**MIPS Assignment 1**

Jonathan Isely

ECE-4612

Due: 9/22/23

**Objectives**

The objectives of this assignment were to refamiliarize us with assembly programming and learn the MARS IDE for MIPS development. Additional objectives were split across the three primary tasks we were to complete. The first task involved copying an array with our name to a new array in memory. The second task was used to familiarize us with basic I/O functionality including syscall, as well as utilize given code and our own modifications to find the factorial of a number. The final task involved using I/O functionality and more complex assembly code to print out a diamond of a width specified by the user.

**Tools/Equipment**

The primary tool utilized for this assignment was the MARS 4.5 IDE for use in compiling our MIPS assembly code. Additional tools include items such as laptops, calculators, etc.

**Analysis/Algorithms/Procedure**

Task 1

Task 1 involved utilizing a combination of given code (Figure 1), which was provided in the lab manual, and our own code to copy an array containing our name into a new spot in memory. The given code provided the functionality to copy an array, Y, to an array, X. This meant that all we had to do was enter an array containing our name into $a1, and we could copy it over to X, which was in $a0.

A screenshot of a computer code

Description automatically generated

Figure 1 - Code Provided for Task 1

To actually populate the array with our name, we utilized the .asciiz directive to specify a string, and then simply loaded it into $a1. The simulated memory was then analyzed to observe how the various ascii characters were organized in memory.

Task 2

Task 2 required our code to display a prompt in the I/O window, read in an integer, calculate the factorial, and finally display the result to the I/O window. As in Task 1, there was a sample of provided code, which handled the actual calculation of solving for the factorial. This code (Figure 2), required that the integer that the calculation would be run on be loaded into $a0. The given code would then store the result in $v0. Because the various syscalls for I/O utilize both $a0 and $v0, it was important to move the data that we wished to be stored to the relevant locations. In this case, the input had to be moved from $v0 to $a0 after it was read in, and the output had to be moved from $v0 to $a0 so that it could be displayed.

A screenshot of a computer program

Description automatically generated

Figure 2 - Code Provided for Task 2

Task 3

Unlike Task 1 and 2, Task 3 had no provided code. To accomplish this task, we broke it up into several component pieces. The first was the initial set up, which was comprised of reading in an integer that represented the width of the diamond, as well as assigning specific registers to maintain persistent data throughout the code (i.e., current line number, expected number of leading or middle spaces, total expected lines, etc.). After this set up, a set of instructions ran while the current line number was less than or equal to the specified width. These sets of instructions were specific to organizing the diamond while it was increasing in size. Once the current line count surpassed the original input width, the code iterated through a set of instructions that printed out formatted characters for the decreasing in width portion of the diamond. Per the assignment manual, detailed comments explaining the thought process behind each line of assembly are included in the assembly file.

Regarding portability of the code, none of the calculations are done via hardcoding. This ensures that no matter how small or large the specified width is, the code will output a formatted diamond. Part of this was figuring out how to calculate the number of leading spaces and middle spaces for each portion of the diamond. This is where storing the expected values of these two components in registers came in handy. By having a third register act as an incrementing counter of the currently printed spaces for either the leading or middle spaces, we were able to compare against the expected values to see whether we needed to loop through the print instructions again. When the condition was met, the counter was reset to 0, and either the expected leading space or expected middle space register was incremented or decremented based upon whether we were in the first or latter half of the diamond.

This was not a perfect process and some edge cases needed to be accounted for. An example of this is that when the diamond is at its widest, rather than go to the decreasing label immediately, it must first execute the commands under the decrement label. These commands only run once, but they allow the expected middle spaces value to behave properly. Additionally, to juggle between returning to the address we had “jump and linked” from under specific circumstances, I had to create a return label that’s sole purpose was to return to the address stored in $ra. This approach allowed me to check for various conditions and branch to the return label where needed. There likely was a better way to approach this, however I have not found it yet.

**Test/Results/Observations**

Task 1

Task 1 resulted in a variety of characters being stored in 2 separate locations in memory. The first was the input string, while the second was the output string. Figure 3 shows the input string (array Y) before the array of characters had been copied in memory. Figure 4 highlights the output string (array X) and shows how the characters were simply copied over.

A screenshot of a computer

Description automatically generated

Figure 3 - Input String

A screenshot of a computer

Description automatically generated

Figure 4 - Output String

An interesting thing to note is that while the words of 4 characters are displayed left to right graphically, the individual characters themselves are read right to left within the GUI. This is why Figure 4 is highlighted in two separate boxes.

Task 2

For task 2, the results displayed as expected. It’s important to note that the accuracy of the results is dependent upon utilizing the right number in $v0 during the syscall, as without it, the output would not be formatted correctly. With that being said, this task seemed more about refamiliarizing us with assembly, as the vast majority of code was provided for us. If we loaded in the right registers and handled the syscalls properly, we were able to get the appropriate output. An example of the output when computing the factorial of 9 can be seen below in Figure 5.

A black and white text

Description automatically generated

Figure 5 - Factorial of 9

Task 3

As mentioned in the procedure for Task 3, this task was significantly more in depth than the others. The amount of coding required was much greater, as there were several components that had to work in conjunction with one another. However, the results were successful in that they matched the task requirements and worked with large, small, even, and odd numbers. An example of the I/O window with a diamond of width 10 can be seen below in Figure 6.

A screenshot of a computer screen

Description automatically generated

Figure 6 - Diamond of width 10

Throughout the coding of this project, one observation made was that while some segments of code were reusable, several weren’t. This refers to the decrement instructions, the separate star instructions for either the left or right asterisk, and the separate increasing and decreasing functions. This makes it seem that there is likely greater efficiency to be found within this task, likely through the use of stack pointers and various additional temporary registers. Additionally, to complete this, a lot of the registers were always occupied with data. For this code it worked out, but if the assignment was more complex we could have feasibly run out of registers to store data. In short, there are several additional optimizations to be found within the code for this task.

**Conclusion**

This set of tasks served as a great refresher on assembly, while also providing a suitably challenging project in task 3. Given that it has been several semesters since last writing assembly, it was nice to be able to get the opportunity in tasks 1 and 2 to refresh our memory. Overall, the outcomes of the individual tasks were a success and allowed us to practice standard I/O, branching, syscalls, movement of data within memory, and the MIPS instruction set. All tasks were completed successfully, although there is definite room for improvement in task 3 once additional strategies for approaching assembly are learned. The only major roadblocks came in the form of tasks 1 and 2 at the start of the project, as it took a while to get used to writing assembly again. However, once the code was parsed through it made a lot more sense and allowed for a solid foundation upon which task 3 could be completed.