CS 252 (Summer 20): Computer Organization

# Assembly Project #1

Introduction to MIPS

due at 5pm, Fri 26 Jun 2020

## 1 Purpose

In this project, you'll be getting some basic experience with how to write assembly language. You will write a simple function (which we'll name `studentMain()`), and in it you will perform some simple tasks.

### 1.1 Required Filenames to Turn in

Name your assembly language file `asm1.s`.

You will also turn in a short report about the code that you wrote. The purpose of this is primarily to make sure you actually start using MARS for development (rather than simply leaning on the grading script). Using MARS will help you understand things better - even if it takes a little bit of time to get to know it.

To make sure we can read it, your report **must** be a PDF.

### 1.2 Resources

Before you get started, make sure to check out my Panopto video, which I've posted to the class, which gives you a good introduction into how to use the MARS simulator.

Also, you should go to the class website, and look up the assembly language style guide that I've posted. The TAs will be checking to verify that you followed the style guide!

### 1.3 Files

You can find the files for this project in the following locations:

- The GitHub Classroom repository
- The project directory on the class website: `http://lecturer-russ.appspot.com/classes/cs252/summer20/asm/asm1/`.
- The mirror of the class website, accessible on any department computer: `/home/russelll/cs252m20_website/`

## 2 The Report

**NOTE:** You will need to compose your code before you can write the report. So read the section on the task requirements first, and also read up on how to run using the MARS gui. But after your code is ready, come back and write this report.

For this project, in addition to writing some assembly, you also need to write a short report, detailing some features of your code. To find out the required information, load up your solution (`asm1.s`), and also one of the testcases. The easiest way to do this is to put both of these files in the same directory (with no other `.s` files), and then turn on the "Compile all files in directory" option of MARS.

Assemble these two files, and do a quick test run to make sure that your code works (check to see if the output produced looks right; for this part of the project, you don't have to confirm that it's perfect).

For the first part of your report, you need to find some place where you used an `LA` instruction to get the address of some variable provided to you by the testcase. In MARS, you can do this as follows:

- If it's not already active, click on the "Execute" tab, near the top left of the window.
- Inside Execute, you should see the "Text Segment" at the top. This is your code. Scroll up and down that window until you see your `LA` instruction. (Use the line numbers to help you find it.)
- You will find that the `LA` instruction is actually a **pseudoinstruction** - it actually is implemented as 2 instructions in hardware.
- Set a breakpoint at your LA instruction. (Breakpoints are the check-boxes on the left edge of the Text Segment window.) Then re-run the program again from the beginning. Your program will automatically pause when your LA instruction is **just about to run.**
- Look for the "Run one step at a time" button on the toolbar; use it to run exactly two instructions. You will see that the first one modifies the register `$at` (MARS will highlight that register in the "Registers" pane on the right). The second instruction is the one that actually will modify the register that you mentioned in your LA instruction.
- **First item for your report** - The address of some variable.

  Now, look more closely at what LA did. It put the address of the variable into the register you chose. Give the address in hexadecimal.

  **If all of the registers are in decimal instead of hexadecimal,** look at the Settings menu to change them to hexadecimal.

- **Second item for your report** - The two instructions that make up LA

  Next, look at the two instructions which make up the LA pseudoinstruction (the first is LUI). Write down, for your report, exactly what MARS shows in the Basic column of the Text Segment for these two instructions. The register names won't look familiar - this column uses register **numbers** instead of names - but the second instruction should have a number that you've seen before.

- **Third item for your report** - Hex encodings

  Finally, write down the actual 32-bit hex encodings of these two instructions (which you can find in the Code column). Do you see the constants used by these instructions stored inside those encodings somewhere?

Thus, the first part of your report should include:

- The full 32-bit address of the variable, in hex.
- The full text of the assembly instructions which make up the LA pseudoinstruction.
- The encodings of both instructions, also as 32-bit hex numbers.

## 2.1 Strings and Data

The second part of your report will observe how strings are loaded into memory. (Choose any one of the string constants that you used in your code.)

Find where you declared this constant in assembly (probably an `.asciiz` directive). Look at the first four characters of that string. Compare these four to an ASCII table (I link to one from the class webpage, but basically any ASCII table will work). What are the ASCII codes (in hex) for these four characters?

Now, find the LA instruction in your code which reads this string. Run the program to that point, and use it to figure out the address of this string in memory. (It probably starts with 0x1001....)

Now, look at the Data Segment window (just below the Text Segment). Each row of this table represents 32 (0x20) bytes of data; on the left, you can see the addresses that the rows represent. Scroll until you can see the region of memory where your string is.

(If you are **completely** in the wrong section of memory, use the drop-down at the bottom of the Data Segment window to choose what region you're looking at. You want the `.data` section.)

Once you've found the right row, look across the columns to find the cell that represents where your string is. (Each cell is 4 bytes of data.) Look for hex values that match the ASCII codes you're expecting. If your string address is a multiple of 4, then they should all be inside the same word of memory; if your address is not aligned (which is just fine for strings), then they will be spread across two words.

What do you notice? Probably, you'll see that the characters appear to be in reverse order!

For the second part of your report, write down:

- The entire string constant from your code that you chose to use.
- The ASCII codes (in hex) for the first 4 characters in that string.
- The address where the string ended up.
- The word (or maybe 2 words) in memory which contained the ASCII codes you were expecting.

Finally, write at least one paragraph about your investigation. You can discuss any one (or more) of the following questions:

- What was something new that you learned from this exercise, and how might you use it in the future?
- What questions do you have that were inspired by this exercise?
- Explain your process for finding the information required by this exercise. Include screenshots to show what you saw, and discuss a bit of what you did.
- I mentioned that the constant loaded by the LA instruction is present in the two instructions that LA becomes. Looking at the instruction encodings, what can you conclude about where the constants are stored?

Turn in all three parts of your report to GitHub, along with your code.

# 3 Allowable Instructions

When writing MIPS assembly, the only instructions that you are allowed to use (so far) are:

- `add, addi, sub, addu, addiu`
- `and, andi, or, ori, xor, xori, nor`
- `beq, bne, j`
- `slt, slti`
- `sll, sra, srl`
- `lw, lh, lb, sw, sh, sb`
- `la`
- `syscall`

While MIPS has many other useful instructions (and the assembler recognizes many pseudo-instructions), **do not use them!** We want you to learn the fundamentals of how assembly language works - you can use fancy tricks after this class is over.

**WARNING:** If you use instructions outside of this set - or, if you don't give them the right types of parameters, our "MIPS checker" script will discover this, and you will lose half of your testcase points.

# 4 Standard Wrapper

Your code will need to define a single function, named `studentMain()`. Later, when you learn how to write functions, things will get more interesting; for now, you should just copy-paste the code below.

The following lines of code should be placed before your first instruction. This declares the function `studentMain()`, makes it available to the testcase, and then also gives the function "prologue" - that is, the basic code to start a function.

```
.globl studentMain
studentMain:
    addiu $sp, $sp, -24   # allocate stack space -- default of 24 here
    sw    $fp, 0($sp)     # save caller's frame pointer
    sw    $ra, 4($sp)     # save return address
    addiu $fp, $sp, 20    # setup main's frame pointer
```

The following lines of code should be placed after your last instruction; this implements the "epilogue", which basically is the `return` statement at the end of your function.

```
  lw    $ra, 4($sp)     # get return address from stack
  lw    $fp, 0($sp)     # restore the caller's frame pointer
  addiu $sp, $sp, 24    # restore the caller's stack pointer
  jr    $ra             # return to caller's code
```

(Both of these pieces should be inside a `.text` section.)

# 5 Task Overview

Your code (inside the `studentMain()` function) will read four variables, named `equals, order, reverse, print`. Each is a word, and will be either 1 or 0. Each represents a single operation you might perform; load each one, and if the variable is 1, then perform that task. Note that some testcases will have you perform no tasks; others may have you perform several, or even all of them.

Perform the tasks in the order listed in the spec; if you change the order, you will not pass the testcase.

In addition, you will be given six variables to read: `red, orange, yellow, green, blue, purple`. All are words. These will be used as the inputs to the tasks below.

## WARNING WARNING WARNING

You must **never** assume that you know the relative arrangement of variables in any of the projects this semester - except when you know you have an array. So don't assume that `charlie` is 4 bytes beyond `bravo`, and load it using an offset! Instead, always load the address of each variable independently.

(To check that you're doing this, some of my testcases will shuffle up the order of the variables!)

### 5.1 Task 1: Equals

**NOTE:** For this task, you will **only** use the first four of the six input variables: `red,orange,yellow,green`.

For this task, you will read the four variables and compare them to each other. If any variable is equal to any other, then print `"EQUALS"`. If none of them are, then print `"NOTHING EQUALS"`.

### 5.2 Task 2: Order

**NOTE:** For this task, you will use **all six values.**

Read all of the values. Check to see if they are all in order - either ascending or descending.

If they are all in ascending order, then print `"ASCENDING"`. If they are all in descending order, then print `"DESCENDING"`. If they are **all** equal, then print `"ALL EQUAL"`. If none of these are true, then print `"UNORDERED"`. (Note that, in an ascending list, some values may be equal; for instance the list 1,2,2,3,4 is in ascending order. The same is true of a descending list.)

**WARNING:** This task could require a **monstrous** amount of code, if you aren't careful! Play around with this problem, in the language of your choice, before writing the assembly - there are tricks to make this (relatively) painless. For instance, could you break each possible case down into a single (complex) if() statement in C?

### 5.3 Task 3: Reverse

For this task, read all six values, and write them back to memory in the reverse order they previously were.

Then print `"REVERSE"`.

**NOTE:** It's ok to keep the variables in registers, temporarily. However, this operation **requires** that you write the values out to memory (my testcase will

check!). Also, make sure that the values you print in the next step (if it's turned on) are correct. If you are keeping things in registers, then update those registers.

### 5.4 Task 4: Print

For this task, print out the four values, as shown below. Put each value on its own line, with a string in front of it, giving its name.

It should look like this:

```
red: 123
orange: 456
yellow: 789
green: -1
blue: 1024
purple: 3
```

## 6 Implementation Hint

One possible way to implement this project, which saves on code duplication, is to load **all** of the variables into registers first, before you do any checking. (Note: If you do this, you still ought to store into memory in the "copy" task, even if you also have a copy in a register.)

If you do this, then document which registers are used for each variable, with a block comment at the head of your function. Be careful not to modify any of these registers - otherwise, actions taken during one task might affect another.

### 6.1 Requirement: Don't Assume Memory Layout!

It may be tempting to assume that the variables are all laid out in a particular order. Do not assume that! Your code should check the variables in the order that we state in this spec - but you **must not** assume that they will actually be in that order in the testcase. Instead, you must use the `la` instruction for **every variable** that you load from memory.

To make sure that you don't make this mistake, we will include testcases that have the variables in many different orders.

## 7 Running Your Code

You should always run your code using the grading script before you turn it in. However, while you are writing (or debugging) your code, it often handy to run the code yourself.

## 7.1 Running With Mars (GUI)

To launch the Mars application (as a GUI), open the JAR file that you downloaded from the Mars website. You may be able to just double-click it in your operating system; if not, then you can run it by typing the following command:

```
java -jar <marsJarFileName>
```

This will open a GUI, where you can edit and then run your code. Put your code, plus **one**[1] testcase, in some directory. Open your code in the Mars editor; you can edit it there. When it's ready to run, assemble it (F3), run it (F5), or step through it one instruction at a time (F7). You can even step **backwards** in time (F8)!

### 7.1.1 Running the Mars GUI the First Time

The first time that you run the Mars GUI, you will need to go into the `Settings` menu, and set two options:

- `Assemble all files in directory` - so your code will find, and link with, the testcase
- `Initialize Program Counter to 'main' if defined` - so that the program will begin with `main()` (in the testcase) instead of the first line of code in your file.

## 7.2 Running Mars at the Command Line

You can also run Mars without a GUI. This will only print out the things that you explicitly print inside your program (and errors, of course).[2] However, it's an easy way to test simple fixes. (And of course, it's how the grading script works.) Perhaps the nicest part of it is that (unlike the GUI, as far as I can tell), you can tell Mars exactly what files you want to run - so multiple testcases in the directory is OK.

To run Mars at the command line, type the following command:

```
java -jar <marsJarFileName> sm <testcaseName>.s <yourSolution>.s
```

# 8 A Note About Grading

Your code will be tested automatically. Therefore, your code must:

[1] Why can't you put multiple testcases in the directory at the same time? As far as I can tell (though I'm just learning Mars myself), the Mars GUI only runs in two modes: either (a) it runs only one file, or (b) it runs **all** of the files in the same directory. If you put multiple testcases in the directory, it will get duplicate-symbol errors.

[2] Mars has lots of additional options that allow you to dump more information, but I haven't investigated them. If you find something useful, be sure to share it with the class!

- Use exactly the filenames that we specify (remember that names are case sensitive).
- **Not** use any other files (unless allowed by the project spec) - since our grading script won't know to use them.
- Follow the spec precisely (don't change any names, or edit the files I give you, unless the spec says to do so).
- (In projects that require output) match the required output **exactly!** Any extra spaces, blank lines misspelled words, etc. will cause the testcase to fail.

To make it easy to check, I have provided the grading script. I **strongly recommend** that you download the grading script and all of the testcases, and use them to test your code from the beginning. You want to detect any problems early on!

## 8.1 `mips_checker.pl`

In addition to downloading `grade_asm1`, you should also download `mips_checker.pl`, and put it in the same directory. The grading script will call the checker script.

## 8.2 Testcases

You can find a set of testcases for this project in the GitHub Classroom; they are also present on the class website.

For assembly language programs, the testcases will be named `test_*.s` . For C programs, the testcases will be named `test_*.c` . For Java programs, the testcases will be named `Test_*.java` . (You will only have testcases for the languages that you have to actually write for each project, of course.)

Each testcase has a matching output file, which ends in `.out`; our grading script needs to have both files available in order to test your code.

For many projects, we will have "secret testcases," which are additional testcases that we do not publish until after the solutions have been posted. These may cover corner cases not covered by the basic testcase, or may simply provide additional testing. **You are encouraged to write testcases of your own, in order to better test your code.**

## 8.3 Automatic Testing

We have provided a testing script (in the same directory), named `grade_asm1`, along with a helper script, `mips_checker.pl`. Place both scripts, all of the testcase files (including their `.out` files), and your program files in the same directory. (I recommend that you do this on Lectura, or a similar department machine. It **might** also work on your Mac or Linux box, but no promises!)

### 8.4 Writing Your Own Testcases

The grading script will grade your code based on the testcases it finds in the current directory. Start with the testcases I provide - however, I encourage you to write your own as well. If you write your own, simply name your testcases using the same pattern as mine, and the grading script will pick them up.

While you normally cannot share code with friends and classmates, **testcases are the exception.** We encourage you to share you testcases - ideally by posting them on Piazza. Sometimes, I may even pick your testcase up to be part of the official set, when I do the grading!

## 9 Turning in Your Solution

You must turn in your code using GitHub Classroom. Turn in only your program; do not turn in any testcases.

In addition, you must turn in your report (see details above). Part of your score for Project 1 will come from this report.

**Make sure that your code is actually on GitHub before the deadline.** This will require that you add the file **and** then push it to GitHub. (You can confirm that you have uploaded the files correctly by viewing your repo through the GitHub website.)