Imperial College
London

Lecture 16

The Final Chapter – Part 4 of Experiment VERI

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

- To revisit some of the issues that came up during the 2nd year laboratory experiment VERI
- 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 4 of the experiment.

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

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

Common mistakes

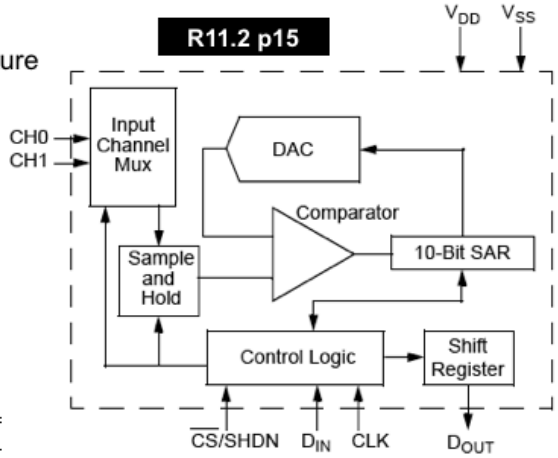
1. Not using h: drive to store design (e.g. Desktop, Library etc.)
2. Bad organisation of design folder – missing versions, files, folder etc.
3. Wrong case for signal names (all names are case sensitive)
4. Wrong number or wrong order of arguments when instantiating a module
5. Different number of bits 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) – only one edge can be specified
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

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

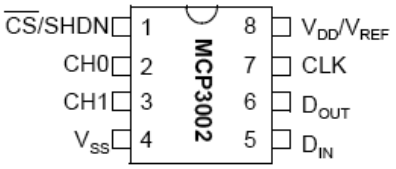
ADC – used in add-on card

- **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

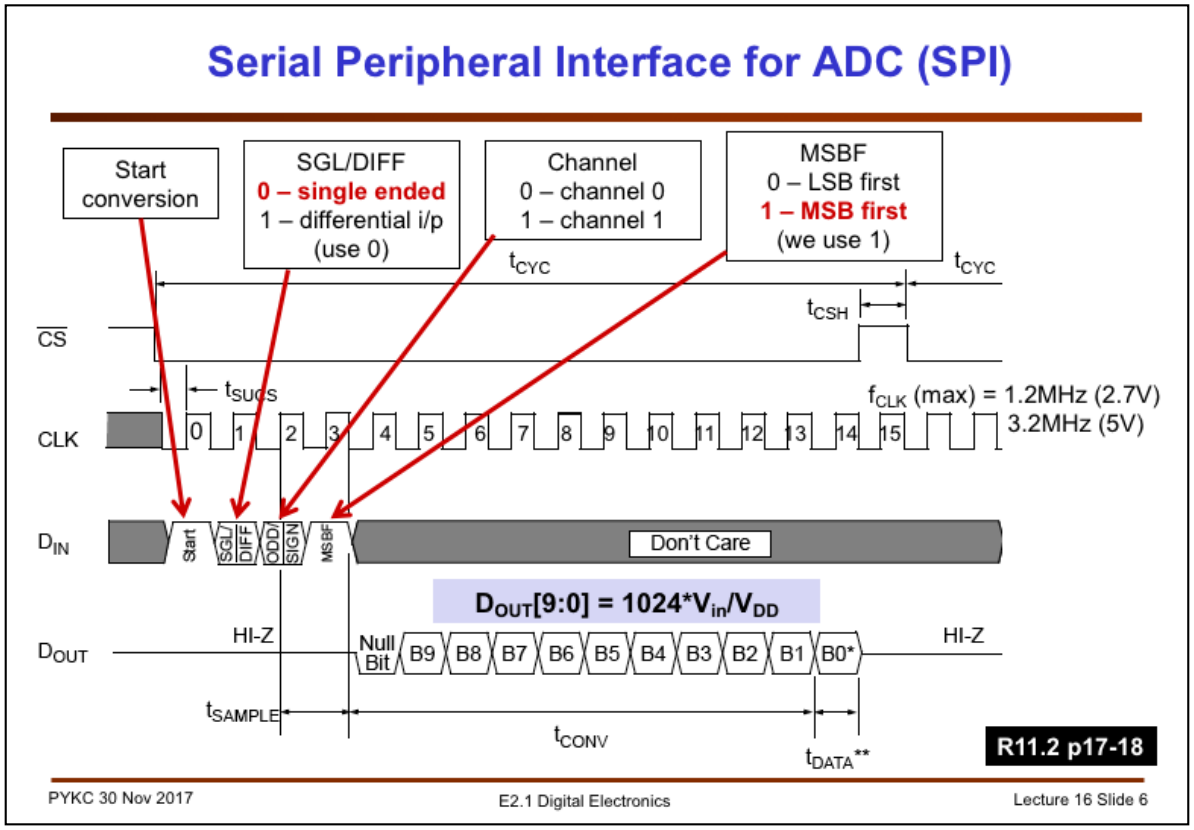


Symbol	Description
$\overline{CS}$/SHDN	Chip Select/Shutdown Input
CH0	Channel 0 Analog Input
CH1	Channel 1 Analog Input
V_{SS}	Ground
D_{IN}	Serial Data In
D_{OUT}	Serial Data Out
CLK	Serial Clock
V_{DD}/V_{REF}	+2.7V to 5.5V Power Supply and Reference Voltage Input



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.

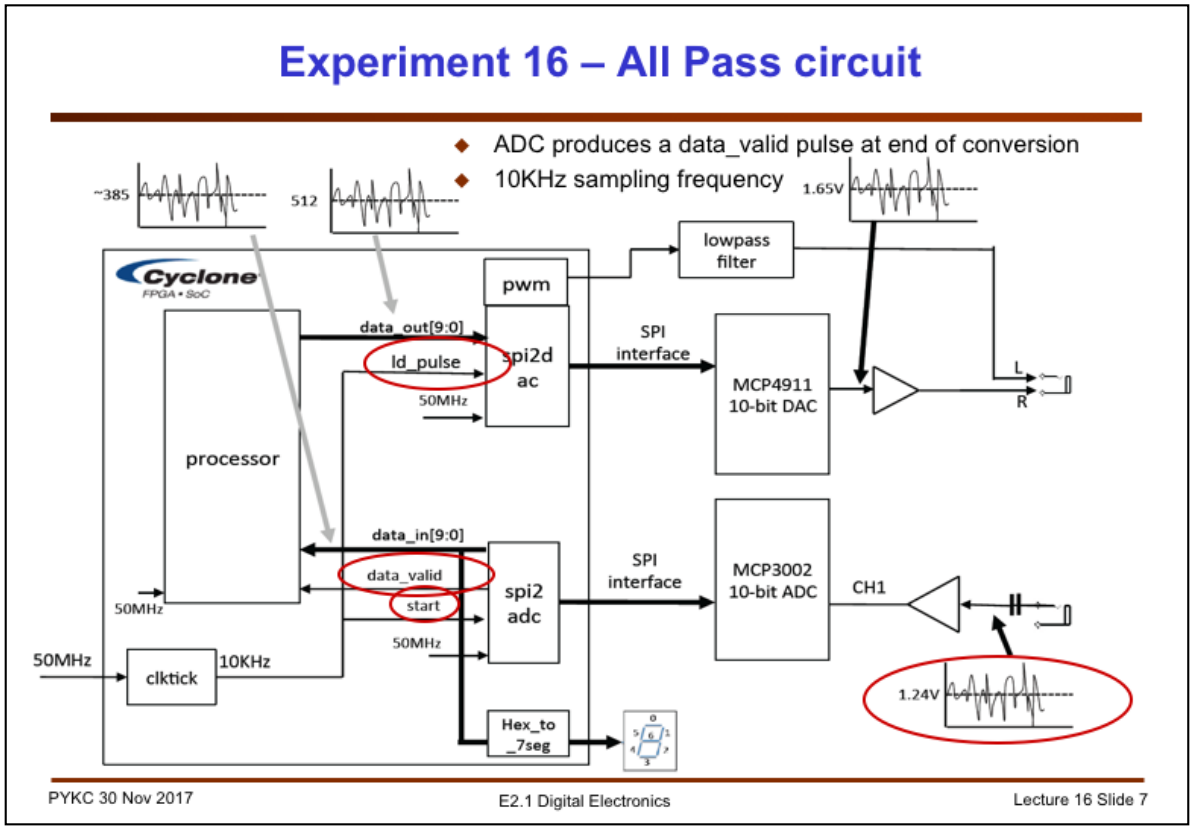


The control of the ADC is slightly more complicated than that for the DAC. Nevertheless, the idea is similar. The transfer cycle is again 16 states, going from state 0 to state 15.

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.

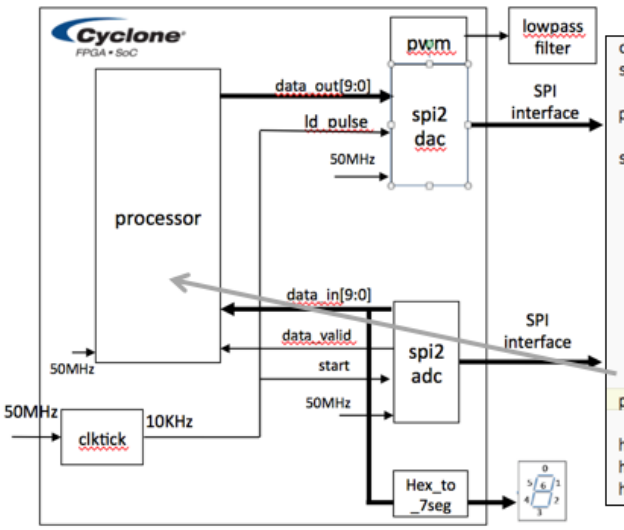
After these four "setup" bits are sent to the ADC, it returns 11 bits to Dout. First bit is always 0. Then the next 10 bits are the converted data MSB first.



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]});
```

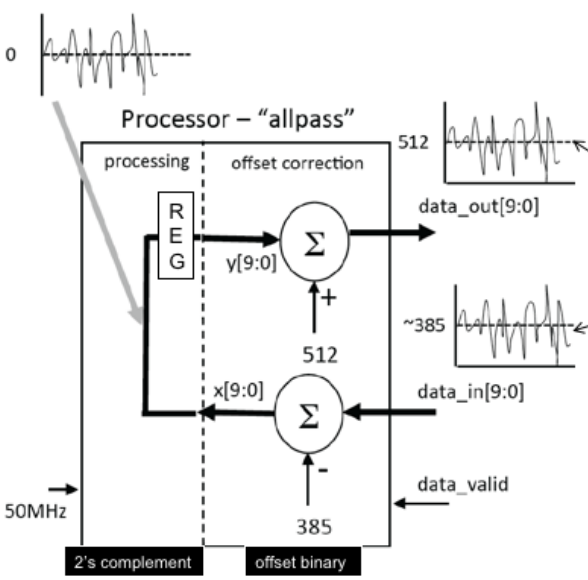
internal_name
external_name
module name
instance name

PYKC 30 Nov 2017 E2.1 Digital Electronics Lecture 16 Slide 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 (sysclk, data_in, data_out);

   input           sysclk;     // system clock
   input [9:0]     data_in;    // 10-bit input
   output [9:0]    data_out;   // 10-bit output

   wire            sysclk;
   wire [9:0]      data_in;
   reg [9:0]       data_out;
   wire [9:0]      x,y;

   parameter       ADC_OFFSET = 10'h181;
   parameter       DAC_OFFSET = 10'h200;

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

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

   //  Now clock y output with system clock
   always @(posedge sysclk)
      data_out <=  y + DAC_OFFSET;

endmodule
```

2's complement

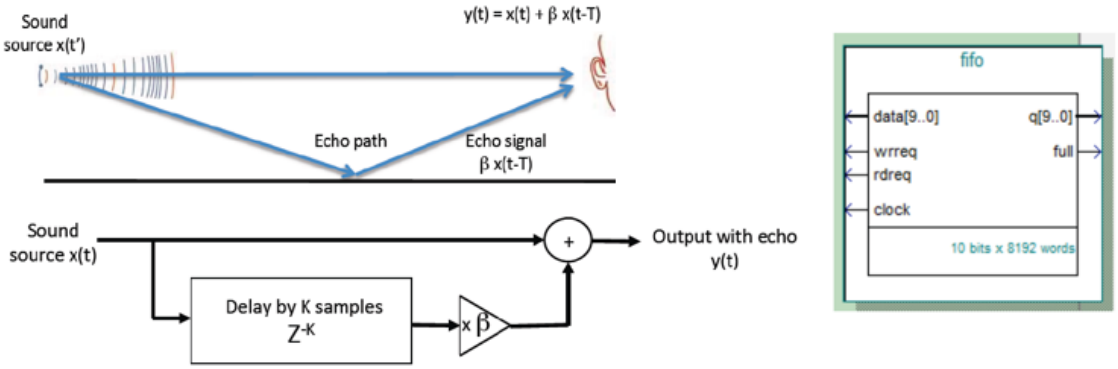
offset binary

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 30 Nov 2017 E2.1 Digital Electronics Lecture 16 Slide 10

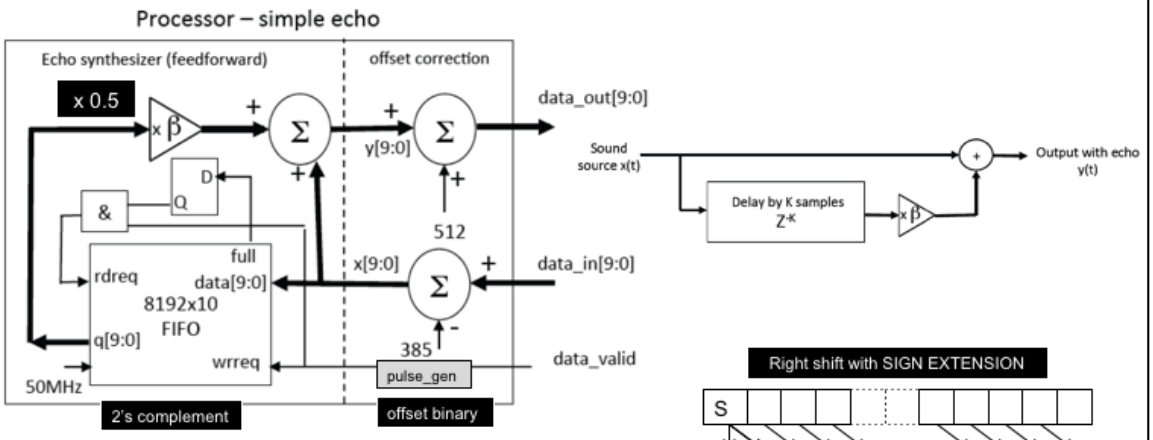
The final compulsory exercise is to create an echo synthesizer. The basic idea is simple: an echo is recreated when the listener receive the source signal via a direct path AND a delayed echo path as shown.

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

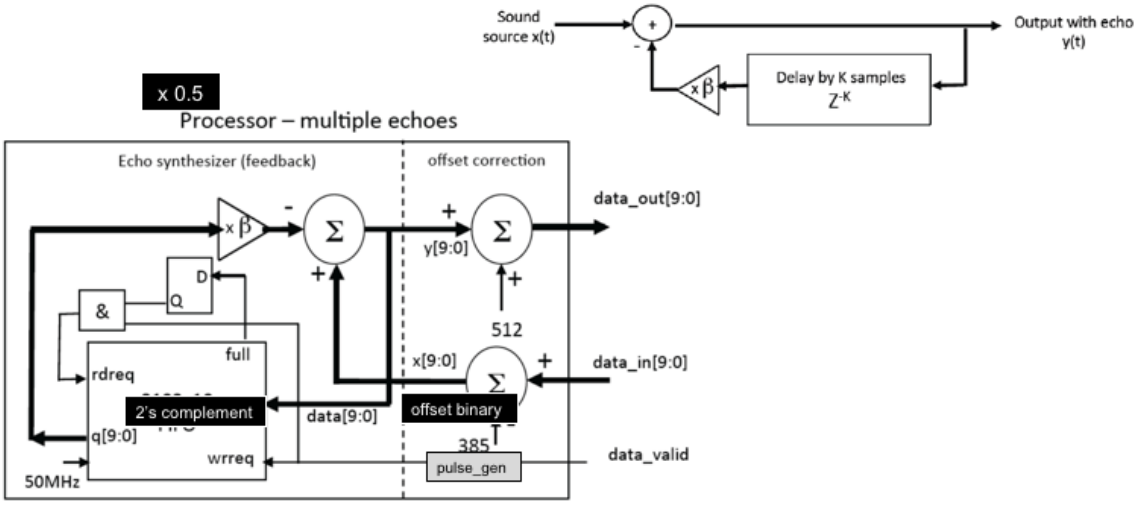


- Computation in 2's complement for signed integers
- x 0.5 = signed right-shift by 1-bit (sign-extension)
- Verilog: y[9:0] = x[9:0] + {q[9], q[9:1]};
- Additional signal to processor module: data_valid = a high pulse whenever there is a new data_in
- Need to fill to First-in-First-out memory before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

PYKC 30 Nov 2017 E2.1 Digital Electronics Lecture 16 Slide 11

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).

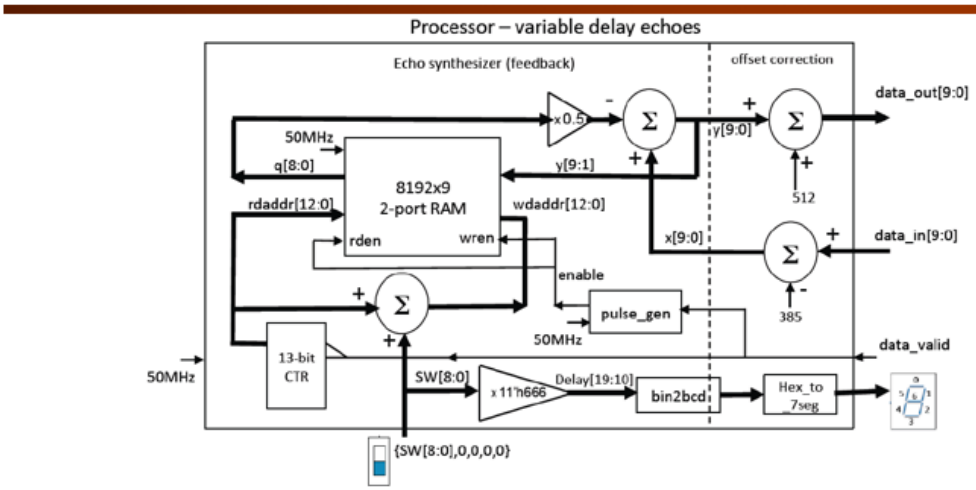
Experiment 18 – multiple echoes synthesizer



- 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.

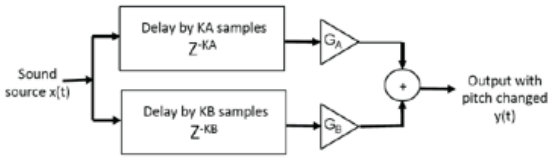
Experiment 19 – variable delay echoes (optional)



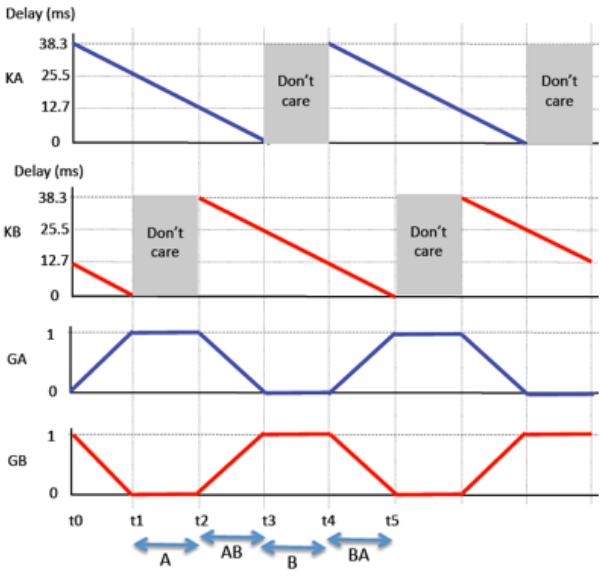
- Entirely optional – do this only if you have time and is truly interested (but at least test my solution)
- Use 2-port RAM instead of FIFO for delay block
- RAM – only 9-bit wide (10-bit not a option), so store most-significant 9 bits y[9:1]
- Write_address = Read_address + delay value from SW[8:0]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



- 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)



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.