**DRAFT of description for ELEC 371 Lab 2 preparation**

Review course material in Chapter 7 (Input/Output Organization) on parallel port concepts.

Review course material in Chapter 3 (Basic Input/Output – Focus in Interrupts) that discusses the relevant concepts such as main program vs. interrupt-service routine, enabling/disabling of interrupts, special instructions to access the processor control registers, and guidelines for software/hardware initialization code in the main program.

The course material uses generic-RISC control register names PS, IPS, IENABLE, and IPENDING. For the Nios II, the corresponding names are `status`, `estatus`, `ienable`, and `ipending`. The course material uses a generic-RISC *MoveControl* instruction. The Nios II actually has two distinct special instructions: `rdctl` and `wrctl`. In slide 20 of the Chapter 3 interrupt-related material, the *MoveControl R4, IPENDING* instruction would be written as `rdctl r4, ipending` for Nios II. Similarly, *MoveControl IENABLE, R3* would be written as `wrctl ienable, r3` for Nios II.

Review the documentation on the vendor-provided computer system for the DE0 board, focusing on Figure 1 on page 2, Section 2.3, the beginning of Section 3 at the top of page 15 (the table identifies the `ienable`/`ipending` bit position for each interrupt source), and Sections 3.1 and 3.1.1.

In the *template.s* assembly-language source file for Lab 2, study the organization of the code and read the explanatory comments. Relate the code organization to the course material for Chapter 3.

Your individual preparation of code for Lab 2 involves copying *template.s* to a new *lab2.s* file, then completing the unfinished portions of *lab2.s* to produce a basic interrupt-based program that responds to pressing and *releasing* pushbutton 1. Further guidance for this preparation is provided in the remainder of this discussion.

But as further individual preparation that is straightforward and built on the foundation of previous coursework, incorporate the `PrintChar(ch)` subroutine in the *lab2.s* file. Then, introduce a `PrintString(str_ptr)` subroutine that calls `PrintChar()` to print a sequence of zero-terminated characters that define a string in memory. Define a string in memory using the `.asciz` directive with the text `ELEC 371 Lab 2 by` *<group member names>* `\n` (newline at end) and have the main routine make a call to `PrintString()` at the beginning of execution with a pointer to this string. Doing so will display a useful heading in the Terminal window.

In-lab activity will involve using specifications to generate additional code that extends the basic interrupt program above with more computation and/or input/output activity involving parallel ports and character printing.

Use the CPUlator simulation tool to test the prepared *lab2.s* code before your scheduled lab session. In the simulator, it is necessary to use *two* left-mouse-button clicks on the pushbutton 1 checkbox to model the physical action of first pressing (and holding) the button, and then releasing the button.

Create a LAB2 folder on your personal computer; if working on a lab computer use a USB drive or your network storage. Place the template assembly-language file in your LAB2 folder. Open a plain-text editor and view the contents of the template file. Read the comments and understand the appropriate structure of the portions of the code that have been left incomplete.

Use the vendor documentation for the DE0 Computer to create `.equ` symbol definitions for the button interface and for the green LEDs (other address may later be similarly defined in a symbolic manner). Also review the technical information in the vendor documentation for the button interface registers.

The detailed specifications for the button-interrupt program to be prepared by you before your lab session are summarized below.

- There should be a global variable `COUNT` in the data portion of the code.
- The main (infinite) loop of the program should simply increment `COUNT` in memory.
- In the `Init` subroutine, set up the button interface for interrupts on button 1. For this purpose, write an appropriate bit pattern to the mask register so that interrupts are generated only for button 1.
- In the interrupt service routine, *include code that reflects the general form of checking multiple sources of interrupts*, even though for this part there is only one source. The course material in Chapter 3 shows the desired approach in generic RISC format: retrieve the `ipending` contents and place the bit pattern in a register (just *one* read of `ipending`), perform an AND operation with an immediate value that has the bit set in the position assigned to the specific interrupt source with the AND result placed in a *different* register, and use a branch to compare the AND result with all-zero in r0 and if the AND result is zero (i.e., bit is 0) then take the branch to the next interrupt-source check. If the branch is not taken (i.e., bit is 1), then execution falls through to the code for responding to the interrupt. That interrupt-response code could involve calling a dedicated subroutine to do the necessary work, but for the *lab2.s* prepared file the relevant code sequence should just be inserted in-line immediately after the branch that skips to the next check. See slide 25 of the Chapter 3 material.
- In the response to a button interrupt (and in fact *any* interrupt), *remember to clear the interrupt source*, otherwise the interrupt service routine will be re-entered continuously and no progress will be made in the main program. For the button interface, there is an inconsistency between the official specification in the vendor documentation and the way that the author of the CPUlator tool has modelled the button interface. The vendor documentation states that writing *anything* to the edge register of the button interface clears all pending button interrupts. The CPUlator tool, however, reflects a different approach that clears one or more individual button interrupts by requiring a write to the edge register with a bit pattern having one or more bits set to 1 corresponding to the appropriate button(s). *Use the more specific approach that applies to CPUlator*; the correct behavior will then be obtained on both the simulator and the hardware.
- The response to a button interrupt in this program is simple: toggle the on/off state of bit 0 of the LED port data register. In digital logic, XORing a value/signal *w* with 1 produces the complement of *w*. For the software in this case, first read the current LED data register value, use an `xori` instruction with an immediate value of 1 to toggle the desired bit, then write the toggled value back to the LED data register.

After the contents of the template assembly-language file have been completed according to the above specifications, open the Web-based CPUlator tool and select the DE0 system (not the generic system) as the target to have the needed input/output interfaces (parallel ports and Terminal window for printing).

Use the *File→Open* menu option to read your *lab2.s* source code into the simulator. Use *Compile and Load* to assemble the code.

If there are errors, use your text editor on your local copy of the *lab2.s* file to make changes, save the file, use *File→Open* again in the Web-based simulator, and *Compile and Load* again.

If assembly is successful without errors, click on *Continue* to simulate the execution.

A correct program should result in the right-most simulated green LED in the Web-based simulator turning on or turning off (i.e., toggling) each time button 1 is pressed *and released*. As stated previously, modelling the press and release actions requires two left-mouse-button clicks on the button 1 checkbox in CPUlator. The change to the LED occurs on each interrupt is caused by the *release* of button 1. When the processor recognizes a button interrupt, it temporarily stops executing code in the main program and instead executes the code in the interrupt service routine. After completing the interrupt service, execution returns to the main program.

A correct *lab2.s* program should also be incrementing the COUNT variable in memory as part of the main program execution. Use the Memory view in the Web-based simulator and enter the hexadecimal address 1000 (presuming that hex 1000 is the `.org` address specified before the data directive that reserves space for the COUNT variable). Not only should the contents at memory location 1000 be changing as the execution of the program is simulated, but the value in the appropriate general-purpose register used for the increment operation within the main loop should also be changing in the register window on the left side of the simulator window.

The contents in the register window can be scrolled down to view the other processor registers, including `ea`, `status`, `estatus`, `ienable`, and `ipending`.

Finally, use the breakpoint feature of the Web-based simulator to cause simulated execution to halt at the beginning of the interrupt service routine (either at the branch at address 0x20 or at the actual start of the routine at the `isr:` label). Single-step through the interrupt service routine code to better appreciate how the software operates and the effects of the software execution on the hardware (specifically `ipending`).