LAB Assignment #5 for Co-Design (495/595)

Assigned: Tue., March 31, 2009
Due: Tue., April 7, 2009

Description: Build a hardware DFT that uses the communication mechanism and LABVIEW code from previous labs.

Transfer the y integer values of the waveform to the PPC using LABVIEW, compute a 'hardware' DFT using the PPC to 'serve' data to the hardware and send the values back to LABVIEW for conversion and viewing.

You can use the LABVIEW code you developed in lab #3 for sending the data and retrieving the results to the PPC directly. You will also use the scanf and printf loops in the C code that runs on the PPC from that lab. Instead of carrying out a software DFT, however, you will implement the DFT as a state machine in the reconfigurable logic. You will use the CoreGen to instantiate a Sin and Cos core within ISE to serve as the lookup table for the sine and cosine functions.

1) Under 'programs/ISE xx/accessories', click on 'CORE Generator'
2) Select 'new project' under the file menu
3) Enter a name and directory
4) In the CGP form, select Virtex2P, xc2vp30, ff896, -7. Click Ok.
5) Expand Basic Elements in the list on left in CORE Generator, then Math Functions, then Trig Functions.
6) Select Sine-Cosine Lookup
7) Click Customize and name the component SinCos.
8) Set Output Width to 14 and THETA Input Width to 9, select Function "Sine and Cosine", leave Memory Type "Distributed ROM", click next.
9) Leave options at defaults on next screen ("Non-registered and Symmetric"), click next
10) Unclick "Handshaking", click Generate.

Once the CORE Generator finishes, you need to copy the .vhd file to your pcores/xxx/src/hdl/vhdl directory and the .ngc file to pcores/xxx/devl/projnav directory. When you run EDK, you need to copy the .ngc file into the implementation directory. (Perhaps there is a better way -- I'm all ears!) The filename with extension '.vho' has a template for the module that you can cut-and-paste into your VHDL file with the FSM. You'll need to change the names in parenthesis to match those you use in your code.

Your state machine needs to implement the DFT C code from lab #3[1]:

```
for ( j = 0; j < num_pts/2; j++ )
    {
    real[j] = 0;
    imag[j] = 0;
```

1. 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.*

```
    for ( i = 0; i < num_pts; i++ )
       {
       real[j] += y_data[i]*GetCos(i*j);
       imag[j] += y_data[i]*GetSin(i*j);
       }
    }
```

My state machine uses 8 states with the following definitions

****

IMPORTANT (4/22): I developed my state machine to work with the Bus Master, NOT the client server, so your state machine may require a couple more states -- see IMPORTANT notes below.

****

start_state: Wait for *num_pts* to be placed in the *Value* register (see lab #4) -- save this number a *num_pts_register*. Wait for the 'command' bit to be set to '1'. Once set, goto **out_loop_state**.

out_loop_state: If *j* is equal to *num_pts_register*/2, goto **done**. Else set *real* and *imag* registers to 0 and goto **in_loop_state**.

in_loop_state: Compute *i*j* and save in a register (you need carry out the operations in the loop body in stages or you will end up violating timing constraints). Issue a read command for the data value at *y_data[i]* and goto **sin_cos_state**.

sin_cos_state: Wait for read to complete. Once completed, save the *y_data* value as well as the *sin* and *cos* values present on the output buses of the SinCos core into registers. Goto **accum_state**.

accum_state: Create *signed* versions of the *y_data*, *sin*, *cos*, *real* and *imag* register values. Add to the *real* and *imag* registers the products *y_data*cos* and *y_data*sin* where these arguments are the signed versions. The output of the SinCos core is a signed value -- you MUST tell the synthesis engine this otherwise you'll get incorrect results. Goto **inc_i_state**.

inc_i_state: If *i+1 = num_pts_register*, reset *i* and goto **write_state**. Else add 1 to *i* and goto **in_loop_state**.

write_state: Write both the *real* and *imag* register values to location *j*. You may need to add a control bit to the *Command* register to indicate which array to write. Or you can add *num_pts_register/2* to the *j* index for the *imag* index location. In this case, be sure to declare a single array, *real_imag[num_pts],* in the C code. Finally, increment *j* and goto **out_loop_state**.

done_state: Write *Command* register with *done* command, and loop forever.

IMPORTANT (4/21/09): If you use my example code for lab #4, then you will need to add a 'clear' command, otherwise consecutive read commands will not be processed because the C code triggers ONLY on a **change** to the command register. For consecutive reads, the command register does not change and so only the first read will be processed. To handle this, add a clear command:

```
go_bit <= slv_reg0(31);
get_bit <= dft_command(30);
put_bit <= dft_command(29);
done_bit <= dft_command(28);
clear_bit <= dft_command(27);
```

And change the code that updates the command register:

```
-- Load the command register when the hardware DFT wants to communicate
-- with the PPC.
            else
               if ( get_bit = '1' or put_bit = '1' or done_bit = '1' or clear_bit = '1' ) then
                  slv_reg0 <= dft_command;
               end if;
            end if;
```

Now, in the dft core state machine, when data_written is 001, then the update to the value register (slv_reg2) will occur on the NEXT clk. You can now issue the **clear** command, as follows:

```
   if ( data_written = '001' )
      dft_command <= "00000000000000000000000000010001";
      next_state <= wait_for_clear
```

Note that the dft_server.c code must remain as is, i.e., it ONLY executes a command when the command_reg **changes**. This is enforced with an if stmt as follows:

```
   if (command_data != old_command_data)
      ...
```

IMPORTANT (4/22/09): You will NEED to wait until the the PPC reads the clear command from slv_reg0 before you move forward to issue your next read command, otherwise you may end up writing the clear command and then overwrite it with the read command BEFORE the PPC has read the clear command.

```
   when wait_for_clear =>
      if ( data_read = "100" )
         (clear has been acknowledged -- it is safe now to issue your next read)
```

Laboratory Report Requirements:

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

Grading:
Coding style and comments: 20%
Proper operation: 80%