

# Homework #2 – Assembly Programming

## Due Tuesday, Feb 15 at 5:00pm

---

**You must do all work individually. You may not look at each others' code, you may not help each other debug, etc. Invariably some students think that they will outsmart the tool for detecting cheating, and they end up receiving a zero on the entire assignment and a referral to the Office of Student Conduct.**

All submitted code will be tested for suspicious similarities to other code, and the test will uncover cheating, even if it is “hidden” (by reordering code, by renaming variables, etc.).

**You will be graded strictly on what you submit and when you submit it. Please double-check what you have uploaded to Gradescope.**

**IMPORTANT NOTE 1:** For programming assignments, Gradescope allows you to “activate” older submissions, but we will grade your latest submission. You may not make a submission after the deadline and then activate an older, on-time, submission if your late submission does not improve your score. You’ve been provided with a (non-comprehensive) test kit which you must use to test your code locally before submitting to Gradescope. Gradescope is meant for submission and not testing. You will be penalized for late submissions even if they are identical to older, on-time submissions.

**IMPORTANT NOTE 2:** Lost or corrupted files are NOT a valid reason for an extension, so please back up your work frequently! The Unix container you use is on a file system that should be reliable, but you are still responsible for making additional backups. (This is an important lesson beyond just this class – always back up your work frequently.) You can copy files from your Unix container (or your personal laptop or wherever) to Duke’s Box storage at box.duke.edu. If you’re comfortable with git, you can also backup to git, **as long as you ensure that your repository is private.**<sup>1</sup> You can use other backup services, too, but the responsibility is on you to backup your work.

### Directions:

- For short-answer questions, submit your answers in PDF format as a file called <NetID>-hw2.pdf (e.g., jd123-hw2.pdf). **Word documents will not be accepted.**
- For programming questions, submit your source file using the filename specified in the question.
- **You must submit your work electronically via Gradescope.** Submission consists of 4 files: <NetID>-hw2.pdf, mersennes.s, recurse.s, covidtracker.s.

---

<sup>1</sup> Please be extremely careful about repo privacy! If your repo is not private, it is accessible by others and is thus academic misconduct.

## MIPS Instruction Set

- 1) [5 points] What MIPS instruction is this? 0010 0011 0010 0010 0010 0001 0000 1000
- 2) [5] What is the binary representation of this instruction? lw \$r2, -8(\$r12)

## Compiling C Code to MIPS Assembly

3) [10 points total] In Homework 1 Q3 (“Compiling and Testing C Code”), you observed how changing the level of compiler optimization altered execution time. In this question, we will examine how that works.

The Unix container used in Homework 1 is a (virtual) PC based on the Intel x86 64-bit architecture, so the compiler generated instructions in that ISA for that CPU. In this question, we want to examine the resulting assembly language code, but we aren’t learning Intel x86 assembly language, so we’ll need a compiler that produces MIPS code instead. There’s a great web-based tool for testing various compilers’ output to various architectures, including MIPS: [Compiler Explorer](#). This web-based tool can act a front end to a g++ compiler which has been set to produce MIPS code. Further, it’s been set up to show us the assembly language code (.s file) rather than build an executable binary.

A small piece of the program from Homework 1, **prog\_part.c**, is linked on Sakai->Resource->Homework Resources. [This hyperlink](#) will take you to a view in Compiler Explorer where this code is being compiled to assembly language with optimization disabled (-O0) and set to maximum (-O3). Locate the `get_random()` function in the two versions of the code to answer the following questions:

- (a) [1] How many total instructions (not dot-prefixed directives like `.frame`) are in each of the two versions? Please include NOP (“no op”) instructions.
- (b) [1] How many `lw` and `sw` instructions were in each of the two versions?
- (c) [1] Which version of the code uses more registers?
- (d) [7] Based on the above, what are some general strategies you suspect the optimizer takes to improve performance?

## Assembly Language Programming in MIPS

### Directions and Rules

For the MIPS programming questions, use the QtSpim simulator that you used in Recitation #4. Please note that the TAs will be grading your assembly programs using QtSpim. If you use some other MIPS simulator (e.g., MARS), there is NO guarantee that what works on that simulator will also work on QtSpim. We will grade strictly based on what works when your program runs on QtSpim.

**Note:** Do not try to use Compiler Explorer (from question 3) to “automate” the programming questions below. In addition to being academically dishonest, it also won’t work very well for technical reasons that are beyond the current scope of the class.

## Testing Your Code

As in Homework #1, we provide a self-test tool to help you validate your code. The tool is provided in Sakai->Resources->Homework 2-> `homework2-kit.tgz`. You can move your `.s` files into the tool directory and run “`./hw2test.py`” as in HW1, which will give you some information on which test cases you pass/fail. As always, do all of your work on the Unix container machines to avoid unexpected errors.

Before testing on the Unix container machines, you can also manually provide input and visually check the output of your program in the QtSpim console against the expected output for a given test. Test cases and expected value are provided in `/tests` directory. Note that some test cases from HW1 no longer apply, and have been eliminated.

Each of your programs should prompt for input and display output via the **QtSpim Console window**.

If you want to execute a manual test, type in the **Input** listed in the tables associated with each program when prompted. After you have run your program, your program's output should match the **expected output** from the file indicated. For problem (c), the input comes from the file listed in the **Input file** column. Each line in the file should be typed in individually per prompt as instructed in problem (c).

## Similarity to Programming in Homework #1

These programming questions are almost the same as those from Homework 1 with the following key differences:

- In HW1, input came from command line arguments. In this assignment, input is typed into the console.
- Because the program is now prompting for input interactively, your program will output prompts before reading values. We intend to use an automated tool to assist with grading, so **please end all your user prompts in a colon**. (e.g. “Please enter an integer:”). The autograder will ignore the text of the prompt itself, so you can print any text you want before the colon.
- Question 6 has a difference that is explained in its specification.

## The Programming Tasks

4) [20] Write a MIPS program called `mersenne.s` that first prints a prompt (“Please enter an integer:”). The user types in an integer (call it  $n$ ) and hits return, and then your program prints the  $n^{\text{th}}$  Mersenne number. (This is just like in Homework #1, when you coded this in C). You may assume that the input will be non-negative and that the result will fit in 32 bits.


You will upload `mersenne.s` to Gradescope.

5) [40] Write a MIPS program called `recurse.s` that first prints a prompt ("Please enter an integer: "). The user types in an integer that is greater than zero (call it  $n$ ) and hits return. Your program computes and prints  $f(n)$ , where  $f(n) = 3*n + [2*f(n-1)] - 2$ . The base case is  $f(0) = -2$ . Your code must be recursive, and it must follow proper MIPS calling conventions. **The key aspect of this program is to teach you how to obey calling conventions → code that is not recursive or that does not follow MIPS calling conventions will be severely penalized!**

You will upload `recurse.s` to Gradescope.

6) [50] Write a MIPS program called `covidtracker.s` that is similar to the C program you wrote in Homework #1. However, instead of reading in a file, your assembly program will read in lines of input as strings. That is, each line will be read in as its own input (using `spim`'s `syscall` support for reading in inputs of different formats). The input is in the following format. The input is a series of patients and who they infected, not including Patient Zero, the Blue Devil. To simplify life, each patient can infect no more than two other people. **UNLIKE in Homework #1, in order to simplify the programming, each patient is a pair of lines: the patient's name first and then who infected him/her.** For clarity, it is a good idea to have two different prompt messages for the two strings you will read (e.g., "Please enter patient" and "Please enter infector"), because you will only read one string at a time. This is for readability, but does not affect correctness of your programs. After the last patient in the list, the last line of the input is the string "DONE".

For example:



```
Smith
BlueDevil
Jones
Smith
BuckyBadger
Jones
Doe
Smith
Johnson
BlueDevil
White
Doe
DONE
```

I guarantee that (a) no patient will appear in the list before the patient who infects him/her, and (b) no patient will be infected more than once, (c) no patients have the same name, and (d) every name is less than 30 characters.

Your program should output a list (data structure hint!) of all patients, including BlueDevil, that is ordered alphabetically. Each line of the output is the patient's name, followed by the names of the patient(s) they infected, if any, in alphabetical order. For the example input above, here is the output:

```
BlueDevil Johnson Smith
BuckyBadger
Doe White
Johnson
Jones BuckyBadger
Smith Doe Jones
White
```

**IMPORTANT:** There is no constraint on the number of patients, so you may not just allocate space for, say, 10 patients; you must accommodate an arbitrary number of patients. You must allocate space on the heap for this data. **Code that does not accommodate an arbitrary number of patients will be penalized!** Furthermore, you may not read all of the input lines into a buffer to first find out how many patients there are and \*then\* do a single dynamic allocation of heap space; instead, you must dynamically allocate memory on-the-fly as you read the lines. To perform dynamic allocation in MIPS assembly, I recommend looking at: [http://chortle.ccsu.edu/assemblytutorial/Chapter-33/ass33\\_4.html](http://chortle.ccsu.edu/assemblytutorial/Chapter-33/ass33_4.html)

Unlike in homework #1, you do NOT have to free memory that you dynamically allocate.

You will upload covidtracker.s to Gradescope.