ECE 200/400: Project 1 - MIPS Assembly Language Programming

Due Date: February 28th, 2020 11:59p NO EXTENSIONS

1 Introduction

The goal of this project is to help you to better understand low level machine instructions by getting your hands dirty on writing, running, and debugging assembly code. You will implement four algorithms (from easy to hard) in MIPS assembly language.

You can form a group of **at most two (2)** students to finish this project, but you don't have to if you feel more comfortable working by yourself. If you work in groups, please make sure both students do the same amount of work. You are encouraged to help each other. However, do not share your code (unless you are in the same group, of course).

2 Environment and Tools

In this project, we will use MARS (MIPS Assembler and Runtime Simulator). You can write MIPS assembly code in its built-in editor, assemble and run code by using its assembler and simulator. To download and learn more about MARS, please visit:

https://courses.missouristate.edu/KenVollmar/MARS/

MARS is written in Java so it can run on major platforms (Windows, OSX, Linux), including school library computers. You should have a Mars4 5.jar file after downloading. To get it running, you need the JavaSE environment. If you do not have it, please follow the instructions on the MARS website to install J2SE (download J2SE at:

https://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html

If you can run MARS successfully, you should see its editor UI as shown in Figure 1. You can edit your code here. Alternatively, you can use your favorite text editor and come back to assemble the source code and run.

To assemble the code, click Run => Assemble on the menu or press F3. If your code is successfully assembled, you will come to the execution interface automatically (as shown in Figure 2). Here, you can see the memory space and all your registers. You can either choose to run all of the code at one time, or you can run your code instruction by instruction (step in). There are also many other useful features in MARS; explore them as you are using it.

3 Task

In this project, you are to implement the following four algorithms:

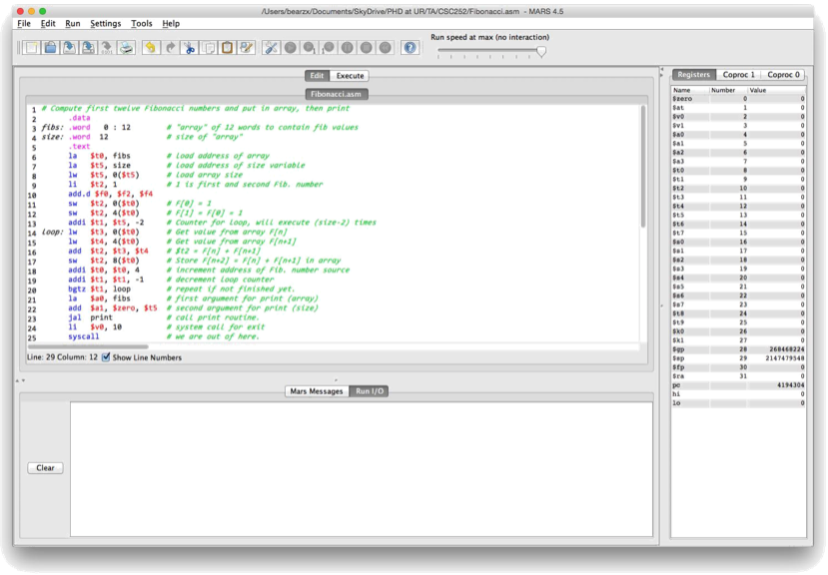

Figure 1: MARS Editor

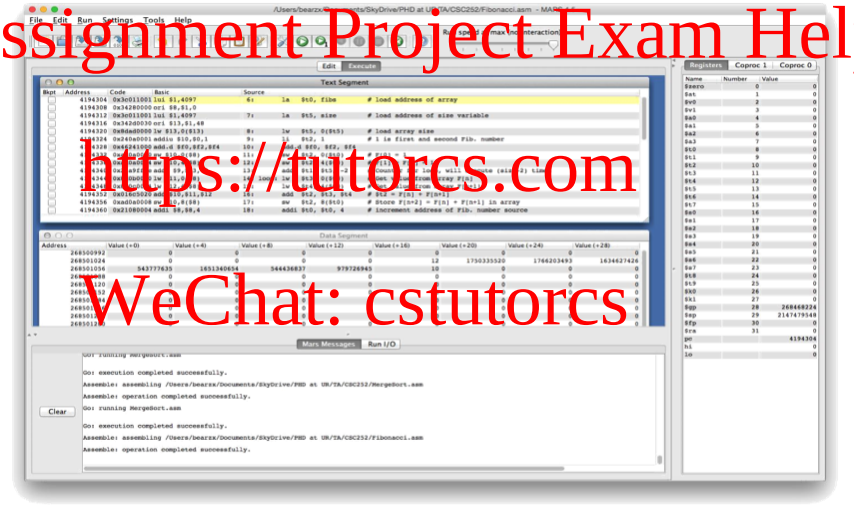

Figure 2: MARS Execution Environment

- Binary Search
- Matrix-Matrix Multiplication
- Find min, max, and median
- Merge Sort

3.1 Binary Search

Binary search (See https://www.geeksforgeeks.org/binary-search/) is an algorithm that can quickly find out if a certain element is in a sorted array or not (and return the index if it is

in the array). You can define the array (integer, sorted) and the target number in your data segment beforehand. Print the index to the console (by a syscall) if the element is found, otherwise print a "Target not found" message.

You can define your input in the data segment as shown below:

.data

length: .word 10

nums: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

target: .word 11

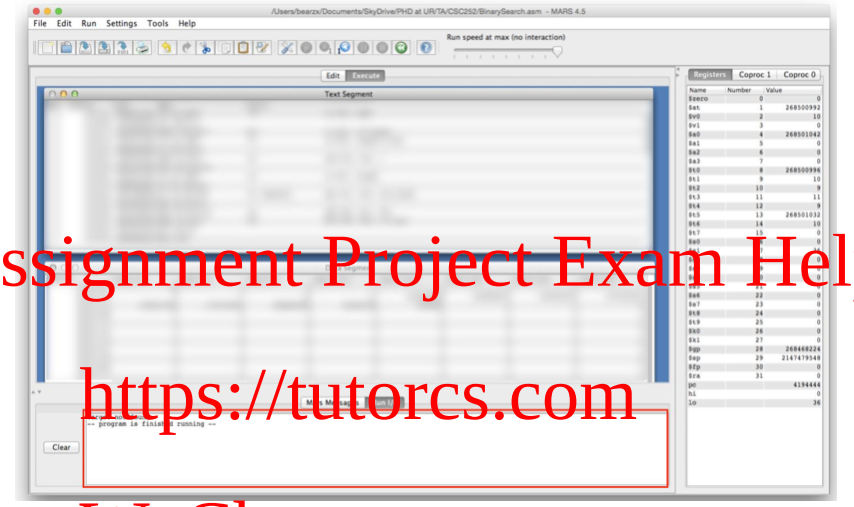

Figure 3: MARS Print to Console

In this example, we are searching for "11" from 1 to 10. Apparently "11" is not in our array, so we print a "Target not found" message (as shown in Figure 3). If the target number is "2", we should print its index in the array (which is 1).

3.2 Matrix-Matrix Multiplication

Define two column vectors X, Y and one Matrix B in data segment. Write an assembly language program to transpose Y. Multiply X to transpose Y and perform multiplication between their product and Matrix B. (See https://www.geeksforgeeks.org/c-program-multiply-two-matrices/)

For example: dimension of X and Y are (3,1) (3 rows, 1 columns) $--> $ Transpose Y will have dimension of (1,3). Then X.Y will give you a 3x3 matrix

You can define your input in the data segment as shown below:

.data

sizeB: .word 2, 3

matrixB: .word 5, 6, 7, 8, 9, 10

sizeX: word 3, 1

sizeY: word 3, 1

matrixX: word 1, 2, 3

matrixY: word 1, 2, 3

result: .word 0:9

You should put these numbers (row- first) in the memory space pointed by the "result" label.

3.3 Find min, max, and median

Define an integer array (unsorted) in your data segment, and write an assembly program to find the minimum, maximum, and median of the array. Note that you may want to sort the array first, because once the array is sorted, minimum, maximum and median are just numbers at certain indices. You can use your merge sort code. However, if you find it difficult to implement merge sort, just use a sort algorithm that is simpler to implement (e.g., bubble sort). Results should be saved in a pre-defined memory space. See geeks for geeks or other resource for implementation.

You can define your input in the data segment as shown below:

.data

length: .word 10

nums: .word 92, 31, 92, 6, 54, 54, 62, 33, 8, 52

min: .word 0

max: .word 0

median: .word 0

In this example, you should save the minimum, maximum and median to the memory space pointed by the labels "min", "max" and "median", respectively. Normally, for an even size array the median is defined as the average of the two numbers in the middle. For simplicity, you can just take the number at index [length/2].

3.4 Merge Sort

Merge sort (See https://www.geeksforgeeks.org/merge-sort/) is one of the classic sorting algorithms. The idea of merge sort is to constantly merge small pieces of sorted list into bigger ones. Implement merge sort to sort a pre-defined array in your data segment in ascending order. The sorted results should be saved in a pre-defined memory space.

You can define your input in the data segment as shown below:

.data

length: .word 10

nums: .word 10,9,8,7,6,5,4,3,2,1

sorted: .word 0:10

In this example, you should save the sorted array to the memory space pointed by the label "sorted". You are also welcome to save the sorted array back to "nums".

Please note that merge sort can be implemented iteratively or recursively. If you decide to use a recursive implementation, make sure you handle the call stack correctly.

4 Restrictions

The following section specifies some settings that are or are not allowed.

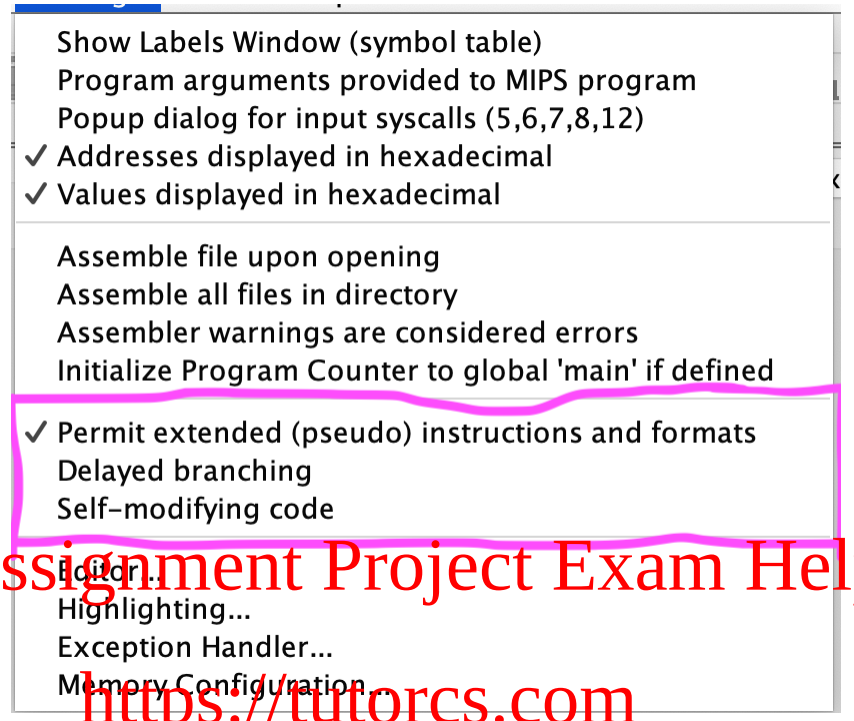

Figure 4: MARS Settings

4.1 Delayed Branching in MARS

We mentioned in class that the instruction following a branch instruction will always be executed no matter what the condition of that branch is. This is known as the "branch delay slot". Usually we put a "nop" instruction in the slot. You can also use this property to optimize your code (e.g., reducing instruction counts).

However, the assembler of MARS will by default disable this feature. To enable it you need to check the ag at Settings Delayed branching. Alternatively, you can just keep the default setting if you do not feel like manually adding "nop" after every branch instruction.

To toggle this functionality refer to Figure 4.

4.2 Pseudo-Instructions

You may not use pseudo-instructions. We will test your code with this box disabled. Thus, if you have "pseudo-instructions" in your code it will not run and you will receive a **zero**.

To disable this functionality refer to Figure 4.

5 Tips

While you are working on these tasks, you may find that even the simplest stuff (e.g., reading an array element A[i]) is not that trivial in assembly language, and it is much easier to introduce bugs that are not easy to notice. Here are some tips:

- When writing code in assembly language, try to think at a low level (machine level) of abstraction.
- If you can't write code in assembly language directly, at least you can implement the algorithms in a high-level language first. Then translate it from your high-level code to low level assembly code.
- Running your code step by step is always a helpful way to find bugs. Make assumptions on the register contents, memory contents and branch conditions, and compare your assumptions with the real results.
- Always comment your code. For example, you may want to put notes on which register is for which purpose.
- The programs do not have to be recompiled line by line from C. The programs have to execute the described functionality. C programs are only there to assist you.

6 Submission

You are to electronically turn in the following via Blackboard:

1. Source file named as "BinarySearch.asm" for task 1
2. Source file named as "MatrixMultiplication.asm" for task 2
3. Source file named as "MinMaxMedian.asm" for task 3
4. Source file named as "MergeSort.asm" for task 4
5. A README file, in text format, that contains:
   - Names of the students in your group, and a description of which partner implemented which code.
   - Other important things you want us to know about your code.
   - If you are participating in the extra credit challenge.

- You may work in groups of up to two (2)
- Only one partner should submit the code if working in a group.
- You may submit multiple times. **Only the last submission before the deadline will be graded.**

- **Please do not submit version control files.**

You should submit your files as a single .tar.gz archive, named as "LastNameFirstName.tar.gz" ("name1_name2.tar.gz" for a group).

7 Grading

Ten (10) points for each task; you get the point if your programs run correctly (no assembling error, no runtime error, outputs correct results) and your implementations satisfy our requirements (e.g., implementing linear sort in the merge sort task does not count as a correct solution although the two algorithms yield the same results). And five (5) more point for your comments (both README and in-line comments) and using the right formats.

7.1 Extra Credit

For the merge sort task, (assuming all of your code works correctly) we will choose three (3) implementations with the smallest dynamic instruction counts. We will measure the dynamic instruction counts by using the Tools Instruction Counter tool (Figure 5). The test will be merge sorting an array of length n. **In your code, clearly label spaces to add the array, and the size.** For example, a test may be sorting[10,9,8,7,6,5,4,3,2,1] to [1,2,3,4,5,6,7,8,9,10] given a length of 10.

You must complete project 1 completely and correctly in order to qualify for extra credit. When attempting the extra credit, some optimizations you may want to consider are analyzing the branch delay slot (disabling), the load delay slot, and other excess instructions that do inconsequential "work".
Prizes[1] are:

1. 100% of the points in Project 1
2. 50% of the points in Project 1
3. 25% of the points in Project 1

8 IMPORTANT!

If you fail to meet the following requirements you risk getting a large deduction to your project grade.

1. Do not make assumption about the input to any of your programs. In other words, all 4 of your programs and extra credit assignment must work with an arbitrary input. For example, your Merge Sort algorithm may be tested with an array of size 50 with

[1]Prizes are awarded to **EACH** member of the 2-person team. That is, in a 2-person group each person will receive the bonus points *e.g.*, In the case of first place, both will receive 100% extra credit. If a prize is won by a team of one (1), then they will only receive 100%.

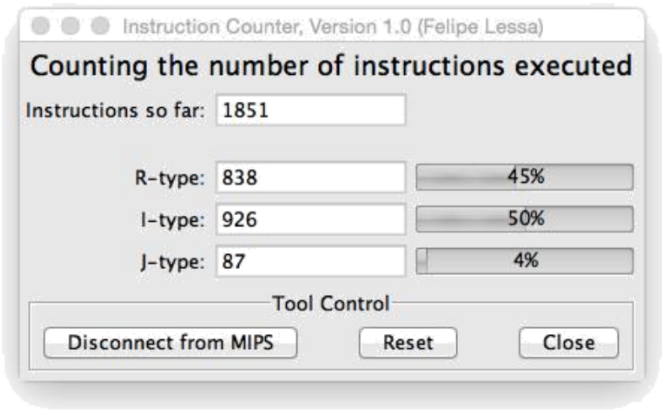

Figure 5: Instruction Counter (just for your reference my un-optimized code consumed 1851 dynamic instructions to sort [10,9,8,7,6,5,4,3,2,1])

number values ranging from 0 to 100, or it may be tested with an input array of size 1. In both cases it must execute correctly and not throw exceptions. The only assumption you may have is that the size data value will indicate the exact number of elements in an array.

2. The programs must print the result. If the program does not print the result, I will not be looking through memory for each value and you will receive a zero for the program.

3. All programs 1-4 must execute correctly for you to qualify for the competition.

4. All programs must be implemented as stated in the description. You may not use different algorithms. For example, you cannot implement Count Sort instead of Merge Sort or Linear Search instead of Binary Search.

5. The use of online code compilers is obvious and will result in a 0 project grade and a report to Academic Honesty board.

6. Pseudo Instructions must be disabled!