Hardware/Software Codesign for Wireless Systems (E168b)

Harris

Lab 1

Memory Mapped I/O Interfaces

Introduction

In the last lab, we got an introduction to the Xilinx Virtex II Pro Development System and programming it using both Verilog and C. In this lab, we will develop a memory mapped interface, which will give an introduction to a hardware/software co-design. The software aspect is performed by Platform Studio whereas the hardware is implemented using Project Navigator.

In this lab, you will

1. Follow a tutorial to learn hardware software codesign by implementing an XOR function in hardware and invoking it with a C program.
2. Write a "popcount" accelerator on your own, using hardware to rapidly count the number of 1's in a 32-bit word.

1. **XOR:** The following steps will guide you through how to find the XOR of the two bits, using a memory mapped interface between the hardware and software. The hardware system consists of an XOR gate and the memory mapped interface. The processor writes to one register. The hardware computes the XOR of the two LSBs of that register. The results are obtained by reading a second memory-mapped register. (This second register doesn't actually have any flip-flops, but rather returns the value computed by the XOR function.) Your program will write the values of two DIP switches to the input register and display the result on the LEDs or on the HyperTerminal window.

   Follow the steps below to implement this.

   - Open Platform Studio and create a new project using the Base System Builder (BSB). Browse to your Charlie directory and create you project in a new folder – lab1_xx. Set the repository path to \\charlie.hmc.edu\Courses\Engineering\E168b\V2P_CD\lib or \\charlie.hmc.edu\Courses\Engineering\E158\V2P_CD\lib if you don't have access to the former. You can also get download it from http://www.xilinx.com/univ/XUPV2P/lib/lib_rev_1_1.zip and extract the files to your Charlie directory. Click Ok after selecting the repository path. (If you have trouble, try copying the repository onto the C drive rather than accessing it from the network.)
   - Now select 'I would like to create a new design' and choose the XUP Virtex-II Pro Development System as the board name and C as the board revision.
   - Select PowerPC as the processor, with 300 MHz as the Processor Clock Frequency and the DATA and Instruction Memory set to 16 KB (Similar to Lab 0).
   - Again deselect all other peripherals except the RS232, the LEDs, and the DIP Switches.
   - Deselect 'Memory Test' but make sure 'Peripheral Selftest' is selected.

- Finally click on ‘Generate’ and then ‘Finish’. This creates the files for the selftest for the LEDs and the DIP switches. You can download the design and test these to ensure that the board is working. Otherwise, select Edit.
- Again in this lab, you want to create a new software application project like you did in lab0. In the project name, type in ‘lab1_xx’ and select the Processor type to ppc405_0.
- Right-click on ‘Project: TestApp_Peripheral’ and unselect ‘Mark to initialize BRAMs’. Right-click on ‘Project: lab1_xx’ and select ‘Mark to initialize BRAMs’. This activates your project instead of the selftest.
- Now, right-click on Sources in Project: lab1_xx, and select add new file. Browse to your Lab1 folder and name the file as lab1_xx.c .
- Copy the following code into the c file. This code will read the switches. Go through this code and make sure you understand it. It is similar to the code used for Lab0.

```
/* lab1_ak.c 20 Dec 2006 */

/* Memory mapped IO interface
   XOR of the 2 LSB's */

/* #include files */

/* define I/O devices */
#include "xparameters.h"

/* define general purpose I/O functions for accessing peripherals */
#include "xgpio.h"

/* define printf */
#define printf xil_printf       /* A smaller footprint printf */


/* Main */
int main(void)
{
   int switches;
   XGpio gpio_switches;

   printf("\r\nPreparing to read switches\r\n");

   XGpio_Initialize(&gpio_switches, XPAR_DIPSWS_4BIT_DEVICE_ID);
   XGpio_SetDataDirection(&gpio_switches, 1, 0xFFFFFFFF); // input

   switches = XGpio_DiscreteRead(&gpio_switches, 1);
   printf ("Switches = %d\r\n", switches);
}
```

- Now, let’s run this to ensure its working. Expand ‘Compiler Options’. You will notice that ‘Linker Script’ does not have an associated file with it. That will cause a problem. So, to overcome that, like lab0, click on **Software -> Generate Linker Script -> lab1_xx**. Click Generate.
- Now select ‘download bitstream’. This creates the files for the project, synthesizes, implements and downloads it onto the board. Make sure the board is on and the hyper

terminal window is open and configured to display the results. Check the results to make sure there is no problem.

- Next, we want to write the value read from the switches to an address in memory and then perform the XOR of the 2 LSBs and write it to another location in memory. This value will then be read and displayed as the result.
- Click on **Hardware -> Create or Import Peripheral.** On the first screen, click Next. Then select 'Create templates for a new peripheral'. Click Next and now select 'To an XPS Project'. Give it a name, 'lab1_xor_xx'. Let the three revisions be set to '1', '00' and 'a' in that order.
- On the next screen, select the Bus Interface as the On-Chip Peripheral Bus (OPB).
- On the next screen, unselect everything else except 'User logic S/W register support'.
- Select the number of software accessible registers to 2, each one being 32 bits. Select 'Enable posted write behavior'.
- Let everything on the next screen be at its default setting.
- Unselect 'Generate BFM simulation' on the following screen.
- On the next screen, select Generate user logic in Verilog instead of VHDL (Figure 1). This may generate a warning – you can ignore that. If you are comfortable using VHDL, you may leave this option and proceed to the next screen. This option simply writes the user logic code in Verilog instead of VHDL.

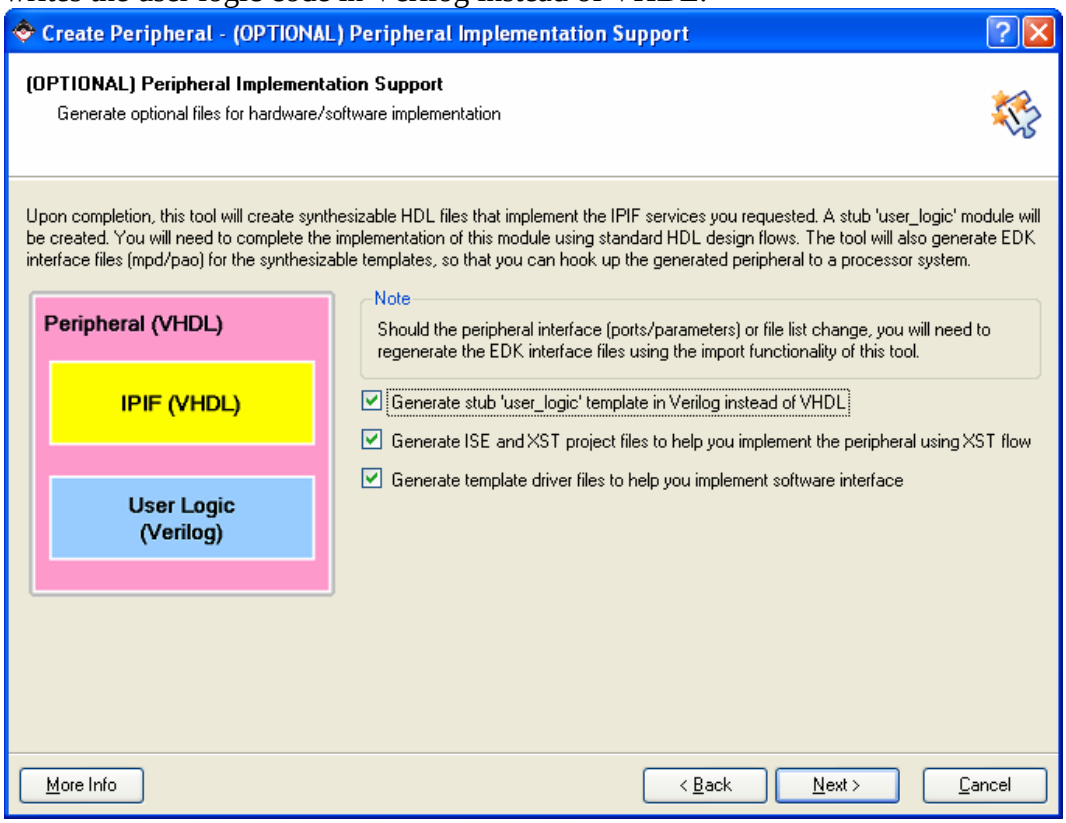

Figure 1: User logic created in Verilog instead of VHDL

- Click Next and then Finish.
- Now, the additional peripheral, 'lab1_xor_xx' has been created. However, you now need to add it to your project. Do this by clicking on the **IP Catalog** tab and expanding **Project Repository** to find **lab1_xor_xx.** Right clock and select 'Add IP'.

- Now on the window on the right hand side, open the ‘System Assembly View1’ tab. (Refer to Figure 2). Expand ‘lab1_xor_xx_0’. Change the ‘No Connection’ for the SOPB to ‘opb’ to hook your new peripheral up to the On-chip Peripheral Bus.

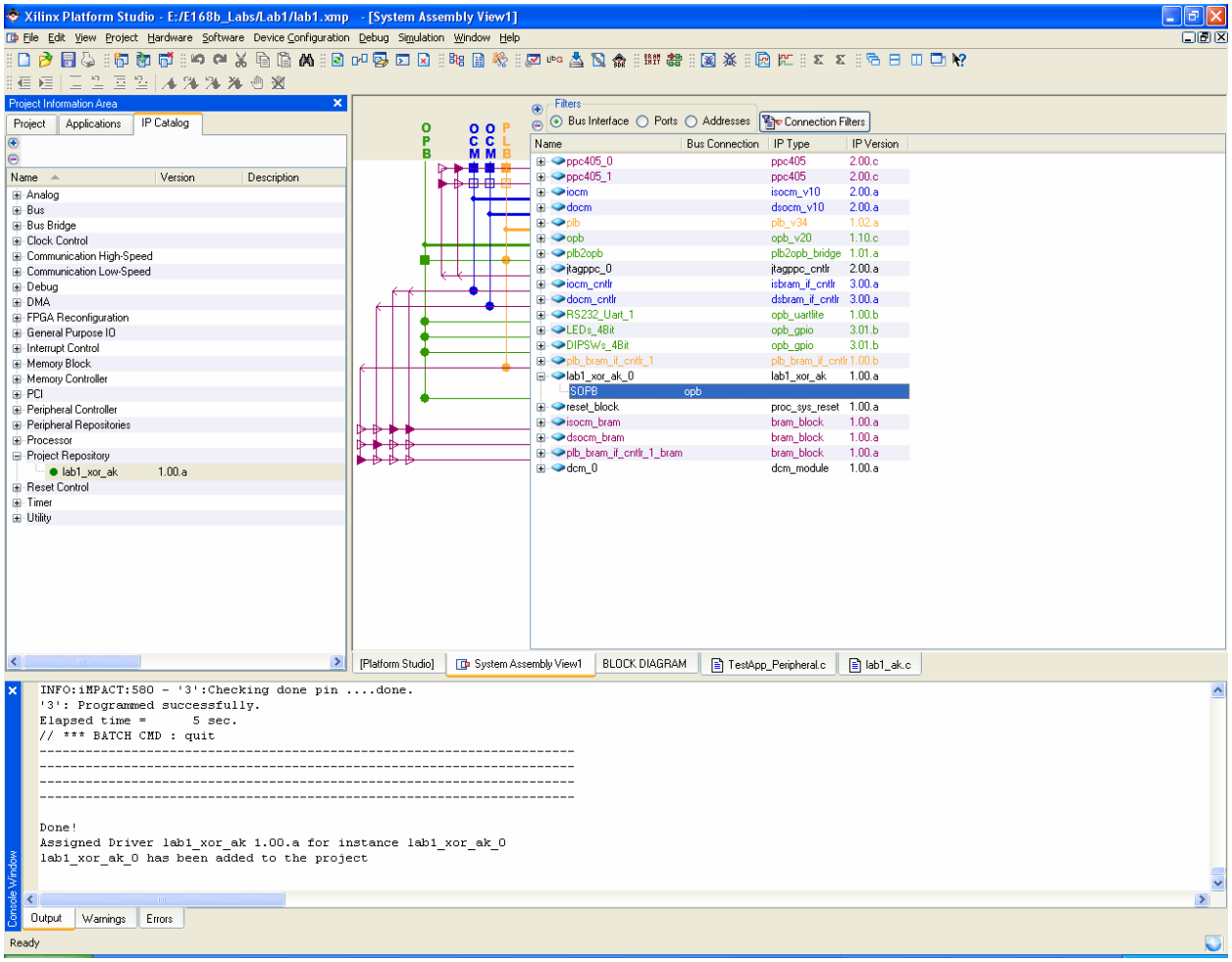

Figure 2: Adding the lab_xor_xx peripheral

- Now click on Ports, and select lab1_xor_xx_0. Click on ‘Filters Applied’ and select ‘All’.
- Now click on ‘Addresses’ and click on ‘Generate Addresses’. This generates the addresses for all the components in the system, including the peripheral just created by us. It will display a warning but ‘Ignore’ it.
- Now let’s go back to working on the code, by clicking on the ‘Applications’ tab on the left window and the ‘lab1_xx.c’ tab on the right window.
- On creating the xor peripheral, BSB generates a selftest code for it. You can access it in your lab folder - …\Lab1\drivers\lab1_xor_xx_v1_00_a\src . Go through the files in the folder to get acquainted with the syntax for the xor peripheral.
- As is evident from the files, “`LAB1_XOR_XX_mWriteSlaveReg0(BaseAddress, Value)`” will write to register0 in memory and “`LAB1_XOR_XX_mReadSlaveReg1(BaseAddress)`” reads the result from register1.

The Base Address is simply the address assigned to the peripheral. You need this to access the peripheral. This base address is saved in the xparameters.h file and can be accessed using 'XPAR_LAB1_XOR_XX_0_BASEADDR' .

- Thus, we can now include additional code to write to memory, perform the xor and read back from memory. Include the following lines of code in appropriate places in the above code to implement this:

```
// This is the include file for the xor peripheral created
/* define custom peripheral-lab1_xor_xx -the memory mapped interface*/
#include "lab1_xor_xx.h"

// the XOR
int xor_result;
// writes switches to Reg0
LAB1_XOR_XX_mWriteSlaveReg0(XPAR_LAB1_XOR_XX_0_BASEADDR, switches);
// reads result from Reg1
xor_result = LAB1_XOR_XX_mReadSlaveReg1(XPAR_LAB1_XOR_XX_0_BASEADDR);
printf("XOR Result = %d \r\n", xor_result);
```

- However, the code will not successfully work right now because we haven't yet defined the HDL for the new peripheral. Now we need to edit the user logic in Project Navigator, where we will perform the XOR. After writing the switches to Reg0, the XOR of the 2 LSBs is performed and the result is stored in Reg1. This is the value we read.
- To edit the user logic, open the **lab1_xor_xx** project file in Xilinx Project Navigator. It is in your Lab1 folder - ..**\Lab1\pcores\lab1_xor_xx_v1_00_a\devl\projnav.** The file to be edited here is the user logic file. It is also in your Lab1 folder - **…\Lab1\pcores\lab1_xor_xx_v1_00_a\hdl\verilog** and the file is 'user_logic.v'. In Project Navigator, expand the top module, 'lab1_xor_xx' and then open 'USER_LOGIC', the user logic module. **DO NOT** edit any other file or any other section of the code unless specified.
- Browse through as much of the code to see what so much code is actually doing. Then, scroll down to where the User Logic starts (it is line 118 in my code).
- Scroll down to where it says '//implement slave model register(s)'. This is where the registers are written with the values. The case statement `'case(slv_reg_write_select)'` selects which register is assigned the value on the data bus. In the case block, for case '10', reg0 is written whereas for case '01', reg1 is written.
- Since for reg1 to be written with the XOR of the values, we have to actually write something to it so that the case '01' is enabled. Thus, we can now read the value written to reg0 and send this to reg1. We do this because, although not here, some computation could otherwise have been performed on reg0. And hence we want to read the latest value. This is done by including the following lines of code in the .c file:

```
Reg0 = LAB1_XOR_AK_mReadSlaveReg0(XPAR_LAB1_XOR_AK_0_BASEADDR);
printf("Reg0 Value = %X \r\n", Reg0);
LAB1_XOR_AK_mWriteSlaveReg1(XPAR_LAB1_XOR_AK_0_BASEADDR, Reg0);
```

- Now, we edit the case statement for reg1 to perform only an xor on the 2 LSBs of the value sent to it. Comment out the code included for the case 2'b01 and instead insert

the line – `slv_reg1 <= Bus2IP_Data[31] ^ Bus2IP_Data[30];` This line computes the xor of the data coming in. The data sent to reg1 is on the bus 'Bus2IP_Data'. This code now writes the value of the xor to reg1. **Note,** we use bits 31 and 30 because it is in the little endian format, i.e. the LSB is written to the highest address and the MSB is written to the lowest address. The code should look like the following:

```
case ( slv_reg_write_select )
  2'b10 :
    for (byte_index = 0; byte_index <= (C_DWIDTH/8)-1;byte_index=byte_index+1)
      if ( Bus2IP_BE[byte_index] == 1 )
       for ( bit_index = byte_index*8; bit_index <= byte_index*8+7;
             bit_index = bit_index+1 )
           slv_reg0[bit_index] <= Bus2IP_Data[bit_index];
  2'b01 :

   // Edited this line for xor
    slv_reg1 <= {31'b0, Bus2IP_Data[31] ^ Bus2IP_Data[30]};

          default : ;
endcase
```

- After doing this, you can close Project Navigator, and go back to Platform Studio. Your final C code should look like the one included below. Click on **Device Configuration -> Download Bitstream**. This, again, synthesizes, implements and downloads the project onto the board. It takes about 15 minutes to do this.

C code:

```
/* lab1_ak.c 20 Dec 2006 */
/* Memory mapped IO interface
   XOR of the 2 LSB's
*/

/* #include files */
/* define I/O devices */
#include "xparameters.h"
/* define general purpose I/O functions for accessing peripherals */
#include "xgpio.h"
/* define custom peripheral - lab1_xor_ak - the memory mapped interface */
#include "lab1_xor_ak.h"
/* define printf */
#define printf xil_printf       /* A smaller footprint printf */

/* Main */
int main(void)
{
   int switches;
   int xor_result;
   int Reg0;
   XGpio gpio_switches;

   printf("\r\nPreparing to read switches\r\n");

   XGpio_Initialize(&gpio_switches, XPAR_DIPSWS_4BIT_DEVICE_ID);
   XGpio_SetDataDirection(&gpio_switches, 1, 0xFFFFFFFF); // input

   switches = XGpio_DiscreteRead(&gpio_switches, 1);
   printf ("Switches = %d\r\n", switches);
```

```
    LAB1_XOR_AK_mWriteSlaveReg0(XPAR_LAB1_XOR_AK_0_BASEADDR, switches);
    Reg0 = LAB1_XOR_AK_mReadSlaveReg0(XPAR_LAB1_XOR_AK_0_BASEADDR);
    printf("Reg0 Value = %X \r\n", Reg0);
    LAB1_XOR_AK_mWriteSlaveReg1(XPAR_LAB1_XOR_AK_0_BASEADDR, Reg0);
    xor_result = LAB1_XOR_AK_mReadSlaveReg1(XPAR_LAB1_XOR_AK_0_BASEADDR);
    printf("XOR Result = %d \r\n", xor_result);
}
```

- Check the hyper terminal to make sure you get what you expect, e.g. switches = 15 gives xor = 0 and switches = 14 gives xor = 1;

2. The Population Count (PopCount) function returns the number of 1's in an N-bit word. It is sometimes used in applications such as pattern matching for genetic analysis. For example, PopCount(11001010) = 4 and PopCount(1001111). PopCount is rather time-consuming to do in software (think about why). In this lab, you will write a hardware-based PopCount accelerator that performs the function in a single cycle.

- Follow steps similar to the ones above to create a PopCount accelerator. The hardware unit should have two registers. The software writes a word to one of the registers. The value read from the second register is the number of 1's in the first register. Write a C program to write some test values and display the counts in a HyperTerminal window. Choose a set of test values that is as small as possible while sufficient to convince a skeptical viewer that your code is correct.

This lab was developed by Anu Kohli

What To Turn In

1. The new Verilog code that you added to user_logic to do PopCount.
2. The C code that tests the PopCount function. Be sure that your code would convince a skeptical viewer that PopCount functions correctly. Describe your test results.
3. How long did you spend working on this lab? The time will not affect your grade but will help adjust future labs to a reasonable workload.