CMPE 12L Lab 5 - Fall 2014
PIC32 Function Calls and IO

Prof. Matthew Guthaus
Due: November 26, 2014 5pm
75 Points (20 Report, 55 Work)

1 Background

Interfacing with the outside world requires that the processor understand time in human terms. However, instructions run much faster than we can see. In fact, the clock frequency very roughly specifies the number of instructions that run per second. Our processor has an 80 MHz clock speed, there are 80,000,000 cycles per second. This is an approximation, however, because some instructions take more than one cycle to run if, for example, they access a slow memory.

In this assignment, we are going to write a function in assembler to read the state of some input switches and control the rate of the LEDs blinking.

2 Prerequisites

This lab has a lot of reading, because this is also the reading for lecture. The PIC32 material is not in the textbook so you do not have any textbook readings for the next couple weeks.

- Read through this **entire** lab assignment.
- Review the lecture notes on the PIC32 architecture.
- Read section 4.0 of PIC32MX320F128.pdf in eCommons
- Read section 12.0 of PIC32MX320F128.pdf in eCommons
- Look at section 27.0 of PIC32MX320F128.pdf in eCommons
- Sections 1.0-3.0 of PIC32MX320F128.pdf are also interesting, but not required reading.

3 Tutor/TA Review

Your lab tutor/TA will cover the following items in the first portion of the first lab:

- How to use the MPLAB X
- How to use the Arduino, IO module, and debugger/programmer.
- What's required

4 Assignment

This assignment has 4 parts. You should finish first part 1 (and possibly part 2) quite quickly, but the other three parts may take significant time. Part 3 requires writing your first MIPS assembly function. Part 4 requires learning about the PIC32/Uno32 hardware to receive input from the switches.

4.1 Part 1: Tutorial

Read through and experiment with the options in the **entire** tutorial. Run the demo.c program and answer some of the initial questions at the end. You do not need to show this to your TA or tutor, but it is required to do the other parts.

4.2 Part 2: Debugging

In this section, you are to trace the execution stack of the printf statement in the demo program. Specifically, what addresses and labels does it jump to in order to print the output. Trace through the execution and observe what happens. Note that these are in a C library routine which you can see in the Window→PIC Memory Views→Execution Memory. By looking at the Window→PIC Memory Views→CPU Registers, you can see what addresses and return addresses are used in the jal and jr instructions.

How many levels deep is the call? What functions/labels and at what addresses? Can you figure out how many and what arguments are passed to call?

4.3 Part 3: Subroutines in MIPS

In this section, you are to write your own function that will return a constant 0x80000 for the delay loop in the demo.c program and print out the number of times that it has been called to the serial output. The output should look exactly like:

```
Hello, world! 0
Hello, world! 1
Hello, world! 2
...
```

You may miss a few of the initial ones by the time you attach the terminal program. That is ok!

In order to successfully do this, you need to properly implement the caller and callee save register schemes on the stack. **Do not save registers to local memory. Use the stack.**

What registers did you save and why? Where do you store the number of times that the function has been called?

4.4 Part 4: I/O in MIPS

In this section, you should "tune" the delay values using the input switches. First, read the state of all four switches and **encode the boolean values from the switches into the lower 4-bits of a single word.** This can be stored in a register or memory if you like.

Now that we have the switch value, we want to cycle through all 8 LEDs at approximately the following rates:

SW4	SW3	SW2	SW1	Description
0	0	0	0	0 seconds (constantly moving, no delay)
0	0	0	1	1 second
				...
1	0	0	0	8 seconds (one LED moving every 1 second)
				...
1	1	1	1	15 seconds (one LED moving roughly every 2 seconds)

These don't have to be exact, but they should be close enough to convince your TA/tutor. **The actual values should be interpolated (or extrapolated) using arithmetic operations (e.g. ADD, MULT, MULTSUB and DIV).** You can create your own multiple and divide if you want, but it is probably easier to use the hardware instructions. Remember, interpolation/extrapolation is simply using two values to create a linear fit and estimating the other points based on this line.

Rather than estimate what the return values should be to get an appropriate delay, can you compute two points (say 1 and 15) and estimate using the 80MHz CPU clock? What is the delay per iteration of the for loop in the C program? How do you implement the correct return values?

5 Input/Output

You are given C code that light the LEDs. In addition to looking at the C, you can look at the disassembled output to see the exact instructions to do this. This is a useful place to start for implementing your device input from the switches. For part 4, you need to read the values from the switches during each call to your getDelay function.

Input/output in our microcontroller is based on device registers mapped to certain addresses. Specifically, the switches have different names on the Uno32 board, PIC32 package, and in the CPU Ports. The following is from the Basic I/O board and Uno32 reference manual:

Name	Uno32 Pin	PIC32 Pin	Port/Bit
SW1	2	42	RD8
SW2	7	43	RD9
SW3	8	44	RD10
SW4	35	45	RD11

This means these are all on Port D. As an example, the port D address starts at 0xBF88_60C0 and is shown in Table 4-26 of PIC32MX320F128.pdf. There are 4 registers that let us configure the input/output: TRISD (0xBF88_60C0), PORTD (0xBF88_60D0), LATD (0xBF88_60E0), and ODCD (0xBF88_60F0) which are described in Section 12 of the PIC32 reference manual. The document describes all the ports as TRISx where x can be A to G for each of the IO ports (A to G) at different addresses.

Each of the above registers actually has four addresses that perform different functions. The base register can be read or written to change all of the bits at once. Three additional registers are used to modify one bit at a time: clear (offset 4), set (offset 8), and invert (offset 12). The ones in a mask specify which bits to clear, set or invert. So, for example, if you want to set bit 0 of the TRISD register, you can write a binary mask of 0x1 to address 0xBF88_60C8 (0xBF88_60C0 + 8).

The TRISD register controls whether a pin is an input or output. Setting a bit to a 0 (active low) means that it will be an output. Setting it to a 1 makes it an input and is the default after power on.

The PORTD register is where you can read or write data. Similarly, LATF can read or write data but this is synchronized to the system clock in a latch. Writing the LATF as an output will hold the value after the current clock cycle. Is there a difference between writing LATF and PORTF?

Last, ODCD controls whether the output is a CMOS logic voltage or is an "open drain" connection. You will learn more about this in other classes, but we don't need it here.

The full schematic of the I/O port is shown in Figure 12-1 of the PIC32 manual.

6 Format

Your program must be two text files: the demo.c file with the function call to getDelay and the assembly code that assembles with no errors and implements getDelay.

7 Lab Submission

Your lab will be submitted via your eCommons account. Please log in to eCommons using your UCSC account and attach the following files to your "Lab 5" assignment submission:

- demo.c which contains the function call (slightly modified to do this)
- lab5.s which contains the subroutine call with variable delay and the printf of each call number
- lab5_report.pdf

Note that the final report must be submitted in PDF format.

Make sure to confirm that your assignment is SAVED and SUBMITTED before the deadline. You may resubmit your assignment an unlimited number of times up until the due date.

7.1 Check-off

For this lab, as with most labs, you will need to demonstrate your lab when it is finished to the TA or tutor and get it signed off. You will also need to submit your lab files using eCommons. Any demos after the submission deadline require the files to be downloaded from eCommons.

7.2 Grading template

This is a suggested grading rubric. It is also a good general guideline before submitting your lab to check off these points.

7.2.1 Requirements

☐ (25 pts) Part 3

- Does the program return from the function properly to the main C program?
- Did you properly save/restore caller/callee save variables to the stack?
- Does your function properly print the text?
- Does your function properly print the current iteration count?
- Does your delay return value work so that the LED delay is the same as the original demo.c?

☐ (30 pts) Part 4

- Does a delay of 0 work?
- Does a delay of 8 work (1 second per LED)?
- Does a delay of 15 work (just under 2 seconds per LED)?
- Does the program correctly interpolate/extrapolate?

7.2.2 Lab write-up requirements

In the lab write-up, we will be looking for the following things. The lab report is worth 20 points. We do not break down the point values; instead, we will assess the lab report as a whole while looking for the following content in the report.

Along with the usual items, you should answer the following questions:

- Is "li $t0, 0xBF886110" an instruction or a pseudo-operation in the assembler? Explain.
- What instructions does the "mask = mask << 1;" get compiled into?
- Which LEDs are on before executing "PORTECLR = 0x00FF"? Why do you think this is?
- In what memory ranges does the program store your data such as the mask in the demo program? What type of memory is this?
- In what memory ranges does the program store your instructions? What type of memory is this?
- How much memory do the instructions in your final program occupy? How much data does it use?
- Does the program or data size change when it is in debug mode? Why do you think this is?
- How do you read from an input switch?
- In part 2: How many levels deep do the calls go? What functions? What addresses? What are the arguments at each level?

- In part 3: What registers did you save and why? Where do you store the number of times that the function has been called?
- In part 4: How did you compute your delay numbers and what are they? What does a delay of 0 look like?

8 Supplementary Reading for the Curious

- `http://en.wikibooks.org/wiki/MIPS_Assembly`
- `http://en.wikipedia.org/wiki/Executable_and_Linkable_Format`
- `http://en.wikipedia.org/wiki/Name_mangling`
- `http://tigcc.ticalc.org/doc/gnuasm.html`