LAB Assignment #4 for ECE Co-Design (495/595)

Assigned: Wed., March 3, 2010
Due: Wed., March 29th, 2010

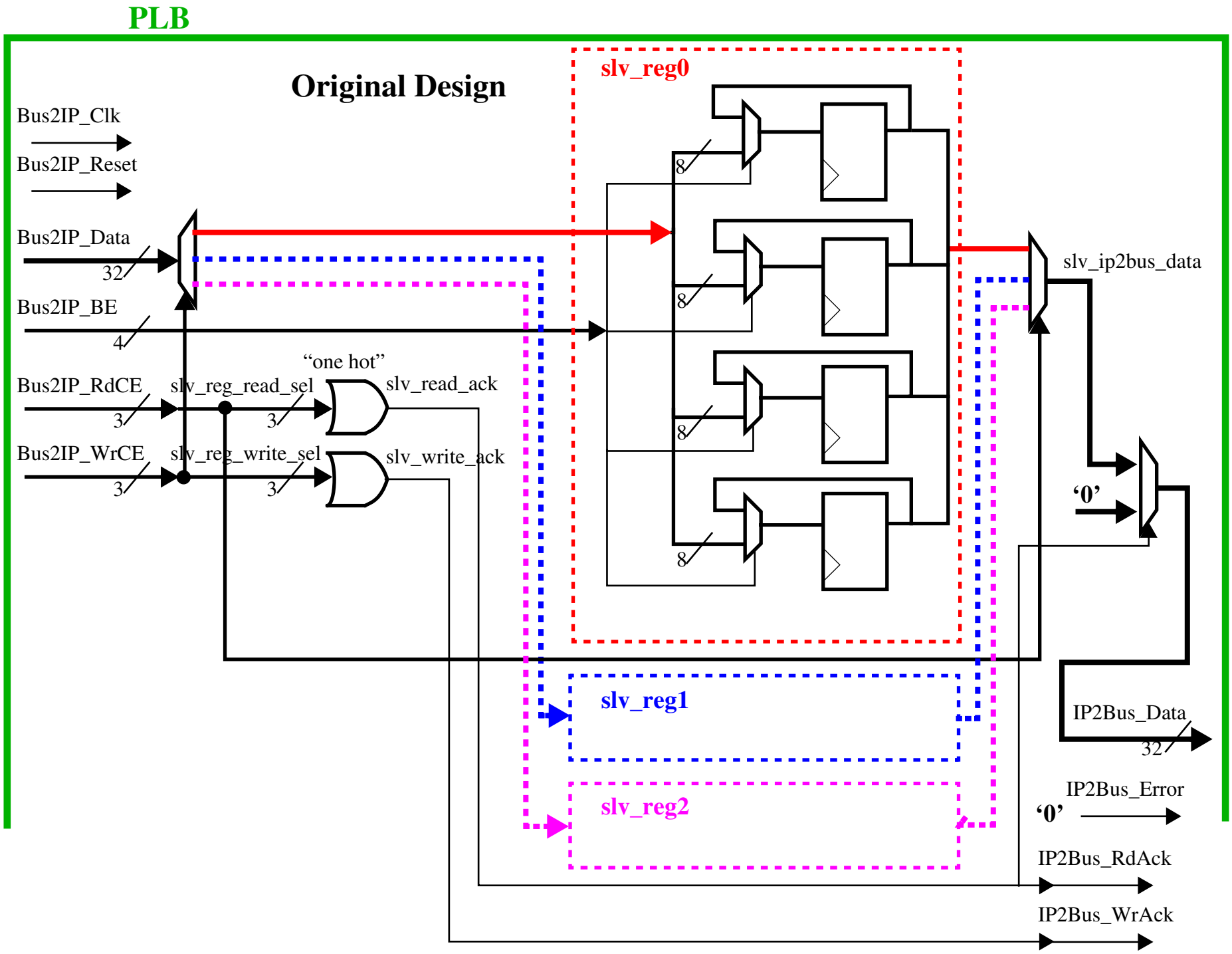
Description: Build a state machine the implements a communication protocol using software registers with the PPC.

This lab utilizes the UNM EDK Tutorial 2, in which you build a custom peripheral. (I found the tutorials at http://www.fpgadeveloper.com useful as well). The basic idea is to build a client-server model between a hardware state machine and the PPC designed to allow data to be exchanged. The PPC plays the role of the server, which manages an array of data. The hardware state machine issues read and write commands to the PPC through a set of addressable registers embedded in a PLB bus slave peripheral. The PPC responds by moving values between the array and these registers.

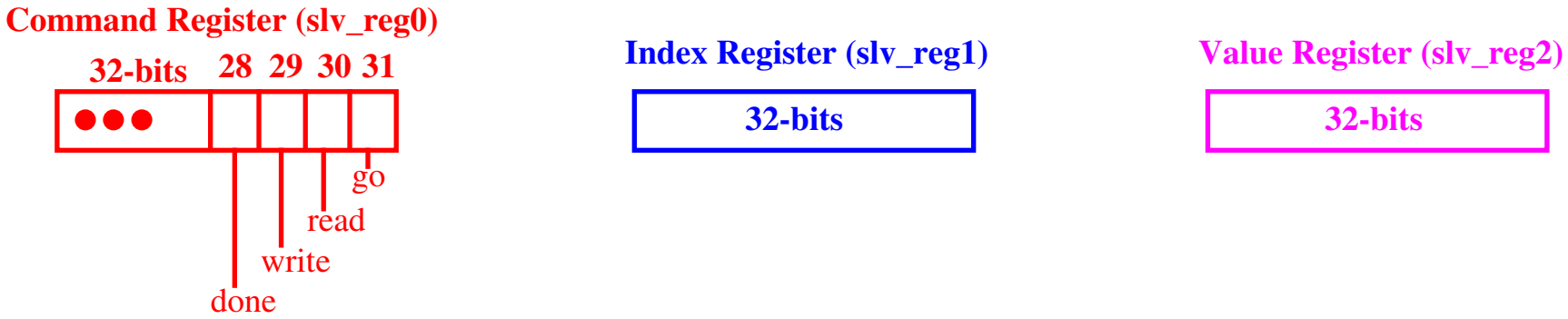
The following describes the steps involved in this lab:
1) Create a custom peripheral following the UNM EDK Tutorial 2. When you create the base project using BSB, add only the UART, i.e., you will NOT need to pushbutton, leds or interrupts as described in the tutorial. On the **Name and Version** screen, use *micro_assist_dft* instead of *switch_debouncer*. Although you are not building the hardware DFT in this lab, you will do so in the next lab, where this name will have more meaning. On the **IP Interfaces Services Screen**, you do NOT need to check the 'Interrupt Control' box -- only the 'User Logic Software Registers'. On the **User S/W Registers** screen, set the number of registers to 3. When you run ISE and open the project in the projnav directory, you will see only the PLBV46_SLAVE_SINGLE and USER_LOGIC modules under the *micro_assist_dft* module. Follow the tutorial instructions to add a module, naming it *dft_core* instead of *switch_debouncer_core*. You'll need to put your state machine in this module, as described below. You'll also need to change the functionality of the USER_LOGIC module. You will probably want to modify it first and write a simple driver in the *dft_core* to test your modifications.

2) User Logic Module modifications. The VHDL code generated by the **Create Peripheral Wizard** in the *user_logic.vhd* module is shown by following schematic. The logic is designed to enable software reads and writes to the three registers, named *slv_regx*. **You should carefully study and fully understand the operation of this code**, simulate it if necessary. For example, the code enables 'byte-level' reads and writes using a 'for loop' construct. More importantly, reads and writes to the registers are indicated by the PLB using the *Bus2IP_RdCE* and *Bus2IP_WrCE* signals. The proper interpretation of these signals is that, when one of the write bits is '1' in *Bus2IP_WrCE*, then on the NEXT rising edge of the clock, the data on *Bus2IP_Data* will be latched into the designated *slv_regx*. When one of the read bits is '1' in *Bus2IP_RdCE*, then the appropriate register is MUXED to drive the IP2Bus_Data, which samples the data on the NEXT rising edge. You need to modify the automatically generated VHDL code in order to enable writes to occur from your state machine. At this point, the existing code only enables the PPC to communicate to the hardware.

The client-server memory model needs to implement the following functionality. The slave registers, 0, 1 and 2 are named *Command*, *Index* and *Value*, respectively, as shown in the following schematic. The *Command* register defines four operations using the least significant bits, 28, 29, 30 and 31. Bit 31 is called the 'go' bit and is special. It keeps the hardware state machine in the ***idle*** state until it is set. In the next lab, this will be necessary to allow the PPC to fetch the y values of the wfm from LABVIEW. Once the PPC sets the 'go' bit to '1', it enters a loop where it serves as a memory controller, effectively becoming a slave to the hardware DFT. In the loop, it will read the *Command* register and respond to read and write commands. The hardware DFT sets the 'read' bit when it wants the PPC to fetch a value from its array. The specific value will be given by the 'index' that the hardware

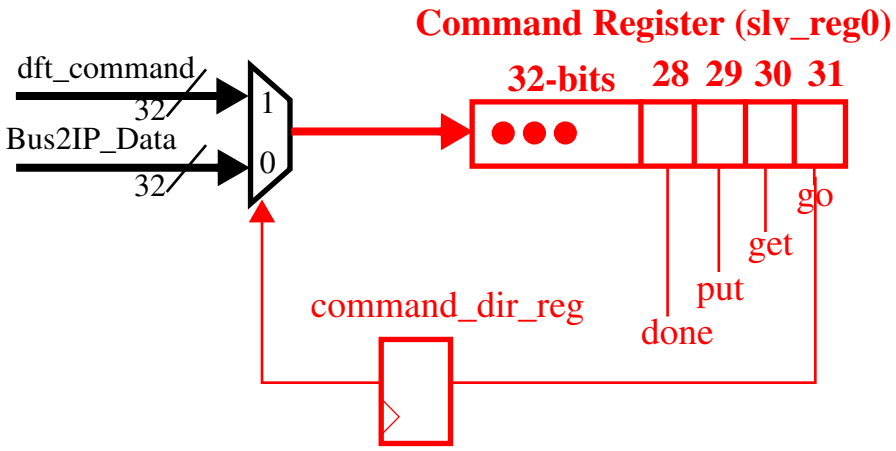

PLB
Original Design
slv_reg0
Bus2IP_Clk
Bus2IP_Reset
Bus2IP_Data
32
Bus2IP_BE
4
"one hot"
Bus2IP_RdCE
slv_reg_read_sel
slv_read_ack
3
3
Bus2IP_WrCE
slv_reg_write_sel
slv_write_ack
3
3
8
8
8
8
slv_ip2bus_data
'0'
slv_reg1
slv_reg2
IP2Bus_Data
32
IP2Bus_Error
'0'
IP2Bus_RdAck
IP2Bus_WrAck

DFT has placed in the *Index* register. The PPC responds by placing the designated array value given by that index value into the *Value* register. Writes are implemented in a similar fashion, i.e., the hardware DFT sets the write bit in the *Command* register, the PPC reads the *Command* register and writes the integer in the *Value* register to the array at the position given by the *Index* register. The hardware DFT will set the 'done' bit in the *Command* register once the DFT is computed. The PPC will then exit the loop and print the values in the array to the UART. In the next lab, the PPC will write the values to LABVIEW using the UART port.



In order to implement this functionality, you'll need to modify this schematic. For example, the following shows one possible modification that enables the PPC to write the 'go' bit of the *Command* register (Note: the 'command_dir_reg' is optional, i.e., you can feed back command register bit 31 directly to the select MUX if you like). Once set, further writes by the PPC are ignored and the hardware DFT is able to write it until the go bit is set back to 0. Your state machine (in *dft_core*) should remain in the initial state until the 'go' bit in the *Command* register is set to '1'. Your state machine must keep this bit set to '1' through its entire operation, i.e., until it reaches the final state (to be described).

From this description, several conclusions can be made. The *Command* register needs to accept input from the BUS2IP_Data bus (as shown above) until the 'go' bit is set. The *Index* register is written ONLY by the state machine -- never by the PPC. The *Value* register is written by the PPC for reads and is written by the state machine for writes. I would recommend that you draw a algorithmic state diagram to represent this functionality and write a simple driver to test each function, before writing the state machine in *dft_core*, as described below.



3) Write a simple state machine in *dft_core* that performs the following functions.
a) Wait until the 'go' bit in the *Command* register is set to '1'
b) Issue a read command for one of the elements in the array (to be discussed below).
c) Wait for the value.
d) Add 1 to the value and write it back to the same (or another memory location) in the array.
e) Issue the 'done' command.

The following is the port map that I used for my *dft_core* module (as given in the *user_logic.vhd* module). This is only provided to help you understand what you might need. Your module definition may include other signals.

DFT_CORE_INSTANCE: DFT_CORE

```
      generic map
         (
         SLV_DWIDTH => C_SLV_DWIDTH
         )
      port map
       (
       Clk => Bus2IP_Clk,
       Reset => Bus2IP_Reset,
       data_written => slv_reg_write_sel,
       data_read => slv_reg_read_sel,
       command_bit => slv_reg0(31),
       value_reg => slv_reg2,
       dft_command => dft_command,
       dft_index => dft_index,
       dft_value => dft_value
       );
```

4) Write a C program that implements the server. Here are some key components. Global declarations:

```
unsigned int *dft_command_reg =
              (unsigned int *) XPAR_MICRO_ASSIST_DFT_0_BASEADDR;
unsigned int *dft_index_reg =
              (unsigned int *) XPAR_MICRO_ASSIST_DFT_0_BASEADDR + 1;
unsigned int *dft_value_reg =
              (unsigned int *) XPAR_MICRO_ASSIST_DFT_0_BASEADDR + 2;
```

These give you pointers to the three hardware registers. Note that you add '1' and '2', not '4' and '8' -- the compiler knows these point to 32-bit locations.

Declare the following array:

```
int vals[3];

vals[0] = 0x12;
vals[1] = 0x20;
vals[2] = 0x40;
```

Issue the 'go' command (note: the most significant bits in the hardware correspond to the least significant bits in the software!):

```
*dft_command_reg = (unsigned int) 0x00000001;
```

Then, enter a 'while(1)' loop and 'busy wait' on the value of the *Command* register, waiting for one of the three commands, 0x3, 0x5 and 0x9. (Note: since the state machine MUST keep the 'go' bit set, 0x3 indicates a read, 0x5 indicates a write and 0x9 indicates done.

IMPORTANT: BE SURE TO DECLARE ALL VARIABLES THAT ARE UPDATED BY THE RECONFIGURABLE LOGIC AS 'volatile' or the compiler may 'optimize' some statements out.

NOTEs: (Will be updated as needed)

a) After creating the project using BSB, set the BRAM memory size to 128K as we did in the last lab.

b) After creating and importing the IP (see UNM tutorial 2), set the memory size of the registers to 256 (smallest value) and press 'Generate addresses'. The base address of my registers was set to 0xfffd0000.

c) The directory 'drivers/micro_assist_dft_v1_00_a/src' contains some useful functions (I didn't need to use any for this project, but you should be aware of them).

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%