EECS 452 Lab 4: DSP on the DE2-70 FPGA board

In this lab we will use the FPGA to do some actual signal processing work. In particular we will be developing FIR filters. In the progress of doing those things, we will be using some new Verilog syntax and learning a bit more about signal processing and FPGAs.

1. Pre-lab

In the pre-lab you will do some filter generation using fdatool and get to see some of the effects of quantization on the transfer function. In addition you will learn a bit of new Verilog syntax.

Notice: You *may* do the two last problems with your lab partner.

Using MATLAB to model quantization effects and generate filter coefficients

Use fdatool to generate an equiripple FIR low-pass filter where, assuming a sample rate of 48,000Hz, the filter should pass anything less than 1500Hz with no more than 3% error (ripple) and reduce the magnitude of any input above 2000Hz by at least a factor of 1000.

Q1. In dB, what should be the maximum gain in the stopband?

Q2. Answer the following questions about your filter that Matlab designed for you.

a. What is the order of the filter Matlab designed?
b. If you were to make it a Butterworth IIR filter, what would be the order?

Have Matlab redesign the filter for the equiripple FIR filter. Now let's look what happens if we were to implement this using an 8-bit representation for the input, output and coefficients. Click the "Set Quantization Parameters" icon from the sidebar. Select "Fixed-point" and change the Numerator word length to be 8 on the Coefficients tab. Select the Input/Output tab and change the Filter precision to "Specify all". After that, set the input and output word length to 8. Select the input and output range to be +/- 1 and then press "apply". Now go to "View→Filter Visualization Tool".

Q3. Zooming in as needed, is the passband requirement still met? The stopband? How close are they?

Change the inputs, outputs and coefficients so all are 16-bit Q15 numbers.

Q4. Zooming in as needed, is the passband requirement now met? The stopband? How close are they?

Now redesign a low-pass filter with Fs=48000Hz, an Fpass of 3000 and an Fstop of 4000. You want about -80dB in the stopband. Export the coefficients to your Matlab workspace (File→Export…). Convert them into the integers (using "round") that you would use for 16-bit Q15 representation. Use Matlab code similar to what you did in lab3 to generate code that looks similar to the following

```
assign coef[0] = 11;
assign coef[1] = 28;
assign coef[2] = 15;
```

Where coef[x] is the xth coefficient of the FIR filter.

Q5. Attach the code you used to generate the coefficients in the format shown above. You do not need to attach the actual coefficients to your pre-lab report but you will need them for the in-lab part.

Q6. What is the largest (in terms of absolute value) integer value you have as a coefficient? The smallest? What is the largest Q value you could have used to represent these coefficients (assuming you continued to use 16-bits)?

More Verilog

Consider the following Verilog code. Assume that the module "bh" takes a 4-bit input and generates the corresponding hex digit.

```
module lab4(
     input  [17:0]SW,
     input  [3:0] KEY,
     output [7:0] HEX0,
     output [7:0] HEX1,
     output [7:0] HEX2,
     output [7:0] HEX3
     );

reg [3:0] a;
reg [3:0] b;
reg [3:0] c;
reg [3:0] d;

always@(negedge KEY[0])
begin
  a<=SW[3:0];
  b<=a;
  c<=b;
  d<=c;
end


bh one   (a,HEX0);
bh two   (b,HEX1);
bh three (c,HEX2);
bh four  (d,HEX3);

endmodule
```

Q7. Consider the above code.

a. Describe how you would expect the code to behave. Be detailed.

b. How would it be different if the "<=" in the always block was changed to a "="?

Arrays of busses in Verilog

Thus far, we've seen syntax like "`reg [3:0] d;`" which declares a "bus", or bundle of wires, named d that has 4 bits labeled d[3] down to d[0]. However Verilog does allow arrays of busses also. For example:

```
reg [3:0] delay [5:0];
```

The above code declares an array of 6 busses each 4 bits wide. The language support for arrays of busses in Verilog is fairly weak—there are a bunch of things you can't easily do. For example, if you want wire "2" of bus "4" of delay there is no clean way to get to that wire[1]. Further, you can't pass arrays of busses between modules. Again, there are tricks to enable passing arrays of busses but they are *really* painful[2].

Using the "for" construct in Verilog

When building filters in Verilog, the "for" construct is very useful. Unfortunately it can also be easy to misunderstand. Unlike in C or C++ the for loop isn't a looping construct per se. Rather it is a shorter way to describe hardware. Consider the following code:

```
module lab4(
      input  [17:0]SW,
      input  [3:0] KEY,
      output [7:0] HEX0,
      output [7:0] HEX1,
      output [7:0] HEX2,
      output [7:0] HEX3
      );

bh one   (delay[1],HEX0);
bh two   (delay[2],HEX1);
bh three (delay[3],HEX2);
bh four  (delay[4],HEX3);

reg [3:0] delay [4:1];
integer i;


always@(negedge KEY[0])
begin
      delay[1]<=SW[3:0];
      for(i=2;i<4;i=i+1)
      begin
         delay[i] <= delay[i-1];
   end
end
endmodule
```

[1] You can do by doing something like "`bob=delay[4]`" and then referencing "`bob[2]`"

[2] The two basic options are to "flatten" the array of busses into a big bus or to use SystemVerilog. SystemVerilog is to Verilog as C++ is to C. It is a much more complex but darn useful expansion of the language.

This does exactly the same thing as the code referenced in Q7.

Q8. Consider these two ways of implementing the same functionality.

a. What are the advantages of the second scheme (with the for loop and array of busses) over the first?
b. When would you really want to use the "for" loop?
c. What might be an issue/worry with using the for loop solution?

PMod converters

Thus far we have been using audio inputs and outputs to do digital conversion. This has some clear downsides as we are stuck with filters that are typically associated with audio (generally things less than 7Hz and greater than 20 kHz get filtered out). Now we are going to learn to use two boards: PmodDA2 and PmodAD1. Documentation is available at

- http://digilentinc.com/Data/Products/PMOD-AD1/Pmod%20AD1_rm.pdf
- http://www.digilentinc.com/Data/Products/PMOD-DA2/PMod%20DA2_rm.pdf

Q9. For the PmodDA2:

a. For each male pin, describe what each pin does. Provide more than just a signal name: describe what role the pin plays.
b. As part "a" of this question, but for the female pins.

Q10. Find the datasheet for the ADC. What is the fastest sclk could be according to the specification? (Hint: you should search for the datasheet for the ADC chip instead of the reference manual for the Pmod provided above.)

Q11. Do these two Pmod devices use SPI?

Q12. Write a Verilog module which implements a 3-tap direct form FIR filter. It should use 16-bit Q15 for its input, coefficients and output. Let the coefficient b[0]=0.25, b[1]=-0.25, and b[2]=0.125. Name the input "x", the output "y" and the clock "clk". Do not use a for loop or an array of busses. This problem will be graded fairly generously given that you aren't expected to test your solution. Just think about how it should be written and you don't have to worry about rounding. _You may do this problem with your lab partner._

Level shifting amplifier with a gain of 0.5

Q13. Design a gain of 0.5 level shifting amplifier. The circuit diagram and analysis for such can be found in on the website. With one exception it should be possible to use only 10 kOhm resistors. The exception can make use of two 10 kOhm resistors perhaps connected in series or maybe in parallel… _You may do this problem with your lab partner._

2. In-lab

In labs 1 and 2 we did filtering and a bit of other signal processing using a processor. In this lab we will use do similar tasks on an FPGA. You should come out of this lab getting a sense for what works well on an FPGA and what its limitations are.

The DE2-70 board has a number of I/O devices including switches, buttons, LEDs, an LCD, and GPIO. It also has an audio CODEC chip. We'll use that chip to input and output waveforms so we can see if our filters are working. This in-lab consists of a number of components: Testing the ADC and DAC, writing a simple 3-tap low-pass filter from scratch, using a larger direct-form FIR filter, converting that filter into transposed form and finally some more advanced DDS work.

Part A: ADC conversion using the Audio CODEC

Let us first confirm that we can convert analog signals into digital signals and back again. In lab 3 you did some simple DDS work which effectively tested the DAC. Now you need to test and characterize the ADC. Do be aware that our ADC is highly configurable and that we are just testing in a specific mode.

Download the Part A code from the Ctools and build it. ***Be certain* the function generator is "on" and in High Z mode, has amplitude of 1V peak-to-peak with 0 offset, *AND* has its output turned off.** Connect the function generator's output to the oscilloscope. Turn the output of the function generator on and double check your peak-to-peak voltage and offset. Be sure you have DC coupling set for the scope.

Now change your connections so that the function generator is not only driving the oscilloscope, but also the **left** channel of the "line in" on the FPGA board. Connect the **right** channel of the line out on the FPGA board to the other oscilloscope input. Program the FPGA board with the code you've built. This code takes the line in value and immediately drives it back out. Keeping the voltage under 4V peak-to-peak see how the device reacts to different inputs.

Q1. After playing around with the inputs, answer the following questions:

a. What is the phase shift of a 1 kHZ input? Give your answer as an (fairly approximate) value between 180 and -180 degrees.
b. How much total delay do you see in the system? How did you figure it out and why are you certain you aren't seeing the phase "wrapping around"?
c. Observe the frequency's impact on amplitude. At what frequency is the output amplitude about half that of the input?
d. Measure the gain and phase at 20 KHz

Q2. With the signal generator output set to 500Hz, *carefully* increment the output voltage by 0.1 V until you see clipping. At what output voltage (absolute) do you start to see clipping?

Examine the top.v module and answer the following questions

Q3. What modules are instantiated in top.v? What modules are instantiated in the whole design?

Q4. Our ADC inverts the input so in order to avoid a 180 degree phase shift between the input and output, we have to negate the input values before feeding to the output. Given that, what else do you think the lines below are doing?

```
always@(negedge AUD_DACLRCK)
            DataOut<= (DataIn == 16'h8000) ? 16'h7fff:-DataIn;
```

Change the code above to the code below and look at a 50Hz sine wave with Vpp = 1 V.

```
always@(negedge AUD_DACLRCK)
            DataOut<={-DataIn[15:13],13'd0};
```

Q5. What happens? Explain what that code is doing and why the output changes as it does.

Part B: PMod converters

Download the Part B code.

Q6. Look over the DAC code

a. Draw a figure that indicates what you would expect the sclk and sync signals to look like. Specifically indicate the frequency you would expect to have of each frequency. .
b. Explain how you figured out the answer to (a).
c. What frequency does the DAC output data at? How does that compare to the audio inputs on the FPGA board?

Remove all connections between the C5515 DSP processor board and the FPGA breakout board (this is just done to limit the damage you can cause if you miss-wire things). ***Turn off the FPGA. Read carefully the DE270Break.pdf for the pin configuration of the extension board.*** Connect the PmodDA2 as you think it should be connected. ***Have your GSI confirm that you have connected it correctly before you go on.*** Once your GSI has done that, turn the FPGA back on.

Now modify the code (the commented part in the code generates ramps on both channels) so that you get a 100Hz ramp out of channel "a" and a 400Hz ramp on channel "b" (each with no more than a 10% error). The Pmods do not accept negative voltages, so be sure to set the 'DC Offset' on the function generator to pass in a correct waveform to the Pmods. Your 'Vpp' and 'DC Offset' should not exceed 3.3 V (The accepted range of input for the Pmods). The definition of the ramp module can be found in support.v. (Hint: it's nothing but a DDS…)

G1. Demonstrate your ramps. Explain how you figured out the changes needed to get those output ramps.

Once you've gotten the DAC to work correctly, the next step is to get the ADC up and running.

Q7. Look over the ADC code and the ADC datasheet.

a. How fast is the sclk being sent to the ADC by the supplied code?
b. How many samples per second does the supplied code convert?

Look over the code and diagrams and figure out where to connect the ADC Pmod. ***Once again power off the FPGA board before you connect anything***. Have your GSI confirm that you've connected the Pmod correctly. Now, revert your changes you made for G1 (it may be easiest to just re-download the code). The code should be set up to send the data received by the ADC to the DAC. Input a reasonable signal (in particular being sure to keep the signal sent to the ADC in the 0-3.3V range by setting the offset to be 1.65 Vdc).

Q8. What is the approximate delay from the ADC to the DAC output?

Q9. Change the input signal to 20 KHz, measure gain and phase and compare to your results in Q1(d).

Part C: A simple filter

As a part of the pre-lab you were asked to code a simple 3-tap filter. We'll use that as a starting point. Close Part B and start a brand-new project. Be sure your qsf file is setup correctly. Then do the following:

- Have the input, output and delay stages each be 16 bits.
- Set the coefficients the same as in your pre-lab question #12.
- Have the input be SW[15:0]
- Drive the filter's output (16 bits) to HEX3, HEX2, HEX1 and HEX0.
- Use negedge of KEY[3] as your clock.
- All the input/output are in Q15 format.

You may **not** use a "for" loop in your code—just use variables.

Answer the following question ***before*** you build and test your filter. (You can go back and change them later of course, but the point is to figure out what behavior you *expect*).

Q10. Say, after programming the chip, you were to set SW[15:0]=0x4000 and leave it there. What output would you expect to get after each press of KEY[3]?

Q11. Say, after programming the chip, you were to set SW[15:0]=0x4000, press and release KEY[3] and then set SW[15:0] to 0. What output would you expect to get after each press of KEY[3]?

G2. Demonstrate your filter to your GSI.

Part D: A more complex filter

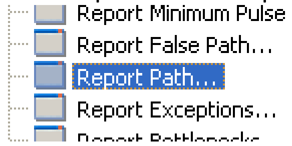
Open the code associated with Part D and build it (expect it will take around 4 minutes). Run the code and play with sine inputs with different frequencies. Make sure your inputs are in the range of 0 to 3.3 V. Below we ask you to figure out what type of filter we have and where the cutoffs are.

Q12. Look at the flow summary under Compilation Report (it should come up automatically when you finish your build).

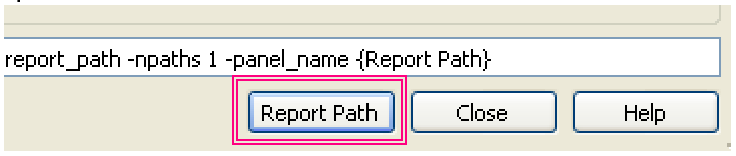
a. Discuss the utilization of resources including the use of embedded multipliers. What is the largest order filter you think we could probably build (assume 90% utilization is as good as we can realistically do).

Q13. What type of filter (lowpass, bandpass, highpass) do we have? What are the cutoffs?

Next we will look at the worst-case combinational logic delay. It's a bit tricky to get to. Click on the "Start TimeQuest Timing Analyzer" icon on the top of the screen. It may take a few seconds to run. Then click on the "TimeQuest Timing Analyzer" icon that is right next to the previous icon. The TimeQuest tool will pop up. Find the "tasks" window (left middle). Scroll down in that window until you find the "Report Path…" option.



In the window that pops up, click on "Report Path" and after a few moments you will see the worst case path. The time units are nanoseconds.


report_path -npaths 1 -panel_name {Report Path}

Q14. What is the worst case delay for this design? Where does it start and where does it end?

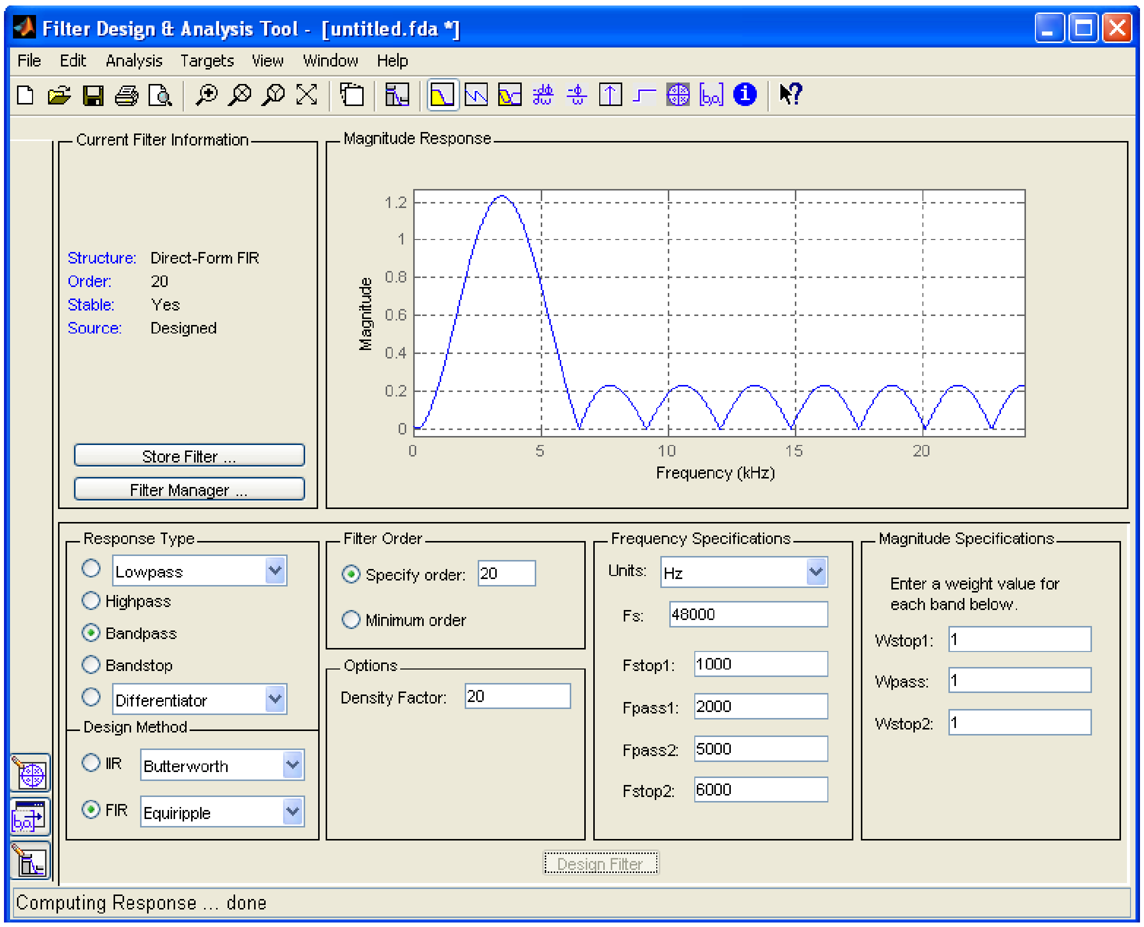
Next, we'll gather a few more data points about resource utilization and delay scaling with filter size. Replace the coef.v file with the coefficients you generated in pre-lab Q5 and make any other necessary changes. Build your project (which may take a while). You can work on the next question in the meantime.

G3. Demonstrate your filter to your GSI.

Q15. Look at the code and explain why "sum" is computed using a blocking assignment while "delay" is computed using a non-blocking assignment.

Q16. For your answer to G3, what did the utilization of resources look like? Did it scale in a reasonable way given the order of the filter? How about the worst-case path?

Build and implement the filter shown below. Notice the maximum gain expected and test to see what actually happens for that frequency.


Filter Design & Analysis Tool - [untitled.fda *]
File Edit Analysis Targets View Window Help
Current Filter Information
Magnitude Response
Magnitude
Frequency (kHz)
Store Filter ...
Filter Manager ...
Design Filter

Structure: Direct-Form FIR
Order: 20
Stable: Yes
Source: Designed

Response Type: Lowpass / Highpass / [✓ Bandpass] / Bandstop / Differentiator

Design Method: IIR Butterworth / [✓ FIR] Equiripple

Filter Order: [✓ Specify order: 20] / Minimum order

Options: Density Factor: 20

Frequency Specifications
Units: Hz
Fs: 48000
Fstop1: 1000
Fpass1: 2000
Fpass2: 5000
Fstop2: 6000

Magnitude Specifications
Enter a weight value for each band below.
Wstop1: 1
Wpass: 1
Wstop2: 1

Computing Response ... done

Now, consider the code below (found in FIR.v). What does it do? Think carefully about what Q value is associated with sum and what Q value is associated with y. Modify the code so that when SW[0] is on, **y** is just equal to **`sum[30:15]`** and when SW[0] is off y is assigned as below and rebuild your filter.

```
        assign y = (sum[31:30]==2'b10) ? 16'h8000:
                   (sum[31:30]==2'b01) ? 16'h7fff:
                                         sum[30:15];
```

G4. Demonstrate your code. Explain to your GSI what is going on and how your code demonstrates the purpose of the code segment above.

Now rewrite the FIR filter in transposed form and use the same coefficients as in G3.

G5. Demonstrate and show your code for your transposed form FIR filter.

Q17. Answer the following questions.

a. What is the worst-case path for your design? How does it compare to the direct form? Does that seem reasonable? Try to explain the results as best you can.
b. What about the utilization of resources? Any change with respect to the direct form? Does this seem reasonable? Try to explain the results as best you can.

The FIR filter is currently running at 48 kHz. Modify the code so that it's running at 100 kHz.

Q18. What is the new cut-off frequency?

G6. Demonstrate your modified filter to your GSI and explain how you would calculate the new cut-off frequency based on the original filter coefficients.

3. Post-lab

Q19. The signal processing we can do with the audio inputs and outputs on the DE2-70 and the C5515 USB stick are fairly limited due to filters built into those devices. Why do you suppose those filters are there?

Q20. Consider the worst-case delay you found for the largest filter you built.

a. What speed (in Hz) could it in theory run at? Assume you need to add 5ns (total) to deal with clock skew, latch delay and setup time.
b. Give a rough idea how that compares to the C5515 doing a similar sized filter.