Matthew James Greci

Banner ID: 101634187

Laboratory 1:

Introduction to MX4ck Trainer System & Simple Programming

February 3, 2015

**Introduction**

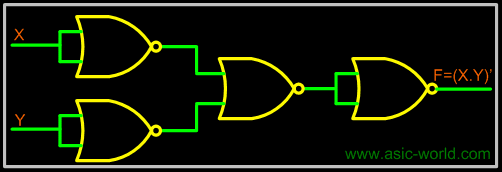
The goal of this laboratory is to learn basic commands and their syntax of the MX4ck PIC32MX460F512L microprocessor board. This involves learning how to access registers, memory locations, create assembler instructions, use the MPLAB IDE and use the breakpoint feature. In order to complete this course, knowledge of the microprocessor is required. However, the most important goal is to gain a deeper knowledge of assembly level code and how code and text is translated to assembly, executable files, and ultimately, zeros and ones.

**Methodology**

The first step to solving this laboratory was to read the given laboratory assignment. Particularly, read over the Objective and Activities sections to understand what was expected and then start reading through problem 1. The instructions are very straight forward so the only other information needed is the syntax for running the load word, load half word, and load byte commands. Using information about syntax and how to use the MPLAB IDE discussed in the classroom in combination with the MIPS Architecture document, this became a very simple problem. Using the “li” command, 32 bit values in hexadecimal can be loaded into a register. For example, “li $t0, 0x12345678”. This command actually translates to two commands since the immediate value can only be 16 values at a time. “lui $t0, 0x1234” and “ori $t0, 0x5678”

Problem 2 was slightly difficult as the “.long” was described well in the documentation. However, the TA was able to explain further. One can use labels to indicate a place in memory, and let the complier determine the exact address. The user can then reference this address by the label. The “.long” can be used to place a value at the label’s location. Then the user can access it by using the lw, lh, and lb commands. This command requires a base (the s0 register points to data) and an offset to access each value, 4 bytes at a time. The lh takes the 2 least significant bytes and loads them into the registers and then sign extends (looks at the most significant bits and adds that bits to the upper 4 bytes). The lb command does the same except loads the least significant byte. In order to show this program works, it is important to clear the registers between the lw, lh, and lb commands and to use breakpoints.

Problem 3 puts one’s knowledge of logical operators into the program. The ADD, SUB, and AND commands are straightforward and require the user to know the first register is the destination and the second and third registers are the source registers. The NAND operation is a bit trickier as there are no NAND or NOT command in the MIPS Architecture. Instead, NOR gates can be used to create an equivalent NAND operation. This is shown in the image from asic-world.com below. Using X as register t0 and Y as t1, NAND can be represented and computed by the equation: !(!(!(X+X)+!(Y+Y)) +!(!(X+X)+!(Y+Y))). This can be confirmed by looking at the values placed into registers.



Problem 4 is a simple loop to increment the value 0x1101 by 7 and store into memory 100 times. The idea of looping is to use a label and a conditional branch statement to determine to continue looping or exit the loop. Also, before starting the loop, the values to determine the conditional branch need to be initialized. It is also important to choose the loop counting variable based on the order of operations inside the loop. Storing the value then incrementing the value to store results in different values then incrementing the value then storing the value. Also the conditional branch and value of the loop counter can change how many loops are completed. It made the most sense to always use the “bgtz” statement so that the loop counter is the number loops that will happen. Finally, instead of increasing the offset of the address to store into, the user can increase the base and use an offset of zero.

Problem 5 involves computing the dot product of two vectors. A vector is defined as an array of values. The dot product is the sum of the product of the corresponding values from each vector. For example:

A = {0, 1, 2, 3}, B={4, 5, 6, 7}, Dot product = 0\*4 + 1\*5 + 2\*6 + 3\*7 = 38

The first step is to generate each vector. This is a loop similar to problem 4 that puts 80 elements starting at 54 (Vector A) and 157 (Vector B) into memory (labeled VECA and VECB) that differ by 13 (Vector A) and 18 (Vector B). Once these vectors are setup, another loop is used to compute the dot product. This requires the addresses of VECA and VEB, a loop counter, and a register to hold the total summation, register t2. One value from each vector is loaded, multiplied, and added to the summation register t2. Then the address of the next values is updated, the loop counter is decremented and a conditional branch checks if the loop counter is greater than zero. Once this conditional branch is false, the value in t2 is stored in memory.

**Source Code**

(See attached pages)

**Testing**

Problems 1 and 2 did not require testing as the values given in the laboratory assignment matched the expected outputs (as verified by the TA).

Problem 3 was tested using the values given in the laboratory assignment to verify the operations ADD, SUB, AND, and NAND generated the correct values. This is confirmed by looking at the register values and knowing which registers represent each operation. The TA confirmed this.

Problem 4 was tested by using an initial value of 0 instead of 0x1101. The loop was ran 100 times, added 7 to the value, and stored it in memory. The expected final result would be 700 in decimal and the final value was 0x2BC, which is 700 in decimal.

Problem 5 was tested by using much smaller values and fewer loops. Creating Vector A = {0, 1, 2, 3} and Vector B={4, 5, 6, 7}, the dot product produced was 38 = 0\*4 + 1\*5 + 2\*6 + 3\*7 = 38 (0x26).

**Conclusion**

The main objective was to learn how to create assembly code for the PIC32MX460F512L board, use the MPLAB IDE, and translate pseudo code and algorithms into working instructions for the complier. After much work to figure out how to solve each problem and then translate into working code, this objective was accomplished. While the problems may have not been incredibly difficult to solve initially, the main focus was the translation to assembly code and understanding how to send the complied code to the board to process.

**What I Learned**

This was the perfect amount of work for the first laboratory. I spent just enough time remembering my assembly language coding and converting my code to work for assembly. What really did surprise me was because I did not have access to the board at home and because the results were a little slow to obtain, it actually forced me to think longer and harder about what I was programming. I did not want to waste 30 seconds each time I complied just find out I did not know where the answer was supposed to be. I spent more time thinking about how to solve the problem and how to properly convert it into assembly code rather than debugging my code constantly.

This is something I will consider when programming in higher-level languages. Instead of just thinking of something and not considering what I am doing, I will think more about what I want to accomplish and if this change will help me or just waste time.

I also like the idea of running small experiments to figure out how a certain command functions or how to make certain parts of my code better.