

Temple University College of Engineering

Department of Electrical and Computer Engineering (ECE)

Course Number : ECE 4612

Course Section : 001

Programming Assignment # : 1

Student Name (print) : Jon Eskow

TUId# : 915222192

Date : 09/30/16

Objective:

The objective of this assignment was to familiarize ourselves with the MIPS assembler by running several sample programs and writing three programs of our own. The sample programs were meant to be used as a reference that we could use in our own programs, we were also given several procedures to aid us in writing the code and combining everything together to finish the tasks that were assigned to us.

Tools:

The tools that were used was the MARS (MIPS Assembler and Runtime Simulator) IDE v4.5 meant for use with the Patterson and Hennessy *Computer Organization and Design* text. We also used our own computers to write and assemble the code. The computer that was used for the project in this particular write up was a Toshiba Satellite E45-B running Windows 10 with an Intel i5 processor running at 1.70GHz with potential boost of 2.4GHz.

Procedure/Analysis:

Task 1:

To start I ran the sample programs, HelloWorld (figure 1), Rtype(Figure 2), IntegerInput2Register(Figure 3), Print_Integer2Console(Figure 4), and StringInput_Output(Figure 5) and familiarized myself with the syntax used in each of these programs. I then used these programs as a guide to write and implement the following examples.

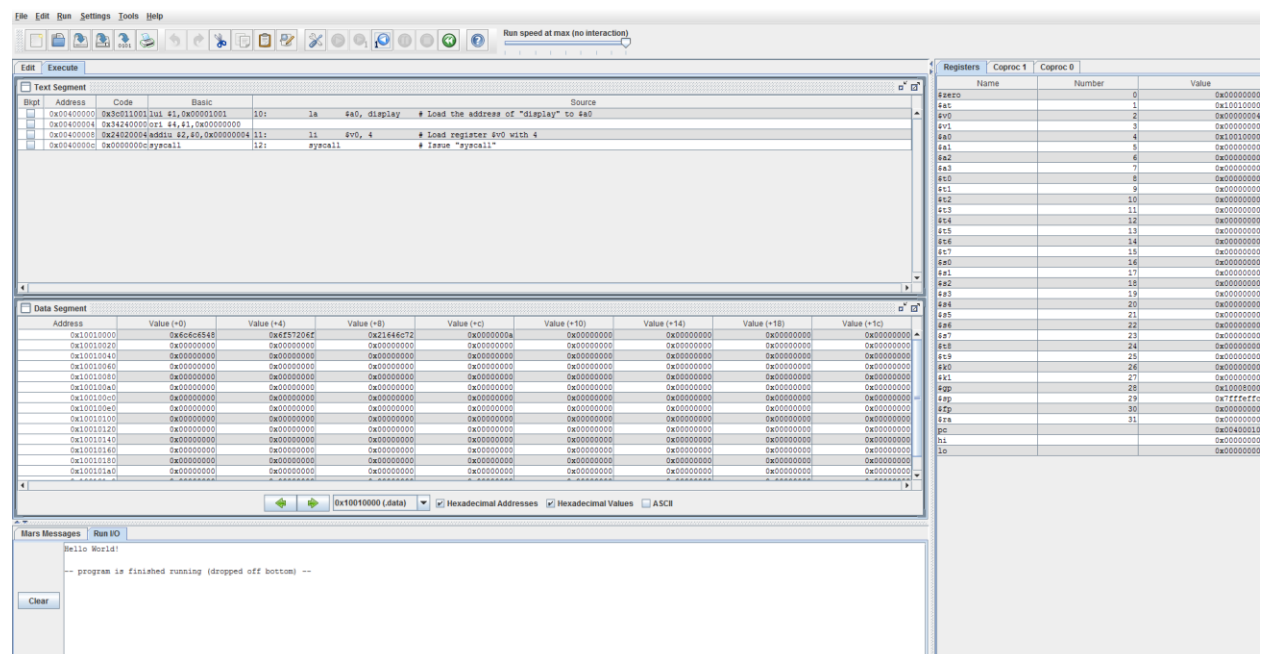


Fig 1. Output of HelloWorld program showing IO window

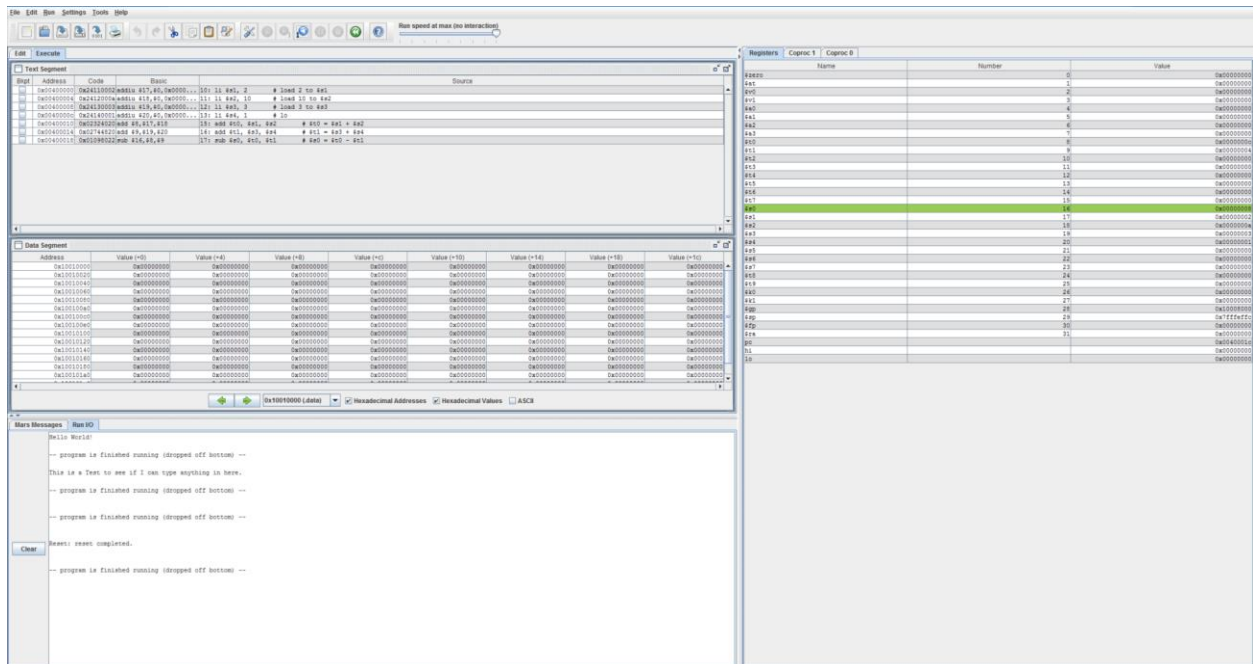


Fig. 2 Output of RType program showing operands and results

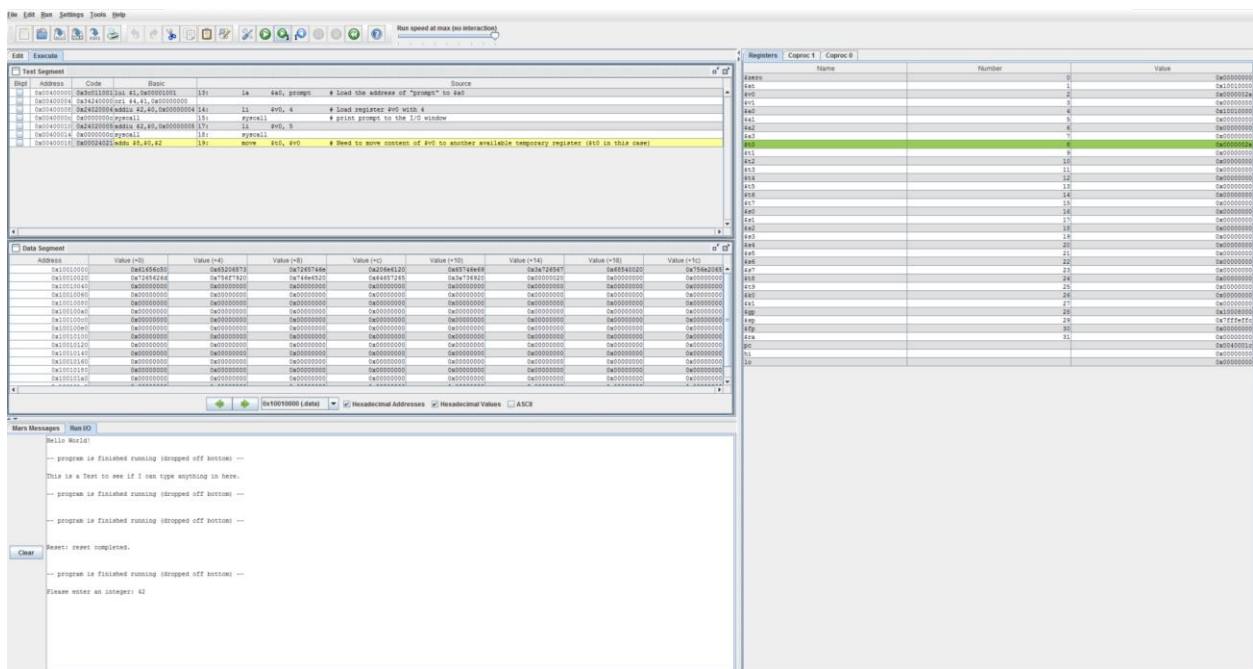


Fig. 3 Input and Output of IntegerInput2Register program

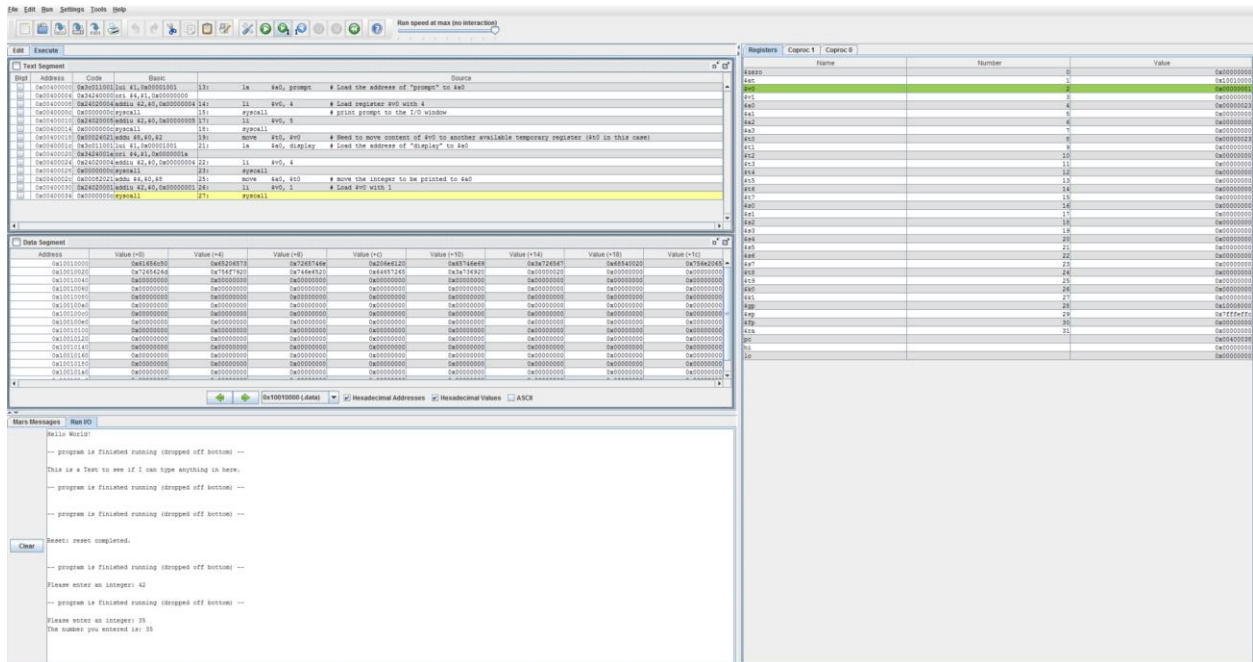


Fig 4. Input and Output of program `Print_Integer2Console`

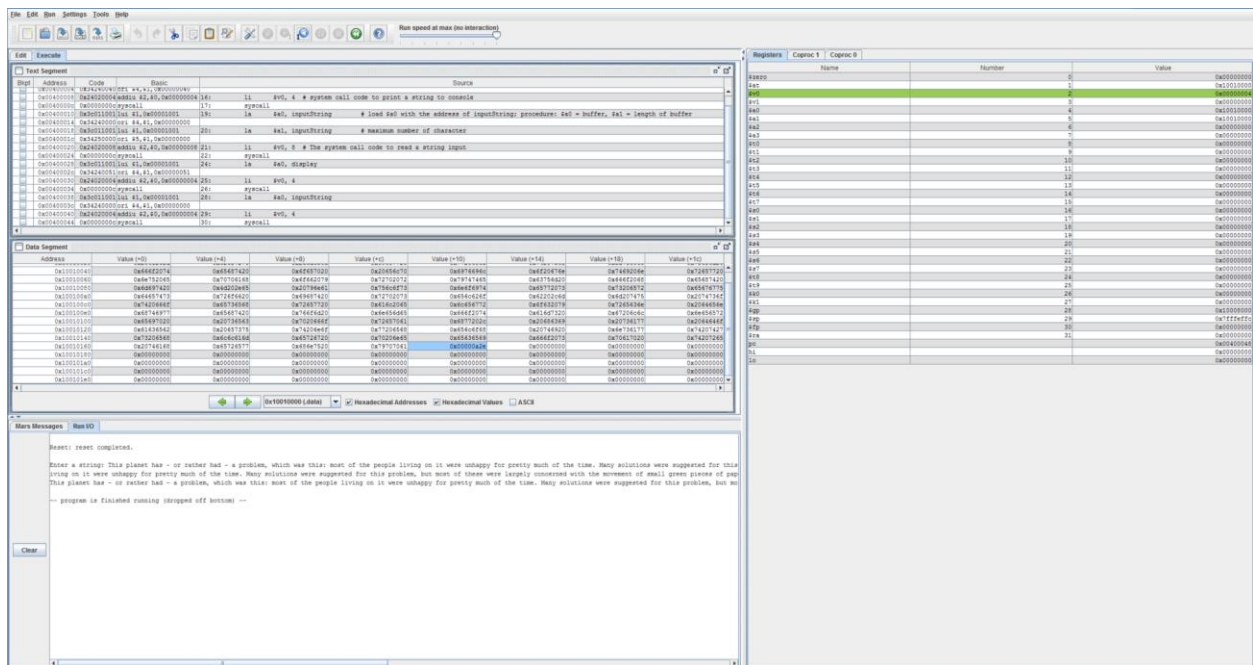


Fig 5. Input and Output of program `StringInput_Output`

Task 2:

For Task 2, we had to write MIPS code that allowed a user to enter a string, load it into an array, and then copy that array to a second array. We were given the copy procedure and were tasked to write the support code to prompt the user to enter the integer. While it was unclear if the copy completely worked by inspection of the Data Segment window, upon completion, the registers `a0` and `a1` (figure 6) contained the same value. As `x` and `y` were in registers `a0` and `a1` respectively, this was taken as a

C:\Users\Jonathan\Documents\Temp\advProc programming assignment 1\strCopy.asm - MARS 3.5

File Edit Run Settings Tools Help

Run speed at max (no interaction)

Edit Execute

Text Segment

Expr	Address	Code	Basic	Source
	0x00400000	0x00110001	0x00000001	0: la \$a0, prompt # display the prompt to begin
	0x00400004	0x00200001	0x00000004	1: sv \$v0, 4 # system call code to print a string to console
	0x00400008	0x00000000	0x00000000	2: syscall
	0x0040000C	0x00110001	0x00000001	3: la \$a0, inputString # load \$a0 with the address of inputString; pr...
	0x00400010	0x00240000	0x00000000	4: li \$v0, 4 # The system call code to read a string input
	0x00400014	0x00110001	0x00000001	5: la \$a1, inputString # maximum number of character
	0x00400018	0x00240000	0x00000000	6: li \$v0, 4 # The system call code to read a string input
	0x0040001C	0x00240000	0x00000000	7: syscall
	0x00400020	0x00300000	0x00000000	8: addi \$sp, \$sp, -4 #adjust stack for 1 item
	0x00400024	0x00000000	0x00000000	9: sw \$a0, 0(\$sp) #save \$a0
	0x00400028	0x00000000	0x00000000	10: addi \$sp, \$sp, 4 #restore \$a0
	0x0040002C	0x00000000	0x00000000	11: addi \$v0, \$v0, 1 #set \$v0 to 1
	0x00400030	0x00000000	0x00000000	12: addi \$v0, \$v0, 1 #set \$v0 to 2
	0x00400034	0x00000000	0x00000000	13: addi \$v0, \$v0, 1 #set \$v0 to 3
	0x00400038	0x00000000	0x00000000	14: addi \$v0, \$v0, 1 #set \$v0 to 4
	0x0040003C	0x00000000	0x00000000	15: addi \$v0, \$v0, 1 #set \$v0 to 5
	0x00400040	0x00000000	0x00000000	16: addi \$v0, \$v0, 1 #set \$v0 to 6
	0x00400044	0x00000000	0x00000000	17: addi \$v0, \$v0, 1 #set \$v0 to 7
	0x00400048	0x00000000	0x00000000	18: addi \$v0, \$v0, 1 #set \$v0 to 8
	0x0040004C	0x00000000	0x00000000	19: addi \$v0, \$v0, 1 #set \$v0 to 9
	0x00400050	0x00000000	0x00000000	20: addi \$v0, \$v0, 1 #set \$v0 to 10
	0x00400054	0x00000000	0x00000000	21: addi \$v0, \$v0, 1 #set \$v0 to 11
	0x00400058	0x00000000	0x00000000	22: addi \$v0, \$v0, 1 #set \$v0 to 12
	0x0040005C	0x00000000	0x00000000	23: addi \$v0, \$v0, 1 #set \$v0 to 13
	0x00400060	0x00000000	0x00000000	24: addi \$v0, \$v0, 1 #set \$v0 to 14
	0x00400064	0x00000000	0x00000000	25: addi \$v0, \$v0, 1 #set \$v0 to 15
	0x00400068	0x00000000	0x00000000	26: addi \$v0, \$v0, 1 #set \$v0 to 16
	0x0040006C	0x00000000	0x00000000	27: addi \$v0, \$v0, 1 #set \$v0 to 17
	0x00400070	0x00000000	0x00000000	28: addi \$v0, \$v0, 1 #set \$v0 to 18
	0x00400074	0x00000000	0x00000000	29: addi \$v0, \$v0, 1 #set \$v0 to 19
	0x00400078	0x00000000	0x00000000	30: addi \$v0, \$v0, 1 #set \$v0 to 20
	0x0040007C	0x00000000	0x00000000	31: addi \$v0, \$v0, 1 #set \$v0 to 21
	0x00400080	0x00000000	0x00000000	32: addi \$v0, \$v0, 1 #set \$v0 to 22
	0x00400084	0x00000000	0x00000000	33: addi \$v0, \$v0, 1 #set \$v0 to 23
	0x00400088	0x00000000	0x00000000	34: addi \$v0, \$v0, 1 #set \$v0 to 24
	0x0040008C	0x00000000	0x00000000	35: addi \$v0, \$v0, 1 #set \$v0 to 25
	0x00400090	0x00000000	0x00000000	36: addi \$v0, \$v0, 1 #set \$v0 to 26
	0x00400094	0x00000000	0x00000000	37: addi \$v0, \$v0, 1 #set \$v0 to 27
	0x00400098	0x00000000	0x00000000	38: addi \$v0, \$v0, 1 #set \$v0 to 28
	0x0040009C	0x00000000	0x00000000	39: addi \$v0, \$v0, 1 #set \$v0 to 29
	0x004000A0	0x00000000	0x00000000	40: addi \$v0, \$v0, 1 #set \$v0 to 30
	0x004000A4	0x00000000	0x00000000	41: addi \$v0, \$v0, 1 #set \$v0 to 31
	0x004000A8	0x00000000	0x00000000	42: addi \$v0, \$v0, 1 #set \$v0 to 32
	0x004000AC	0x00000000	0x00000000	43: addi \$v0, \$v0, 1 #set \$v0 to 33
	0x004000B0	0		

Task 3:

For Task 3 we had to take an Integer input, run a factorial calculation on it, and output the value. Similar to the other tasks we were given a piece of startup code and wrote support code to allow a user input and provide an easy to read output. The solution to this task borrowed from the two integer sample programs and then rewrite some of the code to make sure the inputs and outputs are going to and from the right registers. While running some quick tests, it was found that if the factorial was too great the output would be a negative number. This makes sense as the integers being input and output are not specified as “unsigned” so the output would be a signed number and if the number is large enough the register outputting the answer is going to get confused and show the factorial of the input as a negative. This could be remedied by specifying all values as unsigned integers.

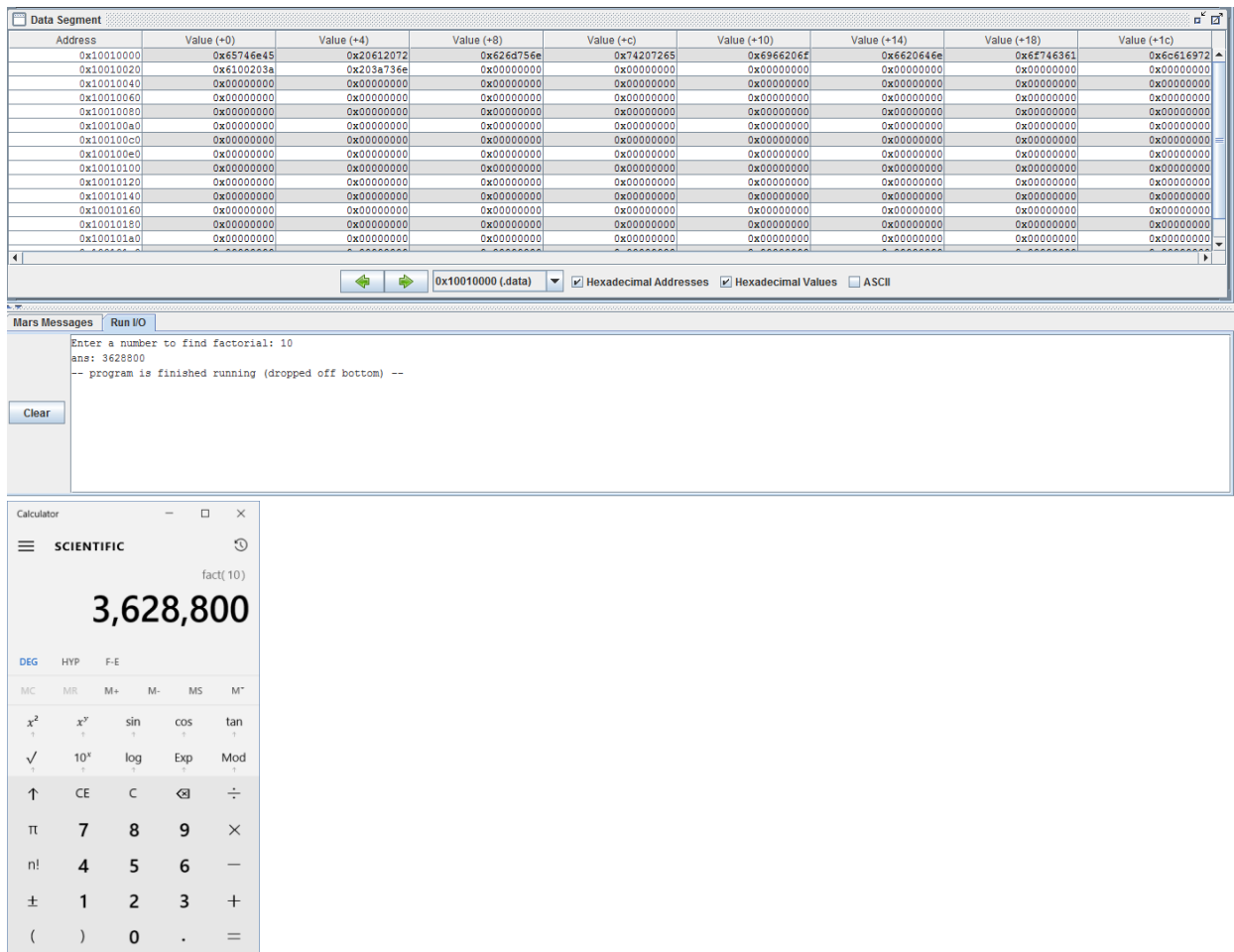


Fig 7: Output of factorial program along with windows calculator to confirm the arithmetic is correct.

Task 4:

The final task was to insert a series of 5 integers and return them sorted. In this example we were given a piece of code that performs a bubble sort on a series of integers. The algorithm in C is

```
void sort (int v[], int n)
{
    int i,j;
    for (i = 0; i < n ; i ++)
    {
        for (j = i-1; j >= 0 && v[j] > v[j+1]; j--)
        {
            swap(j,i);
        }
    }
}
```

It takes a value and compares that value with the next value in the array and if one is smaller than the other, then the positions in the array of both elements are swapped. I was unable to get my own version

of this program to work. I tried to solve it by giving several prompts to load places on the stack with integers and while the bubble sort works, my implementation fails in the output because all values are just popped off the stack and are never read out to the IO window.

Conclusion:

MIPS is one of the most common architectures in use and Assembly language allows the designer and programmer to intimately work within the system with maximum control over everything that happens at the register level. Additionally, assembly provides the designer more space as there are no complex compilers to work with providing a means of maximizing procedure and minimizing space. That said, coding in assembly is arduous and complex programs end up having many times more lines of code than a higher level (i.e. C) script which also can lead to a higher probability of mistakes. Assembly is effective if used appropriately, but just because a program is written in assembly doesn't mean it will take up less space and processing power than a similar program written in a higher level language as an algorithm written in assembly could be half as efficient as an algorithm written in higher level languages through sheer lack of understanding on the programmer's part. Assembly is a tool, and as with any other tool, it must be used properly to be effective.