Lab Conventions

The following conventions are used throughout the labs:

- **Location of additional information or tips.**

- **An important error or step that you should be aware of.**

- **A reference to a chapter in the book.**

Lab Overview

For this lab, we will be expanding your project from the first lab to include interrupts. Interrupts, as you learned in earlier classes, can be used to optimize system performance. If a system doesn’t have to poll a hardware device for its status, it can do other tasks instead.

Prelab

Please make sure you are familiar with sections 3.2.4 and 4.4.1 of the book, as many concepts from this section will be used in the lab.

Read through the lab and any necessary documentation. Look over the given system.c so you have a reference as to what functions will be used. Write down which functions you will be using and a short description of each in your lab notebook. Try to determine ahead of time which values you will be passing to each function, especially the arguments that are defined constants.

Also, several questions are posed in the lab. Try to answer the ones you can before lab and place the answers in your lab notebook. Final answers to the questions can be turned in once the lab is complete, but make sure you have at least some of the questions answered before coming to lab.
I/O and Interrupt system

New functionality is required of your system from Lab 1. Specifically, you need to display the time the program has been running down to the second. Features for which actions need to be very quick or very accurate usually require the use of interrupts.

Think about how you would design a piece of software that displayed the time accurately – perhaps you would use a sleep() function. However, this method is not accurate because it does not account for the time it takes to display the message over the serial port – if your baud rate was slow, it may be significant and noticeable immediately. If you think about it, even if your display rate was very fast, using a simple sleep() function will always result in a time drift. Now think about if you implemented the poor timer with a sleep() function and then were asked to add more time-dependent functionality; the software gets very complicated very quickly. With interrupts, the solution is much simpler and very accurate.

To improve our LED flasher from Lab 1, we will be adding an interrupt controller and several hardware cores. The LEDs will flash at a rate set by a timer. A timer built into the PPC405 CPU core, the Periodic Interrupt Timer (PIT), will be used to display the time accurately. Lastly, we will use an interrupt to handle the push-button input.

In fact, the only code in the main() while loop will be to display characters over the serial port (TeraTerm)! All the actions besides writing to the serial port will be driven by interrupts. A global boolean flag, called terminal_update, is used by the various interrupt handlers to communicate with main(). If a task needs to refresh the display, it sets terminal_update to 1.

In general, interrupt handlers should be as short (in execution time) as possible. Why might writing to the serial port in the interrupt handler be a bad idea?

Part I: Add hardware cores

In order to complete this design, we will first need to talk about how the Xilinx XPS tool handles the addition of hardware to a system. Adding hardware actually implies a hardware component and a software component that your program uses to interface to the hardware component.

In the Xilinx documentation, you will see the word “core” or “pcore”. This refers to a hardware module, usually written in VHDL, that can be added to a system through XPS. Each hardware core also has an associated “driver”, which refers to a library of software functions made specifically to interact with a particular hardware core.

In this lab we will be using three new hardware cores and their associated software drivers. The following is a short description of each type of module:
<table>
<thead>
<tr>
<th>Name</th>
<th>Function Prefix</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gpio</td>
<td>XGpio</td>
<td>Software drivers to interact with a “gpio” hardware module. GPIO stands for General Purpose Input/Output. This type of hardware module is used to access the LEDs, push buttons, and user DIP switches. It could also be used to interface with simple hardware you create. It provides only a simple interface of reading and writing, along with optionally providing an interrupt signal.</td>
</tr>
<tr>
<td>Intc</td>
<td>XIntc</td>
<td>Software drivers to interact with a “intc” hardware module. Intc is an abbreviation of Interrupt Controller. The interrupt controller hardware module consists of up to 32 interrupt inputs and a single interrupt output. Interrupts can be configured as rising/falling edge or high/low level triggered. Read more about interrupts in the book in section 3.2.4.</td>
</tr>
<tr>
<td>Tmrcr</td>
<td>XTmrCtr</td>
<td>Software drivers to interact with a “tmrcr” hardware module. TmrCtr is an abbreviation for Timer/Counter. This hardware module is a simple counter which can count up or down and generate an interrupt when the count reaches user-defined values. The uses of timers are briefly discussed in the book in section 4.4.1.</td>
</tr>
</tbody>
</table>

The first step in adding interrupts to your project is to add an external interrupt controller.

To understand why an external interrupt controller is necessary, we must look into the architecture of the PowerPC 405 core within the Virtex-II Pro FPGA.
The PowerPC 405 block has two external interrupt signals: one is called the external or non-critical interrupt, and the other is called the critical interrupt. A critical interrupt can interrupt a non-critical interrupt; however a non-critical interrupt cannot interrupt a critical interrupt. This allows for a 2-level hierarchy of interrupts. We will only be using the non-critical interrupt, but some real-time or deterministic systems may need to use this 2-level scheme to achieve strict timing requirements for interrupt latency.

As shown in the picture above, it is very possible you will want to connect multiple HW resources to the PowerPC core interrupt input. However since we are restricting ourselves to the non-critical input and there is only one input, we need some external hardware. The INTC core allows up to 32 devices to connect their interrupt signals to it, and it generates a master interrupt signal to send to the PowerPC. The OPB (On-Chip Peripheral Bus) interface of the INTC is used only to enable/disable and otherwise configure the INTC core. All of the interrupt signals do not pass through the bus and are connected directly to the cores.

Open your project in Xilinx Platform Studio.

First, we need to add an INTC.

Go to Project -> Add/Edit Cores …

This brings up a new window. Scroll down the list on the right to find opb_intc, and add it to the Peripherals.
After adding the opb_intc, it should show up as a peripheral in the list to the left at the bottom.

Now that opb_intc is added, click on the “Bus Connections” tab and add the opb_intc_0 sopb line to the OPB bus. The “opb_intc_0 sopb” line means that the intc device is a slave on the OPB bus. By default cores are not connected to a bus, so you must connect them by clicking inside the appropriate bus. An “S” should appear, indicating that a slave connection has been made.
The PowerPC accesses all devices through memory-mapped I/O. Click on the “Addresses” tab to view the current memory map for the system.

Where is the SDRAM located in memory (this is where our program will be stored)?
Where is the BRAM (memory internal to the FPGA)?

To update the memory map so the opb_intc core is included, click on the “Addresses” tab and click on “Generate Addresses” on the bottom right corner. This will automatically generate a valid memory map for your system, although if you needed to you could still edit it manually.
After generating the address space, click on the “Ports” tab. Here we will add specific ports for the opb_intc core, and we will also need to add the interrupt line for the Push Buttons. Refer to the architectural diagram at the beginning of the lab if you’d like to follow along as we connect the interrupts to the processor.

Let’s add the opb_intc core ports. To do this, click on the drop box on the upper right, labeled as “Ports Filter:”, and find opb_intc_0. This is instance “0” of the INTC, since you might want to have multiple INTC cores in your design. When the “List of Ports” text box updates add all three ports listed under opb_intc_0 to the Internal Ports Connections using the “<< Add” button.

The three ports on the INTC core are:

<table>
<thead>
<tr>
<th>Port</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPB_Clk</td>
<td>The main clock signal the core uses for timing and communication over the bus</td>
</tr>
<tr>
<td>Intr</td>
<td>A 32-bit wide input to the INTC core, where each bit is an interrupt from a core</td>
</tr>
<tr>
<td>_irq</td>
<td>The master IRQ (Interrupt ReQuest) that is connected to the EICC405EXTINPUTIRQ input on the PowerPC core</td>
</tr>
</tbody>
</table>

![Add/Edit Hardware Platform Specifications](image)
Now that our INTC is added to the project, we need to connect the Push Button Interrupt request line to the INTC.

On the “Ports Filter:” drop box find **PushButton_5bit**. Now on the “List of Ports” text box find the interrupt port for the Push Buttons, it is label **IP2INTC_Irpt**, add this to the Internal Ports Connections. The line will cause an interrupt when any push button changes its value.

To connect this interrupt signal to the INTC core, click on the “… “ button next to the “**Intr**” Port in the “**Internal Ports Connections:**” listing (it should be listed under the “opb_intc_0” instance. A special window will show up for configuring interrupts, allowing you to connect available interrupt lines to the INTC. Add the Push Button interrupt line (it should be the only interrupt listed) as connected. Click OK.

Now the INTC and the input to it are configured, we just need to connect the output of the INTC to the PowerPC non-critical (external) interrupt input.
In the “Ports Filter:” list find the `ppc405_0` and find the `EICC405EXTINPUTIRQ` port and add that to the “Internal Ports Connections:”.

To connect `EICC405EXTINPUTIRQ` port and the `Irq` port together, highlight them both (using CTRL) and click the “Connect” button. A dialog box will pop up wanting you to name the new wire connecting the two – give it a descriptive name.

Connections cannot be named using spaces or `->`. Naming connections with either of these will cause syntax errors when building the underlying VHDL code for the hardware, and may also cause Xilinx Platform Studio to crash.
Next, by default the **OPB_Clk** input on the **INTC core** is connected to a clock signal that doesn’t exist – we need to change it to match our system. The system clock used for our OPB is called “**sys_clk_s**” (you can verify this by finding another device on the OPB and looking at the signal connected to its **OPB_Clk port**).

Change the INTC core’s **OPB_Clk port** to be connected to the **OPB_Clk signal**.
The push button core also does not have the hardware capability to produce interrupts by default. If an FPGA design is constrained by space, you might want to leave certain features of a core out of the design – many cores work this way, and are configurable. We need to tell XPS to produce push button hardware which has the capability of generating interrupts. Go to the “Parameters” tab, and then choose the “PushButtons_5bit” GPIO device under “Choose IP Instance”. The dialog shows the default values on the right side, and the values which have been overridden on the left side. We need to override the “C_INTERRUPT_PRESENT” from 0 to a 1.
Add the parameter and set it to 1 in the left-hand window. Then click **OK** to apply the hardware changes.

The documentation for the core is available via the “View PDF Doc” button if you would like to know more about the parameters of the core.

We are now finished connecting the push buttons to the interrupt timer. Using the previous process, do the following:

- Add an **opb_timer** core to the system and connect it to the bus. Change the instance name to “**opb_timer_leds**” to avoid confusing it with other timers.
- We will only be using the **interrupt output port** of the **opb_timer**, but the **OBP_Clk** also needs to be connected to give the timer a source clock.
- Connect the **Interrupt output port** of the timer to the **INTC** device.

After you are finished with the hardware changes select the “**System**” tab in XPS. Scroll down and expand “**Project Files**” and open the Microprocessor Hardware Specification (MHS) file. The MHS file is generated in Base System Builder by the “**Add/Edit Cores…**” GUI. It lists, gives parameters to, and connects the different hardware components in the system. The MHS file also defines external ports.

**Write down the lines that connect the pushbuttons and the led timer interrupt to the interrupt controller, and the interrupt controller to the processor.** Also write down the baud rate of the serial connection for TeraTerm. Put these in your Lab Notebook.

Another noteworthy file generated in Base System Builder is the **Universal Constraints File (UCF)**. This file connects all the external ports to actual pins on the FPGA. It will add constraints to certain pins and settings of the system. In other words, all pin assignments are automatically chosen unless the pin is listed in the UCF, then it is given its assigned value. If you remember from LAB01 you added two lines to the UCF. These two lines provide a necessary phase shift in order to interface with the DDR RAM on the Digilent XUP board. Since you copied LAB01 to use for this lab those lines should already be in the UCF. **Remember that you will always need to have these lines in the UCF.** In the case that you don’t have these two lines added, they are provided below:

```
INST "dcm_1/dcm_1/DCM_INST" CLKOUT_PHASE_SHIFT = "FIXED";
INST "dcm_1/dcm_1/DCM_INST" PHASE_SHIFT = "80";
```

When you make hardware changes you will want to be sure that all of the addresses for your system do not overlap. As a final step, go to **Add/Edit Cores…** and **Generate Addresses…** one more time.
Make sure you re-synthesize your hardware before continuing, and download the new hardware to your board. You will also need to generate new libraries for the new hardware (Tools -> Generate Libraries). If you do not do one or both of these, you will not be able to proceed with adding functionality to the system.

The architecture for our system is now complete. Now we must write the software code to run our system. A sample system.c is provided with the basic structure of the code you will need.

To give the system.c code enough room to call functions, we need to increase the stack and heap space. Go to **Tools -> Generate Linker Script...**

This brings up a view of the different portions of your code, and which memory the sections of code are assigned to. Make sure that the `.boot` and `.boot0` sections are assigned to the BRAM on the FPGA – this is a requirement for the PowerPC. Then, change the **stack and heap size** (in the upper-right corner) to be **0x8000**. Click “Generate” to complete the process.

The basic structure of system.c is:

Implement the system so the time is displayed and updated every second, and the flashing delay is displayed and updated when changed on the HyperTerminal. The LEDs should flash at the current “delay” rate (in ms), and all actions should not affect any other action...
occurring on the board. The portions of the code which you need to update are marked with a “TODO:” comment.

Remember: If you need information on a particular software function, documentation is located in the help files. “Help -> EDK Online Documentation -> Documents -> Driver Reference Guide -> API Links”

Draw a UML sequence diagram of our system. See the examples shown on page 118 or page 123 of the textbook. It should show a typical user interaction with the system. Turn this diagram in with your lab notebook.

Demonstrate the complete system to your TA. Print off a copy of your source code and turn it in with your lab notebook.