CMPE 12 Lab Report # 5

John Allard
Steve Lessard
Lab Section #2

November 28th, 2014

1 Overview

For laboratory #5, we were officially finished with the 'little-computer-3' simulator, and moving on to programming with actual microcontrollers and more advanced instruction set architectures. This lab had us using the Microchip Microcontrollers with a 32-bit word length, along with an IO shield that has some built in peropheral devices for us to use. This microcontroller is based on the PIC32 architecture, a common 32-bit architecture, and uses the MIPS instruction set architecture. MIPS is a reduced instruction set architecture, which means it is one of the more simple (less commands to memorize) ISA's to use (as opposed to CISC ISA's). The purpose of this lab was multidimensional, and included the following topics : MIPS subroutines, input using switches on the IO shield, and writing output to both the console and to a set of 8 LED's on the top of the IO shield. By the end of the lab, all of these topics will be combined into a microcontroller that changes the frequency at which the 8 LED's light up based on the numeric input of the switches. A more in-depth discussion can be found in the following sections.

2 getDelay Subroutine

Designing and implementing this subroutine was the core of this lab assignment. The `demo.c` file contains a small `main` function that serves as the starting poiint for the program. Inside this main function is a infinite loop for the microcontroller to execute. The entire purpose of this program is to light up a series of 8 LED's in sequence, with some timing interval inbetween the transistion from LED_n being lit to LED_{n+1} being lit. The timing interval should be determined as follows :

- The user configures the 4 switches on top of the IO shield to represent a number in between 0 and 15 (in binary).
- When the `getDelay` function is called, we read in the numeric value represented by the switches.
- Based on the number read in, we adjust the timing between consecutive LED's being lit so that if the number is n, it will n seconds for all 8 LED's to be lit and for the cycle to restart. So if $n = 8$, which would mean the 4 switches are in the positions $\{1000\}$, then each LED would be lit for about a second, allowing all 8 to be lit in an 8 second time span.

The timing did not have to be perfect, it was supposed to be interpolated from two chosen reference points. I found that by when reading in a binary value of 8 from the switches, then shifting this value left 12 times, it was very close to being a 1-second interval for each LED being lit. This meant that it would take very close to 8 seconds for the whole cycle to complete, which was exactly the results I was looking for. I extended this test to a value read from the switches of 4, and each LED was lit for roughly a half-second, which was again the behavior I was looking for. So right now, all I do to determine the delay is the following :

`LW $v0, 0xBF8860D0`	Load the PORTD word
`SLL $v0, $v0, 12`	Shift the value left by 12 bits

This almost seemed too simple at first, but the microcontroller handles the majority of the work for us. Each bit of the PORTD word (found at address `0XBF8860D0`) represents the state of the corresponding of an input switch on the IO shield. So if the switches are configured as such $\{1011\}$, then the value at PORTD will be 11. I can then shift the value left 12 bits, and return it from my function. Below are the answers to some questions posed to the students in the lab instruction handout.

- Is `li $t0, 0xBF886110` an instruction or a pseudo-operation in the assembler? Explain. This command is actually a psuedooperation for the assembler. This is because there is no `li` command in the MIPS instruction set. The assembler takes this command and translates it into a sequence of actual instructions that can be translated directly into machine code.

- What instructions does the `mask = mask << 1;` get compiled into? This instruction bit shift the mask variable by one to the left. So this would get compiled into the following commands :

```
LW $v0, 16($s8)
SLL $v0, $v0, 1
SW $v0, 16($s8)
```

- Which LEDs are on before executing `PORTECLR = 0x00FF`? Why do you think this is? I would think that this register would hold whatever junk values were stored in it during the last use of the microcontroller. My reasoning for this is that when I tried to run the program without running the command `PORTECLR = 0x00FF`, it seemed that the only ones lit were the last ones to be used.

- In what memory ranges does the program store your data such as the mask in the demo program? What type of memory is this? When I viewed the dissasembled output, it looks like the mask variable and others like it are stored on top of the frame pointer. For example, the mask variable is instantiated and saved with the command `SW $v0, 16($s8)`, which saves it 16 words ontop of the current position of the frame pointer (referred to as $s8 here). This type of memory is known as Data Memory.

- In what memory ranges does the program store your instructions? What type of memory is this? Looking at the PIC32 reference manual, it says that all user program memory is stored in the locations from `0xBF000000 + BMXDUPBA` to `0xBF000000 + BMXDRMSZ(3)-1` This memory is known as Program Memory, and it is the sequence of instructions traversed by the program counter.

- How much memory do the instructions in your final program occupy? How much data does it use? My final project consists of 120 lines of assembly (including both the demo.c file and the getDelay subroutine.) So my program contains 120 words of memory with each word consisting of 32 bits. My program does not use much auxilery data, going through my final dissassembled output I was able to count 14 times when we saved a word on the frame pointer, so we use at most 14 words of memory for the the data directly used by my final program.

- Does the program or data size change when it is in debug mode? Why do you think this is? I do think that the size of the program changes, and this is because in debugging mode more code is inserted to perform checks and analysis on the running code.

- How do you read from an input switch? To read from one of the 4 input switches, you need to read in the value from the PORTD register, and use a mask on this value to extract the bit for the switch in question. If you want to find the value of the first switch, you could AND the value from the PORTD register with 0x1. If the resulting value is 0x1, then the first bit is high which means the first switch is high. There might be better ways of accomplishing this but this worked for me.

- In part 2: How many levels deep do the calls go? What functions? What addresses? What are the arguments at each level?

3 Conclusion

This introduction to both the PIC32 platform and the MIPS ISA was a pleasant experience compared to the LC3, which seems quite primitive in comparison. It was also helpful to be able to program for the actual computer instead of a simulator. This lab showed the importance of putting a lot of thought into sub-routine design, and how bad practice can lead to corrupted data and unexpected results for the caller of the subroutine. On top of these things, this lab showed how C code, which most people in this class are familiar with, gets translated (compiled) down to assembly code. In the future, I will have an easier time debugging C programs because I now have a better understand of how they work 'under the hood'. Finally, this lab gave me a greater appreciation for the higher-level languages and how much they simplify our lives as programmers.