LAB Assignment #6 for ECE 495/595

Assigned: Tue., April 14, 2010
Due: Tue., April 21, 2010

Description: Implement a hardware DFT using a PLB bus master single beat controller.

The use of a client-server model as you've done for lab #5 in which the microprocessor 'served' data requests for the DFT core is one possible implementation of a communication mechanism between the reconfigurable logic and the PPC. In this lab, you'll investigate a second implementation that is in the true spirit of hardware/software codesign. The basic idea is to have the PPC transfer the data from LABVIEW and store it in memory (similar to lab #5), but in this lab the PPC will then simply wait until the hardware DFT finishes. Once completed, the PPC will transfer the data back to LABVIEW for post-processing and display.

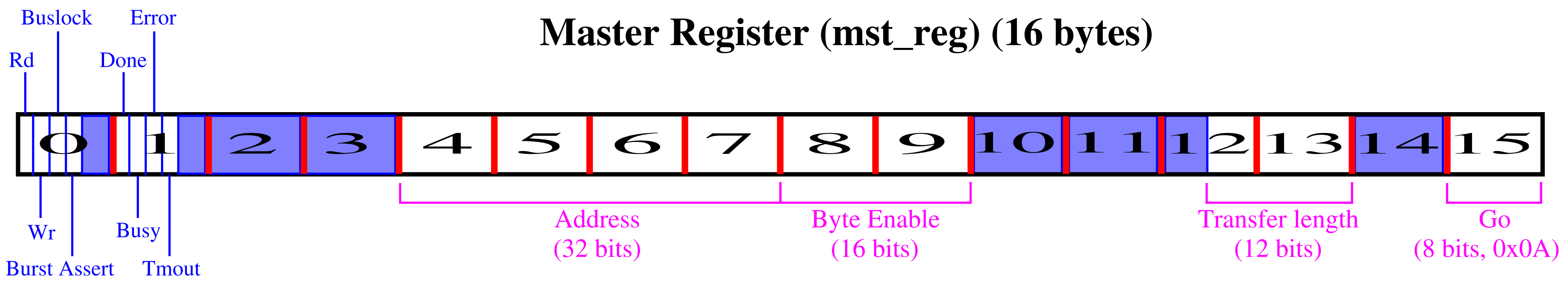
In order to implement this model, it will be necessary for the hardware DFT to become a **bus master**, because only bus master's can initiate memory read and write commands to the bus. Based on this description, the PPC and the hardware DFT will only need to synchronize, i.e., the hardware DFT needs to wait until the PPC has loaded memory and the PPC needs to wait until the hardware DFT completes. Also, since the hardware DFT will 'share' memory with the PPC, it will be necessary for the PPC to pass the base addresses of the arrays to the hardware DFT. This can be accomplished using the same mechanism you used in the client-server, i.e., create a set of software registers embedded in a PLB bus slave.

The following describes the sequence of steps needed to accomplish this.
1) Run BSB and create a base project, dft_bus_master, inserting only the UART as a peripheral.

2) Run Create Peripheral. At the **IP Interfaces Services** screen, you'll want to click 'Interrupt Control', 'User Logic Software Registers' and 'Bus Master' (at the bottom) checkboxes. Under the **Interrupt Services** screen, uncheck 'Use Device ISC' and set the 'Capture mdoe' to *Rising Edge Detect*. I added 3 registers as we did for lab #4 under the **User S/W Registers** screen, one to transfer the number of points and then store the base address of the input array, one for each of the two base addresses of the output arrays (see description below). I used the defaults under the other screens.

3) Edit the user_logic.vhd file and make the modifications necessary to allow the DFT engine to control the bus master logic. Similar to the single beat PLB slave, the default template provided allows only the PPC to read/write the master control register. The bus master template is somewhat more complicated than the slave module (which is also included in user_logic.vhd because you indicated software registers). It is important that you carefully analyze the template code so that you can make the proper changes. The following describes the basic components and operations.

The 'center piece' of the bus master logic is the *mst_reg*, the **master register**. The layout of the register is shown above. It is 16 bytes in length and contains the following fields:
Byte 0: Control register (only most significant 4 bits defined)
Rd: read bit, when set, the control logic will initiate a read operation to a slave on the bus.
Wr: write bit, when set, the control logic will initiate a write operation to a slave on the bus.
Buslock: locks the bus when set (not needed, preserve contents).



Bust Assert: indicates a burst transfer when set (not needed in a single beat master).

Byte 1: Status register (only most significant 4 bits defined)

Done: Transaction completed

Busy: Transaction in progress

Error: Error in transaction

Tmout: Timeout error.

Bytes 2 and 3: Unused

Bytes 4-7: Address register

Memory address to read or write

Bytes 8 and 9: **Byte Enables**

Upto 16 byte enables are provided.

Bytes 10 and 11: Unused

Bytes 12 and 13: Transfer length in bytes

For single beat masters, always set to 4

Byte 14: Unused

Byte 15: Go register

Initiates a bus master operation, i.e., starts the state machine described below. This register is write-only (can not be read by the PPC).

The example code in ‘drivers/dft_bus_master_v1_00_a/src/dft_bus_master_selftest.c’ and ‘drivers/dft_bus_master_v1_00_a/src/dft_bus_master.c’ shows how the PPC can control this register for use as a *pseudo-DMA* controller. The self test module sets up two 4-byte buffers (aligned at 128) *SrcBuffer* and *DstBuffer*, loads them with default values using the loop:

```
for ( Index = 0; Index < DFT_MASTER_CORE_SELFTEST_BUFSIZE; Index++ )
{
  SrcBuffer[Index] = Index;
  DstBuffer[Index] = 0;
}
```

It then calls a subroutine in *dft_bus_master.c* to instruct the master to read a word into the internal FIFO embedded in the master, and then ‘busy-waits’ on the ‘done’ bit in the mst_reg.

```
DFT_MASTER_CORE_MasterRecvByte(baseaddr, (Xuint32) SrcBuffer, DFT_MASTER_CORE_SELFTEST_BUFSIZE);
```

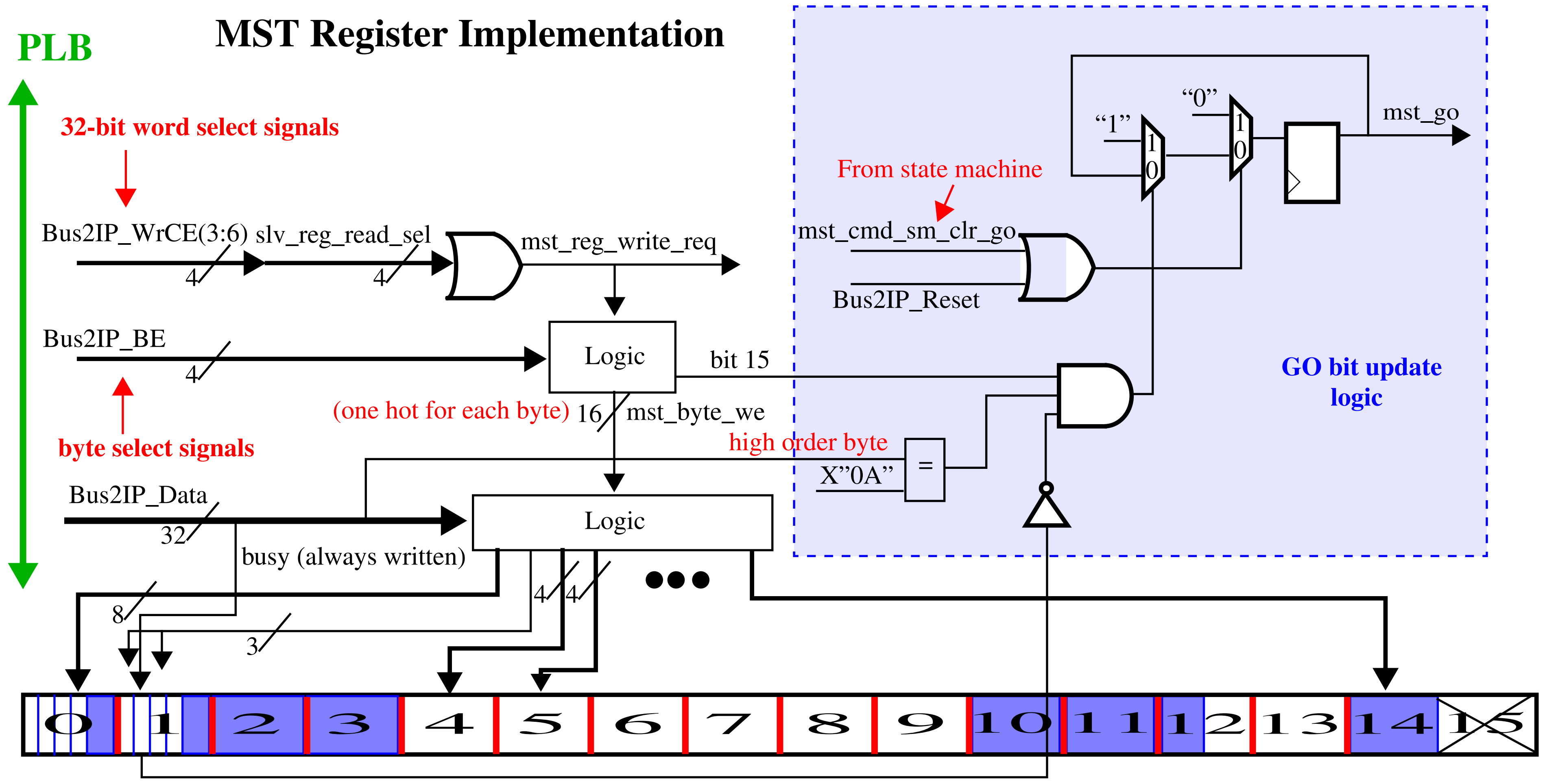
```
   while ( ! DFT_MASTER_CORE_mMasterDone(baseaddr) ) {}
```

It then instructs the bus master to write the 4-bytes read into it's FIFO to the destination buffer.

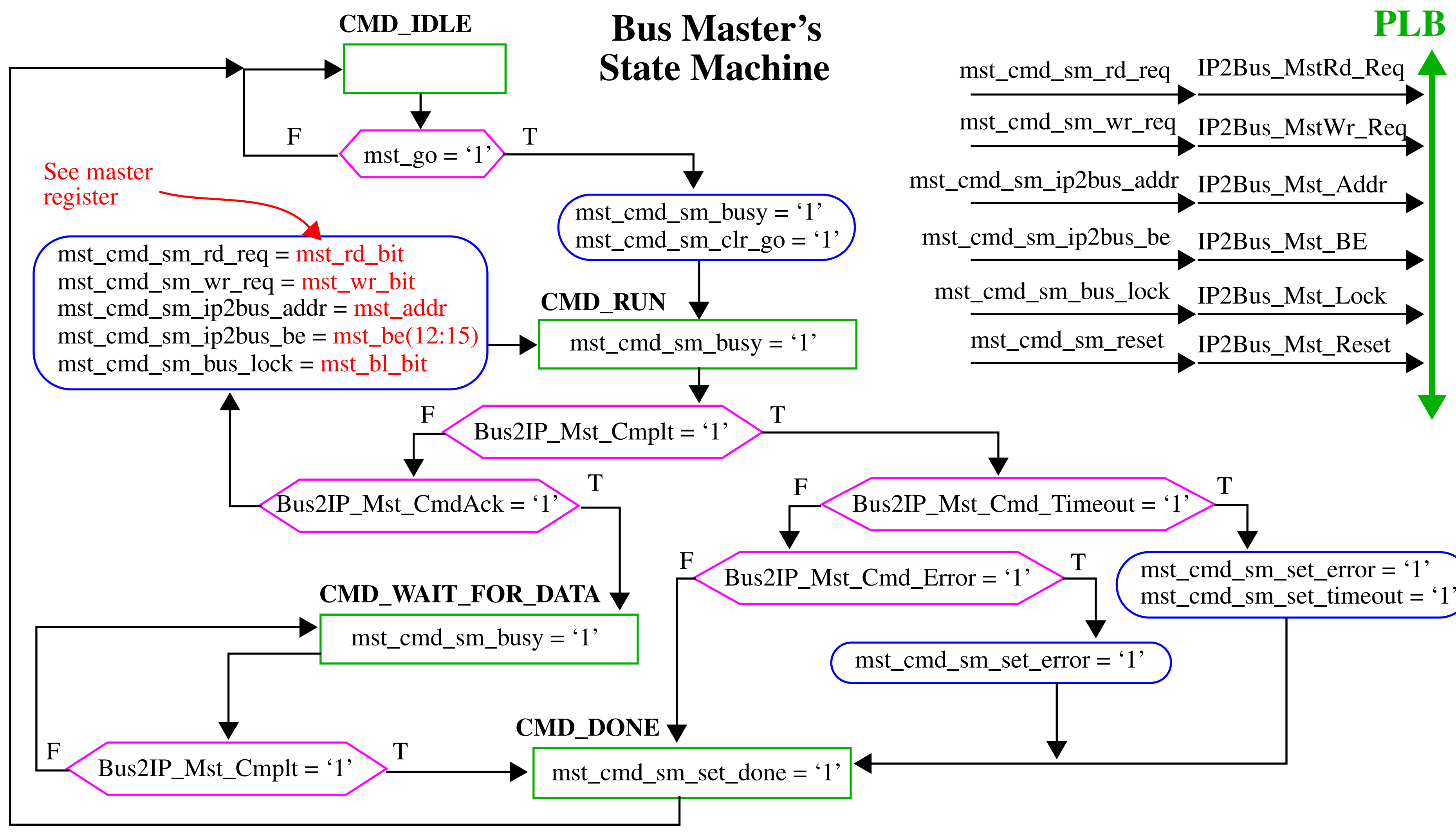
```
   DFT_MASTER_CORE_MasterSendByte(baseaddr, (Xuint32) DstBuffer, DFT_MASTER_CORE_SELFTEST_BUFSIZE);
   while ( ! DFT_MASTER_CORE_mMasterDone(baseaddr) ) {}
```

The 'RecvByte' code in 'drivers/dft_bus_master_v1_00_a/src/dft_bus_master.c' performs 5 operations:

a) Write '1' into the 'Rd' bit of mst_reg.
b) Write SrcBuffer address into 'addr' of mst_reg.
c) Write 0xFFFF into 'byte enable' of mst_reg.
d) Write 4 into 'transfer length' of mst_reg.
e) Write 'go' byte.



The **MST Register Implementation** figure gives the schematic of the template code in ‘user_logic.vhd’ for implementing PPC updates to the master register. You’ll need to modify this in order to allow control to the master from within your hardware DFT logic. I just added to this code (didn’t delete anything) to preserve the ability of the PPC to control it, but this is not required.



Although you will **not** need to modify the generated **Bus Master’s State Machine**, it is instructive to understand how it works (see schematic above). The machine remains in IDLE until the ‘go’ signal. It sets the busy bit to ‘1’ and the go bit to ‘0’ on the rising edge. If the complete bit (‘Cmplt’) is not set and the controller (parent) has not acknowledged a command, then issue a command based on the contents of the master register. If the complete bit is not set but the command is acknowledged, then go to the ‘wait for data’ state. If ‘complete’ bit is set and we are still in this state, set error/timeout and proceed to ‘done’, resetting ‘busy’ to 0. So to succeed, the command must be acknowledged before the controller sets the ‘complete’ bit.

The last change you'll need to make concerns the FIFO. Incoming and outgoing data are normally stored there (when the PPC issues commands to the master reg). The incoming data is placed on *Bus2IP_MstRd_d* and is acknowledged on *Bus2IP_MstRd_src_rdy_n*. Outgoing data is placed on a separate bus, *Bus2IP_MstWr_d*, and acknowledged with *Bus2IP_MstWr_dst_rdy_n*. Again, you can choose to add to the functionality that exists or change it, i.e., eliminate the FIFO.

4) Add a module that implements the hardware DFT. My module **port map** is given below. Your's does not need to be exactly the same but these are the signals I need to implement the hardware DFT.

```
DFT_CORE_INSTANCE: DFT_CORE
   generic map
      (
      DFT_SLV_DWIDTH => C_SLV_DWIDTH,
      DFT_MST_AWIDTH => C_MST_AWIDTH,
      DFT_MST_DWIDTH => C_MST_DWIDTH
      )
   port map
   (
   Clk => Bus2IP_Clk,
   Reset => Bus2IP_Reset,
   dft_busy => mst_cmd_sm_busy,
   dft_go => dft_go,
   dft_intern_req => dft_intern_req,
   dft_time_addr => slv_reg0,
   dft_real_addr => slv_reg1,
   dft_imag_addr => slv_reg2,
   dft_read_cmd => dft_read_cmd,
   dft_write_cmd => dft_write_cmd,
   dft_out_addr => dft_out_addr,
   dft_read_data => dft_read_data,
   dft_write_data => dft_write_data,
   dft_read_data_xfer => dft_read_data_xfer,
   dft_write_data_xfer => dft_write_data_xfer,
   dft_interrupt_done => dft_interrupt_done
   );
```

You will need to modify the **start_state** and **done_state** states of the state machine that I outlined in lab #5. **start_state** mods: If you choose to use the 3 slave registers for the base addresses of the input and output arrays, then you can implement the 'go' signal by montioring the values in these slave registers. For example, when your C code writes the last address, e.g., the *imag* array address, to the third slave register, your state machine can be programmed to move to **out_loop_state** and begin the hardware DFT operation. Unless you added 4 software registers, you'll also need to either hardcode the number of points in your VHDL code or implement a scheme in **start_state** which first reads the *num_pts* in slv_reg0 followed by the monitoring operation described above for implementing the 'go' signal. Althrough the second method is better because it is portable to handle arrays of different sizes, either implementation is acceptable.

The 'done' signal to the PPC needs to be implemented using an *interrupt* in order to eliminate any contention to the bus and/or memory. If you implement a scheme in which the PPC loops reading a slave register (whose pointer is declared as *volatile*) to determine when the hardware DFT completes, then the hardware DFT and the PPC will contend for bus cycles, which will slow the hardware DFT down significantly. If you implement a scheme that checks a shared (*volatile*) memory location, then both the PPC and hardware DFT contend for both the bus and memory. So neither of these solutions is ideal. Interrupts may handle this situation better. The basic idea is to modify your **done_state** to generate an interrupt to the PPC. Inside the interrupt service routine (ISR), you will change a global variable, *interrupt_flag*, to 1. The flow of your main program is to 1) initialize *interrupt_flag* to 0, 2) transfer the input data from LABVIEW, 3) start the hardware DFT by writing the number of points and the base addresses to the slave registers and, 4) loop until *interrupt_flag* changes to 1. Note that you should **not** declare this variable as *volatile* (in fact, doing so might be self defeating because any reads of *interrupt_flag* may force a memory read cycle). The hope is that the compiler 'caches' *interrupt_flag* in a register, noting that the ISR may perform an update to it directly, thereby avoiding any memory reads and writes. This requires the compiler to perform global optimization, and I'm not sure if this is something your compiler can do. Anyone who analyzes the assembly to determine what actually happens will get bonus points.

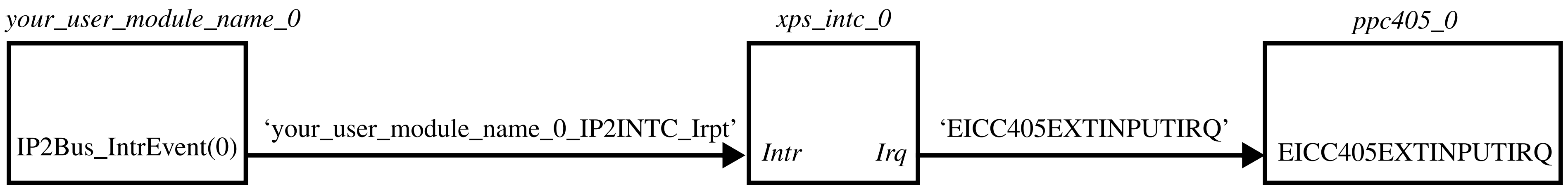
5) Re-import the peripherial (see UNM EDK tutorial 2). On the **Bus Interfaces** screen, leave the 'PLBV46 Master (MPBL)' and 'PLBV46 Slave (SPBL)' boxes checked (default). Other than the changes described in tutorial 2, I made no other changes.

6) Once imported, go to 'IP Catalog' and add the 'XPS Interrupt Controller' under the 'Clock, Reset and Interrupt' list item. Also, under the 'Project Local pcores', add the 'User' module you created.

7) Under the 'Bus Interfaces' tab under 'System Assembly View', connect the *xps_intc_0* component to 'plb0' and connect both the slave and master of the *your_user_module_name_0* component that you added to 'plb0'.

8) Under 'Ports', expand the *ppc405_0* element and scroll down to *EICC405EXTINPUTIRQ* and select 'New Connection'. The connection name should be given as 'ppc405_0_EICC405EXTINPUTIRQ' or 'EICC405EXTINPUTIRQ'. Next, expand the *xps_intc_0* and click on the 'Intr' component. In the dialog that pops up, click on the 'your_user_module_name_IP2INTC_Irpt' signal in the left list box and click the '+' sign in the middle. This should add the signal to the right list box. Click okay. If you expand the *your_module_name_0* component under 'Ports', you should see this signal name given to the 'IP2INTC_Irpt' component. Last, click the 'Irq' element under *xps_intc_0* component and connect it to the 'ppc405_0_EICC405EXTINPUTIRQ' signal name. These operation effectively connects the components together as shown below.

9) Under'Address', give 1K of memory to *your_user_module_name_0* and 64K of memory to *xps_intc_0* (probably can use less here if you want). II would also increase the *xps_bram_if_cntrl* space to 128K. Finally, press the 'Generate Addresses' tab.



10) Do the usual for the software application part, i.e., increase the stack and heap to 0x4000 and 0x2000 respectively. If you use the 'xil_printf', it is probably not necessary to do this but I recommend it just to be safe. Be sure to look at the example 'selftest' code in the drivers/xxx/src directory, particularly the interrupt test code. I've posted the *xps_intc_0* data sheet on my web site. It contains some documentation on the registers in *xps_intc_0* that are written by the code given in the UNM EDK tutorial 2. You'll need to use this code -- I only needed to change the references 'SWITCH_DEBOUNCER' to my module name. The constants used are defined in several header files. Those starting with 'XPAR_' can be found in the ppc405_0/include/xparameters.h file and the others in the drivers/xxx/src/xxx.h file.

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%