Term Project for CMPE415
(UPDATE: 11/19/07)

DUE DATE: For part 1, which computes ONLY the DFT of the time domain waveform. (If you stop here, then the maximum grade you can get is 75%).
DUE DATE: For complete project: (Last day of class).

The laboratories serve to provide a vital piece of the project, that related to transferring data from your computer to a memory block within the FPGA. The project builds on the LABVIEW and Verilog code you've written for LAB #4.

Your FSM of LAB #4 has a state that represents a transition between the process of transferring data to the FPGA and the process of transferring it back. The project involves performing an operation on the data before sending it back. In particular, you will implement a waveform filter operation using a discrete fourier transform (DFT).

The low pass filter accepts as input a time domain waveform, e.g., the voltage waveform measured on the power supply as a circuit switches. A waveform is defined by an array of (x,y) pairs, where each x value corresponds to a specific time value and the corresponding y value is a digitized sample of the signal's voltage at that time instance. We will assume that the digitalization process sampled the signal using a constant time interval such that the delta t between each sample is the same for all points in the waveform. The assumption simplifies the low pass filter design, i.e., only the y data values need to be transferred to the FPGA.

The (x,y) pairs representing the waveforms that I give you as examples will be pairs of floating point numbers. Your LABVIEW code from LAB #4 read integers from a file. The file format is changed for the project to include two numbers per line separated by a space or tab, each represented in scientific notation. Therefore, you need to modify this code. Also, a filter value will follow the (x,y) pairs in the file, i.e., it will be the last value in the file on a line by itself. Its data type is integer.

Once you have the data value pairs read in using LABVIEW (as floating point numbers), you'll need to scale the y data to convert the floating point numbers to **8-bit** integers. One approach is to multiply each of the y values by a constant. This simple approach may not make full use of the entire range that 16-bit integers provide, from -127 to 127. The proper way to scale the floating point values is to *parse* the array identifying the largest and smallest values. This gives the range of the original data. The ratio of the integer range to floating point range gives a scaling factor that you can use to multiply each of the floating point values. To make this work, you'll also need to compute an offset that needs to be added to the scaled y value in order to center the waveform in the range of integers. When completed, the smallest and largest floating point y values will map to the smallest and largest integer values given above. See formula below.

At this point you have the (x,y) values of the waveform stored in floating point format and the y values translated to 8-bit integers. Once the FPGA processes the data (to be described), you will need to know the x delta (time interval between each consecutive value) and the range (you can assume the first x value is 0 to represent time 0 if you like). You can do this before you transfer the

Parse the floating point y values in LABVIEW, save largest and smallest values and compute two constants:

zero = (largest + smallest)/2;

mult = (largest - smallest)/(2^8- 2);

To convert the array of floating point values to integers:

y_int[i] = (y_float[i] - zero)/mult;

y data to the FPGA or you can do it afterwards (you can also do it in parallel with the FPGA for bonus points).

The code that you have written for LAB #4 is designed to transfer 32-bit values. You can modify it to transfer the 8-bit signed values or you can leave it as is (the high order 24 bits will be all zeros (positive number) or all ones (negative number)). You need to regenerate you memory core to either allocate space for **1024** 32-bit values or you can generate three separate cores, one with **512** 32-bit values for the time domain data and 2 for the frequency domain data (real and imag -- see below), each with **256** 32-bit values.

The data transferred to the FPGA is the time domain data, i.e, the 8-bit integer y values that you created in LABVIEW. Following these **512** 8-bit y values, you need to send one additional value, the number of high frequency components to eliminate in the filter operation (more on this soon). You should save this value in a special register instead of storing it in memory.

The low pass filter you will implement transforms the y data from the time domain into the frequency domain using an n-squared DFT. The C code for the DFT is given below:

```
for ( j = 0; j < num_pts/2; j++ )       // Num of frequencies 1/2 num y values
      for ( i = 0; i < num_pts; i++ )
          {
           real[j] += y_data[i] * cos(j*2*PI*i/num_pts);
          imag[j] += y_data[i] * sin(j*2*PI*i/num_pts);
          }
```

As you can see, it's fairly easy to write in C. (NOTE: The full transform actually needs num_pts/2 + 1 points in the frequency domain, so we are not computing the highest frequency component using this formulation. Also, the imag components should be negated but we fix with this later in the inverse transform).

The y_data values are already scaled to be integers once they are transferred to the FPGA, so these are fine (y_int). However, the sin and cos functions output values between -1.0 and 1.0 using the C library routines. On the FPGA, there is no C library so we need to implement these functions in hardware. Moreover, we don't want to deal with floating point numbers, so we need special functions that return integers instead of floating point values.

We will use the Core Generator to generate the sin and cos functions on the FPGA. Fortunately, these functions are built so that they take integers as operands and return integers values. Our task

here is to understand how to translate the arguments shown in the C code to integers and how to interpret the integer returned by these functions.

The Core Generator defines the input argument, THETA, to the sin and cos functions as follows: θ

$$\Theta = \text{THETA} \times \frac{2 \times \Pi}{2^{\text{THETA\_WIDTH}}} \qquad \text{in radians}$$

represents the argument to the sin and cos functions in the C code. THETA is the input that you will provide when you invoke the sin and cos functions. The constants 2 * pi/$2^{\text{THETA\_WIDTH}}$ represent “2*PI/num_pts” in the C code above. This indicates that the core generated sin and cos functions already incorporate these constants so you don’t need to worry about them. When it is all said and done, you will only be responsible for multiplying “j*i” and this will define the THETA that you will use as input to the sin/cos core.

The output of the sin and cos functions will be a 14-bit **signed** integer. This integer represents a fixed point number when divided by the appropriate constant. The constant that you will divide by is 2^{12}. You will perform this division once you have transferred the numbers back to LABVIEW (don’t do the division on the FPGA). More on this later.

The 14-bit values output from the sin and cos functions need to be multiplied by the y data values (which are also 8-bits), as shown in the C code above. The result of the multiplication is a 22-bit value (14+8). The ‘+=’ operation in the loops above will need no more than an additional 9 bits because the number of times the ‘+=’ is performed is 512 (2^9). Just to be safe, you will have 32-bits - 22-bits = 10-bits. This will guarantee that no overflow will occur. Bear in mind that this is the worst case -- you will probably be able to use y values with more precision (instead of dividing by 2^8-2 above, you may be able to use 2^{12}-2 instead -- I know this works for the triangle wfm that I’ve given you).

You need to define a set of parameters when running the Core Generator, just as you did for the memory core. The parameters that I used are as follows:
Output width: 14
THETA input: 9
Function: sine and cosine
Memory Type: Distributed ROM

Input Options: Non-registered
Output: Non-registered
Output Symmetry: Symmetric

One way to perform the filter operation is to clear a set of values in the imag and real arrays that you’ve just computed on the FPGA. For example, if the filter value is 10, then the last (high-order) 10 values of these two arrays should be cleared, i.e., set to zero. The filter is complete by transforming the data back to the time domain using a sequence of operations that are similar to those shown above for the forward transform (see figure below for reverse transform)

```
real[0] /= num_pts;
for ( j = 1; j < num_pts/2; j++ )
    {
    imag[j] *= 2/num_pts;
    real[j] *= 2/num_pts;
    }
for ( i = 0; i < num_pts; i++ )
   y_int[i] = 0;

for ( i = 0; i < num_pts; i++ )
   for ( j = 0; j < num_pts/2; j++ )
       y_int[i] += real[j]*cos(j*2*PI*i/num_pts) + imag[j]*sin(j*2*PI*i/num_pts);
```

One stopping point of the project (with a maximum grade of 75%) is to compute the frequency domain representation and transfer it back to LABVIEW, i.e., don't bother with performing the filter operation at all. It is HIGHLY recommended that you do this even if you plan to do the filter portion so you can determine if your Verilog code is working properly. To verify, you can compute the real/imag frequency domain values using LABVIEW and compare it with the result computed by the FPGA.

Once you have transferred the frequency domain data computed by the FPGA back to LABVIEW, you first divide each of the real and imaginary components by 2^{12}. As mentioned above, the sin/cos core generates 14-bit signed integers that represent the range 1.0 to -1.0, as shown in the formula below This division scales the sin and cos values returned by the sin/cos core to this range.

FPGA_cos_core = $\cos(i*j*2*PI/2^{THETA})*2^{14}$ — Core generates this value given (i*j) as an argument.

$$\text{real}[j] = \sum_{i=0}^{n-1} \text{y\_int}[i] \times \cos\left(\frac{i \times j \times 2 \times \pi}{n}\right) \times 4096$$

This calc for each real -- 4096 is easily removed by division. (Similar for imag components).

The second conversion involves y_int. We used 'y_int[i] = (y_float[i] - zero)/mult' to convert from float to int. Plugging in;

$$\text{real}[j] = \sum_{i=0}^{n-1} \left(\frac{\text{y\_float}[i] - \text{zero}}{\text{mult}}\right) \times \cos\left(\frac{i \times j \times 2 \times \pi}{n}\right)$$

$$\text{real}[j] = \sum_{i=0}^{n-1} \left(\frac{\text{y\_float}[i]}{\text{mult}}\right) \times \cos\left(\frac{i \times j \times 2 \times \pi}{n}\right) - \underbrace{\sum_{i=0}^{n-1} \left(\frac{\text{zero}}{\text{mult}}\right) \times \cos\left(\frac{i \times j \times 2 \times \Pi}{n}\right)}$$

where n is the number of data points, e.g., 512.

Except for real[0], this sum is 0.

Therefore, to convert back to float, you should NOT add the zero when applying the inverse formula (except for real[0] explained below). Instead, just multiply each real[j] by 'mult' as shown below.

```
real[i] = real[i]*mult;
imag[i] = imag[i]*mult;
```

For real[0] (the DC value of the DFT), you ALSO need to add the following constant:

```
real[0] = real[0] + zero*num_pts;
```

Finally, to convert from real and imaginary values to magnitude and phase (the human readable representation of the frequency domain data), use the following:

```
mag[0] = real[0]/num_pts;
phase[0] = 0;
for ( i = 1; i < num_pts/2; i++ )
   {
   imag[i] = imag[i]*2.0/num_pts;
   real[i] = real[i]*2.0/num_pts;
   mag[i] = sqrt(imag[i]*imag[i] + real[i]*real[i]);
   phase[i] = atan2(-imag[i], real[i])*180/PI;
   }
```

To create the x axis for the frequency domain mag and phase values, first define a fundamental frequency as 1/(time range of time domain data). Each frequency component is labeled using:

```
for ( i = 0; i < num_pts/2; i++ )
   freq[i] = i*fund_freq;
```

You can use the LABVIEW express VI to plot the mag and phase spectra as waveforms.

SPECIAL NOTE: Be sure to run GXSTEST each time to power up your FPGA. You should see the 7 segment display cycle through some digits at the end of the test and a 'FPGA passed' message displayed. You then use GXSLOAD to transfer the bitfile generated by ISE to the FPGA. Once you have done this, check to make sure the 7 segment display has ONLY the 'dot' illuminated, none of the 7 segments should be illuminated. If they are, then the transfer process didn't work properly (I've had this happen occasionally). DO NOT start testing your code until this condition holds, i.e., only the 'dot' is illuminated -- power cycle and reload if necesssary.

The grading criteria for the project given in the document "Details of laboratory grading criteria" on my web page. There are lots of ways to earn extra credit. Please discuss these with me if and when you are ready to move beyond the basic tasks associated with the project.

UPDATES WILL BE POSTED AS SOON AS THEY BECOME AVAILABLE (and highlighted in this document). Please check this description periodically for changes!!!