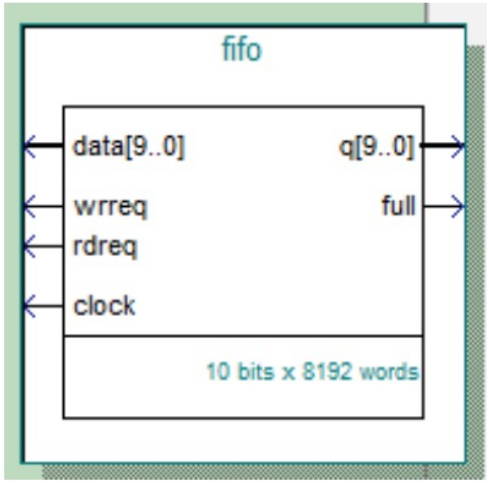

```
module processor (
    input  logic        sysclk,          // system c
    input  logic [9:0]  data_in,         // 10-bit i
    input  logic        data_valid,      // asserted
    output logic [9:0]  data_out         // 10-bit o
);

    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) );

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

endmodule
```

2's complement

offset binary

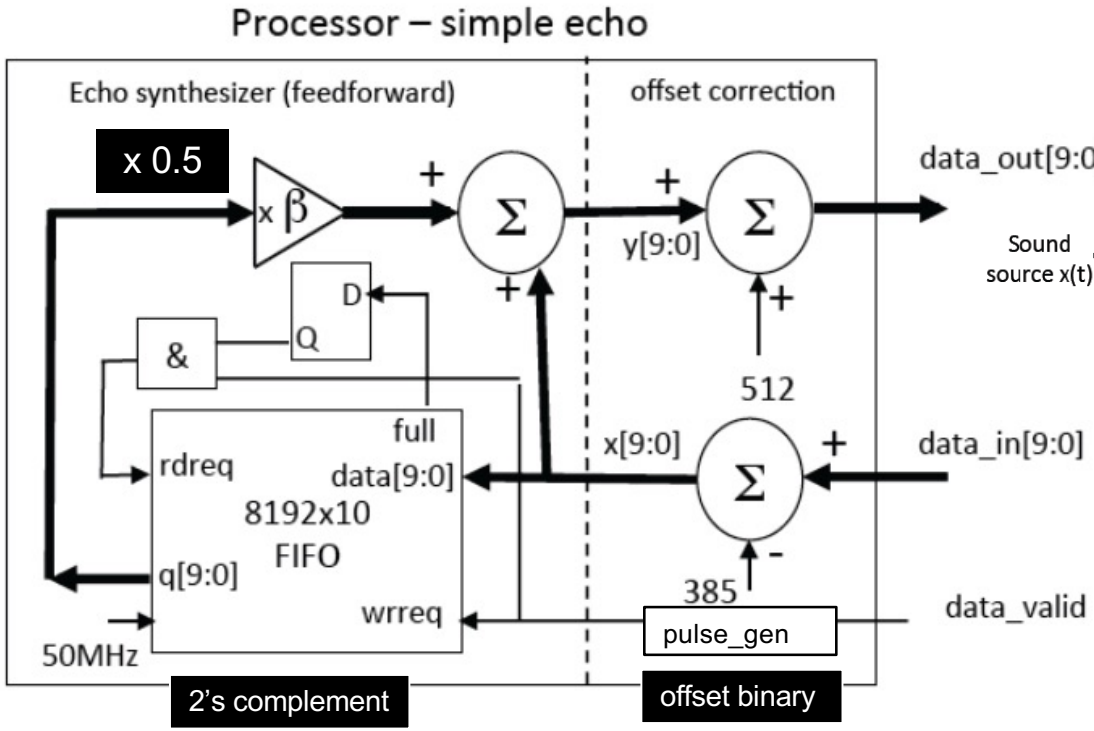
Experiment 17 – single echo synthesizer

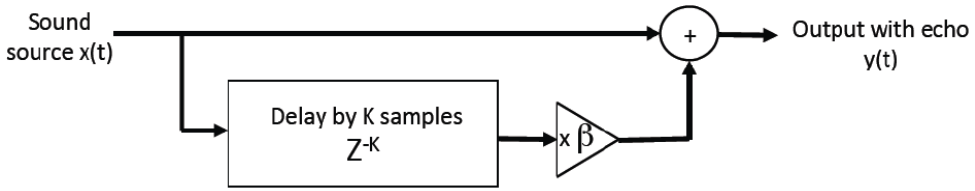


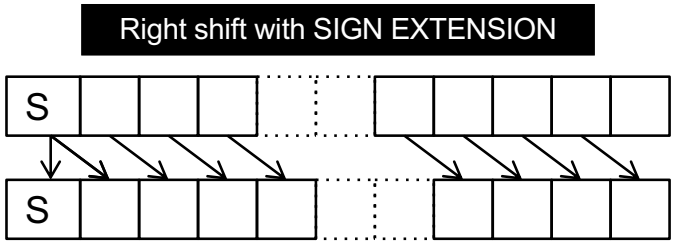


- Single echo of source signal
- Signal flow-graph is simple: a K samples delay block, a gain block and an adder
- Use First-in-First-out memory to store sample: need a status signal “full” to indicate FIFO full
- Sampling frequency = 10KHz, theref a 8192 word FIFO provides 0.8192 second delay

Experiment 17 – single echo synthesizer

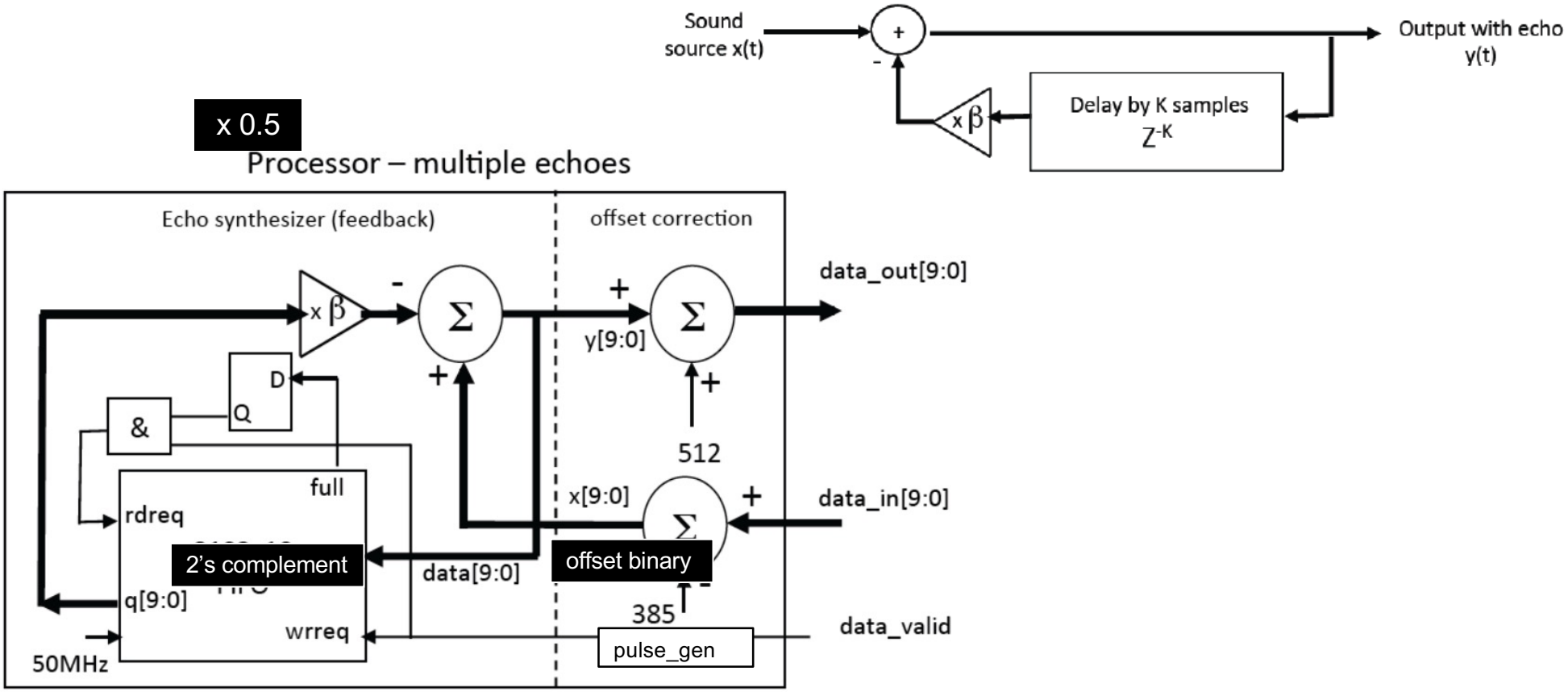






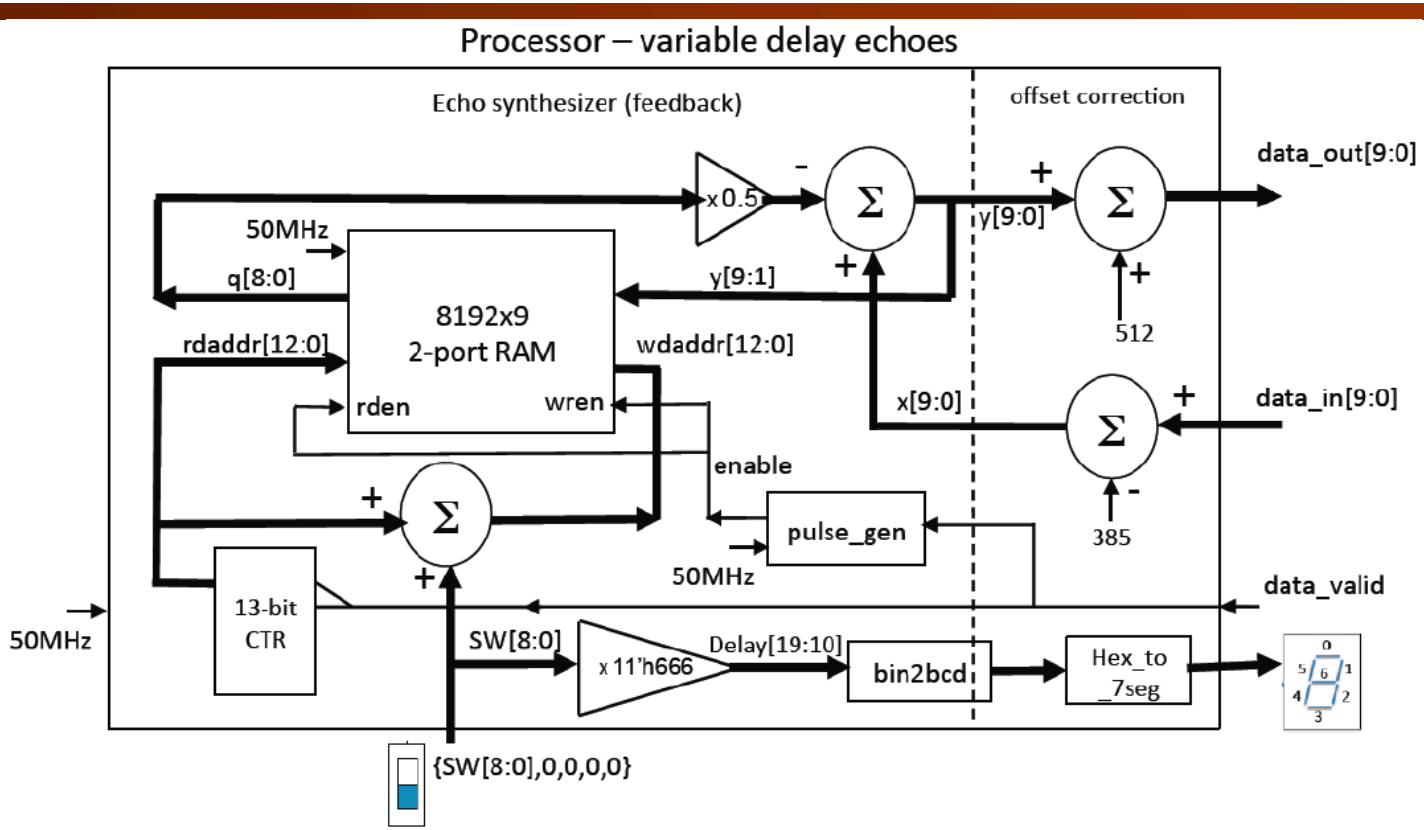
- Computation in 2's complement for signed integers
- x 0.5 = signed right-shift by 1-bit (sign-extension)
- Verilog: y[9:0] = x[9:0] + {q[9], q[9:1]};
- Additional signal to processor module: data_valid = a high pulse whenever there is a new data_in
- Need to fill to First-in-First-out memory before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

Experiment 18 – multiple echoes synthesizer



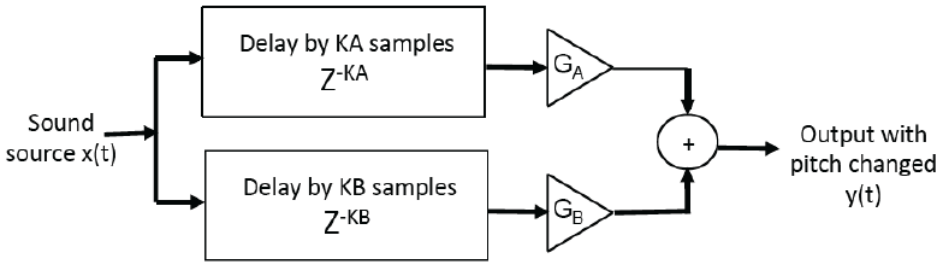
- Instead of feedforward only, this uses a feedback loop
- To avoid instability, you must SUBTRACT delayed echo signal instead of add
- FIFO now stores y[9:0] output, and NOT input

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

Experiment 20 – voice corruptor (grand challenge)



- 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)