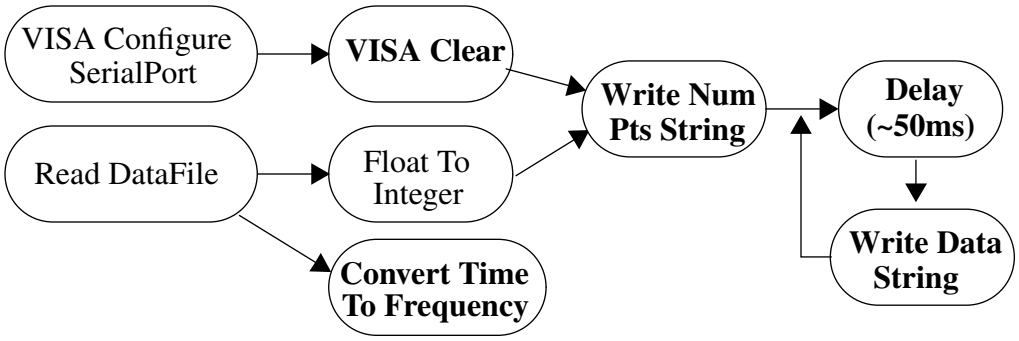
LAB Assignment #3 for ECE Co-Design (495/595)

Assigned: Wed., Feb 17, 2010
Due: Wed., Feb 24, 2010: Part I
Due: Wed., Mar 3, 2010: Part II

Description: Transfer the y integer values from the waveform to the PPC, compute a 'software' DFT and send the values back to labview for conversion and viewing.

PART I:
a) Modify the LABVIEW from Lab #1 and #2 to transfer the y integer values to the FPGA -- see flow chart below:



The **bold** steps in the flow chart are new steps (the others are from previous labs). The **VISA Clear** step is a labview VI. It clears the COM port of data from previous attempts. The **Write Num Pts String** uses the VISA Write widget. Convert the number of points, 512, into a string and write it to the COM port. The next step, **Delay (~50ms)**, is important. You are writing a large number of bytes to the COM port. If you don't use flow control, e.g., check if the COM port has room for more bytes in its buffer, you'll overwrite the buffer and loose data if you let labview just write the entire file of numbers. One way around flow control is just to slow down the write operation. Once you've written the number of points, you convert each integer value of the wfm into a character string and write it to the COM port (**Write Data String**). Add a delay between each write. 50 ms is conservative. You can start with this value and then lower it. I eventually up'ed the BAUD rate to 19,200 and set the delay to 1ms and it works fine. For the **Convert Time To Frequency** step, you are creating the x axis for the frequency domain *magnitude* and *phase* values. First define a fundamental frequency as 1/(time range of time domain data). Then each frequency component is computed using:

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

2) Create a project in EDK and write the C code that implements a Discrete Fourier Transform.

In EDK, create a project as described by the HelloWorld tutorial. You'll need to make some modifications once you've created the project. Before you run the **Generate Linker Script**, click on the 'Addresses" tab under 'System Assembly View. Change the 'xps_bram_if_crtl_1' size to 128K. When you run **Generate Linker Script** step (see EDK tutorial 1), **set the stack size 0x4000 AND the heap size to 0x2000** (defaults are 0x400, which are too small for scanf and printf to work). Create a C program that makes use of the code that I've provided (for the integer lookup table version of *sin* and *cos*). Your code should read in the header (512) and the y integer values using scanf, compute the DFT saving the values in *real* and *imag* arrays, and then output the 256 *real* and 256 *imag* values to the COM port using printf. 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] * GetCos(i*j);
          imag[j] += y_data[i] * GetSin(i*j);
          }
```

GetCos and *GetSin* are calls to the lookup table functions that I've provided on my website.

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 when converting to *magnitude* and *phase*.
NOTE: This DFT implemented on a processor with floating point operations is given by:

```
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);
          }
```

The C library *sin* and *cos* functions output floating point values between -1.0 and 1.0. On the PowerPC, floating point operations are not supported. So we need to write special *sin* and *cos* functions that takes integer arguments and returns integers instead of floating point values. The portion of the argument '2*PI/num_pts' is removed and incorporated in the special *GetSin* and *GetCos* integer lookup functions that I've provided. This implementation emulates what we'll eventually do in hardware, as described below.

In our future lab which implements the DFT in the reconfigurable logic, we'll use the Core Generator to generate a hardware based *sin* and *cos* lookup table. The Core Generator for the *sin* and *cos* function defines the input argument, THETA, to the *sin* and *cos* functions as follows: θ represents

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

the argument to the *sin* and *cos* functions in the C code. THETA is the input that you will provide

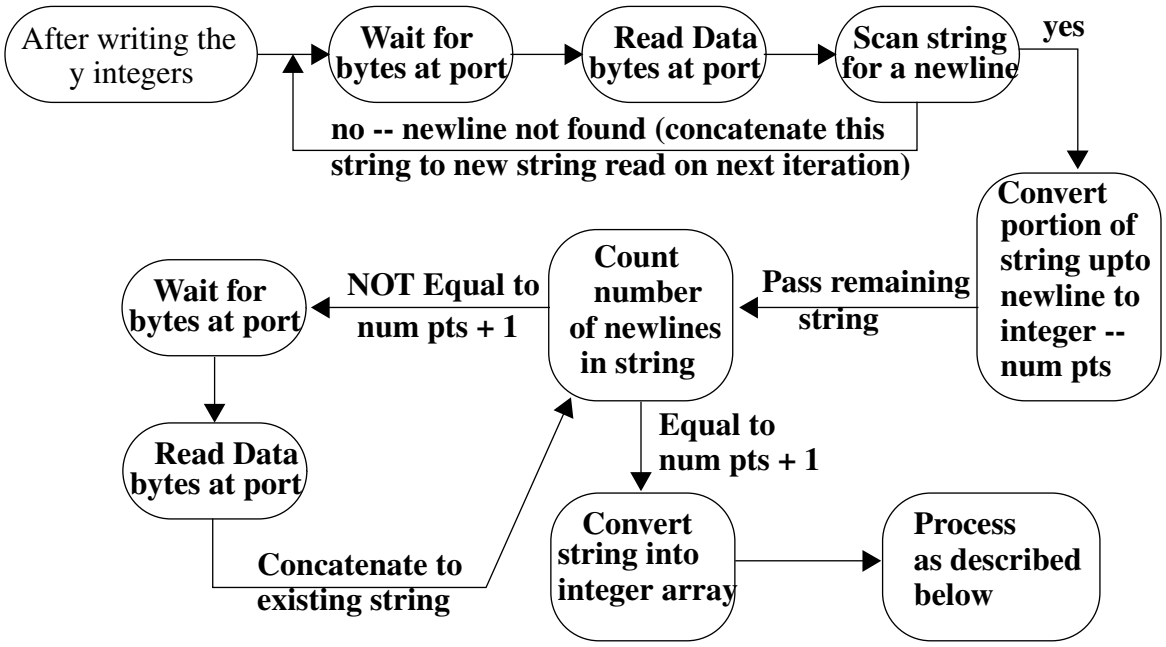
when you invoke the *sin* and *cos* functions. The constants 2 * pi/2^{THETA_WIDTH} represent "2*PI/num_pts" in the C code above. This indicates that the Core generated *sin* and *cos* functions already incorporate these constants as we have done in our C functions. THETA is the remaining portion of the argument, i.e., "j*i", that you will use as input to the *sin*/*cos* Core.
NOTE: See errata for correction to formula for *sin*/*cos* Core lookup table function.

The output of the *GetCos* and *GetSin* functions are 12-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, as described below.

The 12-bit values returned by the *GetSin* and *GetCos* functions need to be multiplied by the y data values (which are 8-bits), as shown in the C code above. The result of the multiplication is a 20-bit value (12+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). You will use 32-bits integers to store the *real* and *imag* components. The additional 12-bits available (32 - 20) will guarantee that no overflow occurs. (NOTE: If you've written you labview code in a modular way, you can extend the 8-bit values to 10-bits safely for higher precision).

Synthesize your project and eliminate syntax errors in your C code. Prepare to show me the labview code and EDK project with successful synthesis in class on the first due date.

PART II:
1) Write labview code to read back the *real* and *imag* components, convert them into *magnitude* and *phase*.
I used the following flow diagram to read the *real* and *imag* components sent from the FPGA.



In order to prevent data loss, this loop keeps grabbing the characters sent by the FPGA on the COM port until all values are provided.

Once you have converted the frequency domain data computed by the FPGA into integers, you first divide each of them by 2^{12}. As mentioned above, the *GetSin* and *GetCos* functions generate 12-bit signed integers that represent the range 1.0 to -1.0. This division restores the *real* and *imag* components to true *sin* and *cos* values.

FPGA_cos = cos(i*j*2*PI/2^{THETA})*2^{12} — C code generates this value given (i*j) as an argument.

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

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;

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

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

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 *imag* 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;
    }
```

Laboratory Report Requirements:

1) No written report required for this laboratory. Be prepared to demonstrate your project in class on the due date.

Grading:
LABVIEW coding style: 20%
Proper operation: 80%