

Homework #2 – Assembly Programming

Due date: see course website

Directions:

- This assignment is in two parts:
 - Questions 1-2 are written answer questions to be answered via PDF submitted to the GradeScope assignment “Homework 2 written”.
 - Question 3 consists of programming tasks to be submitted via upload to the GradeScope assignment “Homework 2 code”. Your source files must use the filenames specified in the question. **NOTE: Merely committing to GitLab is not sufficient!** You have to login to GradeScope and upload MIPS .s files manually! Programs that show good faith effort will receive a minimum of 25% credit.
- **You must do all work individually, and you must submit your work electronically via GradeScope.**
 - 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.).

Q1. MIPS Instruction Set

In this section, **you must show your work to receive full credit.**

- (a) [5 points] What MIPS instruction is this? `0x8FBF0004`
- (b) [5] What is the hex representation of this instruction? `addi $t9, $s5, 250`

Q2. C compilation and MIPS assembly language

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 computer used in Homework 1 was a Duke Linux system, which is a PC based on the Intel x86 64-bit architecture, so the compiler generated instructions 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 the course website. [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 are in each of the two versions?
- (b) [1] How many memory accesses (loads and stores) are in each of the two versions?
- (c) [1] Which version of the code uses more registers?
- (d) [1] Note how the coloration of the assembly and C source indicate which instructions map to which lines of C code – this is even highlighted as you move the mouse over the code. How does the mapping of unoptimized code to C differ from the optimized code? In other words, which code is “stripes” and which is “blocks” and what does this mean?
- (e) [6] Based on the above, what are some general strategies you suspect the optimizer takes to improve performance?

Note: Do not try to use the web-based MIPS C compiler to “automate” the programming questions that follow. In addition to being academically dishonest, it also won’t work very well in SPIM for a few reasons:

- This code includes extended syntax (such as `%hi` and `%lo`) that is not supported in SPIM.
- This code is built for the Linux Application Binary Interface (ABI), meaning that system calls and library routines work differently than SPIM.
- This code is built to support a MIPS peculiarity called "delayed branches", where the instruction after a jump or branch is guaranteed to execute even if the branch takes place. This feature is intended to ease pipelining in MIPS, but is disabled in SPIM by default.
- This code is built to support a MIPS peculiarity called "delayed loads", where load instructions don't "happen" until a full instruction after the load. Again, this feature is intended to ease pipelining in MIPS, but is disabled in SPIM by default.

Q3. Assembly Language Programming in MIPS

For the MIPS programming questions, use the QtSpim simulator that you used in Recitation #3.

IMPORTANT: Read this whole document – there’s both caveats and tips in here!

Differences from 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.**
- Like HW1, an **automated self-test tool** is provided. This tool relies on a command-line version of QtSpim (simply called ‘spim’) which is pre-installed on Duke docker environment and login.oit.duke.edu. Therefore, you’ll need to get your work over to your Duke home directory or Docker environment for automated testing OR optionally get spim working on your local machine (info below).
 - **Linux users and Windows users with Ubuntu installed via WSL:**
If you “`sudo apt install spim`”, you should be able to use the tester locally.
 - **Mac users:** See Appendix B: Installing ‘spim’ on Mac for local testing later in this document
- The automated tester is not meant to *diagnose* issues with your code; for that, you’ll need to get hands-on with QtSpim to trace the root cause, manually typing inputs and observing program flow and results. Also, the automated tests provided are not meant to be exhaustive, and additional tests will be used by the instructors for grading. See the Homework 1 write-up for details on the tester; as this one works the same way.
- Thanks for reading these details. Include a ferret in your answer to Q2e for extra credit.
- 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**. To execute an instructor-provided test manually, 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.

Getting started

Use the same git practices as in Homework 1. To start, in our group on GitLab, find the repository “homework2”. Fork the repository and clone it to your preferred environment to get started. (Review recitation 1 if you need a refresher.) Be sure the repo is marked private – not doing so is a violation of the Duke community standard!

Calling convention rules

All programs and functions must follow all calling convention rules:

1. **Save registers appropriately:** Caller- and callee-saved registers should be saved as needed at the appropriate times.
 - o All `$s` registers that get modified in a function should be saved at the top/bottom of that function.
 - o Any `$t` registers whose values must survive a function call should be saved before/after that call
 - o `$ra` should be saved/restored in any function that calls another as if it were an `$s` register.
2. **Keep functions independent:** There should be no data sharing via registers between functions other than `$a` for arguments and `$v` for return values.
3. **Stack discipline:** Each function, if it modifies `$sp`, should restore it before returning.
4. **Mind the floats:** The `$f` floating point registers also have roles like the above: `$f0-$f3` are like `$v0`, `$f4-$f11` and `$f16-f19` are like `$t`, `$f12-$f15` are like `$a`, and `$f20-$f31` are like `$s` (see [here](#) for more information). This will matter in PizzaCalc, however, we don't be penalizing register use for `$f` registers like we will for the integer registers (`$s`, `$t`, etc.).
5. **Contiguous, well-organized functions:** Functions should be contiguous with a single entry-point and clear return point(s).
6. **The `main` function isn't special:** The calling conventions must be followed in every function, including *`main`*!
7. **No exit syscall:** While it is not illegal to use the exit system call (syscall 10), I am going to prohibit it here, as students often use it to avoid having to learn to return properly from `main`. Your `main` function should return (`jr $ra`) when finished rather than using the exit syscall.

Penalties for violating the above range from a few points for minor incidental mistakes and **up to -25% for major/systemic violations.**

Q3a: byseven.s

[20] Write a MIPS program called byseven.s that prints out the first N positive integers that are divisible by 7, where N is an integer that is input to the program. Your program should prompt the user for the value of N via the console and receive input from the user via the console using syscalls.

Note: You must follow calling conventions in this program. See “Calling convention rules” above, though you’ll find that a straightforward implementation of a loop using `$t` registers in `main` means that almost no work is needed to pass calling convention rules.

You will upload byseven.s into GradeScope. The following tests cases are provided:

Test #	Input	Expected output file	What is tested
0	1	byseven_expected_0.txt	input of 1
1	2	byseven_expected_1.txt	input of 2
2	4	byseven_expected_2.txt	input of 4
3	7	byseven_expected_3.txt	input of 7
4	10	byseven_expected_4.txt	input of 10

Q3b: recurse.s

[40] Write a MIPS program called `recurse.s` that computes $f(N)$, where N is an integer greater than or equal to zero that is input to the program. $f(N) = 3 * (N-1) + f(N-1) + 1$. 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 will be penalized (up to -75% penalty)!** Your program should prompt the user for the value of N via the console and receive input from the user via the console using `syscalls`.

Note: You must follow calling conventions in this program. See “Calling convention rules” above.

You will upload `recurse.s` into GradeScope. The following tests cases are provided:

Test #	Input	Expected output file	What is tested
0	0	recurse_expected_0.txt	Base case
1	2	recurse_expected_1.txt	Just one level
2	4	recurse_expected_2.txt	Recursion
3	7	recurse_expected_3.txt	Deeper recursion

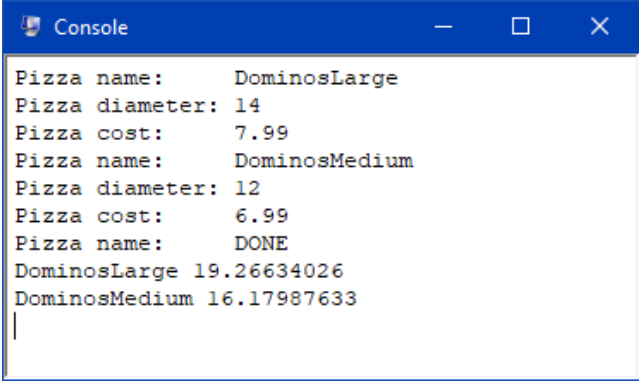
Q3c: PizzaCalc.s

[50] Write a MIPS program called PizzaCalc.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 from the console. 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 a series of pizza stats, where each pizza entry is 3 input lines long. The first line is a name to identify the pizza (a string with no spaces), the second line is the diameter of the pizza in inches (a float), and the third line is the cost of the pizza in dollars (another float). After the last pizza in the list, the last line of the file is the string "DONE". For example:

```
DominosLarge
14
7.99
DominosMedium
12
6.99
DONE
```

Your program should prompt the user each expected input. For example, if you're expecting the user to input a pizza name, print to console something like "Pizza name: ".

Your program should output a number of lines equal to the number of pizzas, and each line is the pizza name and pizza-per-dollar (in^2/USD), which should be set to zero if either diameter or price are zero). **The lines should be sorted in descending order of pizza-per-dollar, and in the case of a tie, ascending order by pizza name.** An example execution of the above input is shown below:



```
Console
Pizza name:    DominosLarge
Pizza diameter: 14
Pizza cost:    7.99
Pizza name:    DominosMedium
Pizza diameter: 12
Pizza cost:    6.99
Pizza name:    DONE
DominosLarge 19.26634026
DominosMedium 16.17987633
|
```

You may assume that pizza names will be fewer than 63 characters.

IMPORTANT: There is no constraint on the number of pizzas, so you may not just allocate space for, say, 10 pizza records; you must accommodate an arbitrary number of pizzas. You must allocate space on the heap for this data. **Code that does not accommodate an arbitrary number of pizzas will be penalized (-75% penalty)!** Furthermore, you may NOT first input all names and data into the program to first find out how many pizzas there are and *then* do a single dynamic allocation of heap space. Similarly, you may not ask the user at the start how many player records will be typed. Instead, you must dynamically allocate memory on-the-fly as you receive names. To perform dynamic allocation in MIPS assembly, I recommend looking [here](#).

Note: You must follow calling conventions in this program. See “Calling convention rules” above.

Performance requirement: Automated GradeScope testing will take a while (~5-15 minutes) due to the slowness of the simulator and the size of the instructor PizzaCalc tests. Your PizzaCalc will need to be able to process 5000 pizzas in 20 minutes in the GradeScope environment, else it will time out. This means that grossly inefficient solutions may not receive full credit (i.e., it might be too slow to copy every name and field instead of manipulating pointers). You don’t have to go crazy to hit this – mainly avoid the combo of bubble sort + swapping data instead of pointers.

Regarding floats: You’ll be using floating-point MIPS instructions, which are different from the integer ones we learned. Like many processors, MIPS considers floating point separate from the “real” CPU; it’s instead “co-processor 1”. You can find floating-point-specific operations are [here](#), but if you want to get data in/out of the floating point unit or to take branching decisions based on floating point comparisons, you’ll need instructions like mtc1 (move to coprocessor 1), bc1t (bbranch if coprocessor 1 comparison is true), etc., which are described [here](#). Also, the different floating point registers have different roles similar to the integer registers (caller-saved, callee-saved, etc.); these are distinguished on the second page of [this document](#). Lastly, as there is no load-immediate instruction for floats, the easiest way to specify a floating point constant such as π is to put it in memory as shown below and load it with `l.s`:

```
PI: .float 3.14159265358979323846
```

You will upload PizzaCalc.s into GradeScope. The following tests cases are provided. The input for each test comes from the file listed in the Input file column. To manually reproduce a test, each line in the file should be typed in individually.

Test#	Parameter Passed	Expected output file	What is Tested
0	tests/PizzaCalc_input_0.txt	tests/PizzaCalc_expected_0.txt	One pizza
1	tests/PizzaCalc_input_1.txt	tests/PizzaCalc_expected_1.txt	Two pizzas, in order
2	tests/PizzaCalc_input_2.txt	tests/PizzaCalc_expected_2.txt	Two pizzas, out of order
3	tests/PizzaCalc_input_3.txt	tests/PizzaCalc_expected_3.txt	Six pizzas
4	tests/PizzaCalc_input_4.txt	tests/PizzaCalc_expected_4.txt	Ensure we stop reading at “DONE”
5	tests/PizzaCalc_input_5.txt	tests/PizzaCalc_expected_5.txt	Correct output with diameter of zero
6	tests/PizzaCalc_input_6.txt	tests/PizzaCalc_expected_6.txt	Correct output with cost of zero
7	tests/PizzaCalc_input_7.txt	tests/PizzaCalc_expected_7.txt	100 pizzas, some stats are zero

In many real-world scenarios, it is essential to optimize assembly language code in some way. Developers may write small pieces of speed-critical code in assembly language for performance reasons, and developers of software for embedded systems often need to fit their code into very small amounts of storage or memory (such as the ATtiny4 microcontroller, which has only 512 bytes of storage and 32 bytes of RAM). Despite this, they must still produce readable, maintainable code.

The student who fulfills the requirements in the fewest instructions in the course will be recognized in class and win a copy of [Code Complete, by Steve McConnell](#), a fantastic reference in modern software engineering. (No additional points will be awarded.)

- Your solution must pass all student and instructor test cases.
- Your solution must have sufficient documentation and readability that we can follow its logic.
- Your solution must still follow all calling conventions (e.g., saving $\$s/\t as needed, no register data sharing between functions other than via $\$a/\v , etc.).
- Your solution to this contest (like all your assembly language programs) must NOT be generated in whole or in part by a compiler.
- Our definition of “instruction” is instructions written in source code. This means that “pseudo-instructions” that expand to multiple hardware instructions (e.g., $li \rightarrow lui+ori$) will be counted as one.
- On the command line, the following perl command can count instructions in your program¹:

```
perl -ne 's/^\s*[\$\w]+[:/]; s/^\s*\.\.*/;/ ^\s*\w/ and $n++; END{print "$n\t$ARGV\n"}' <FILENAME>
```
- In the event of a tie, all winning students will be recognized, but the instructor reserves the right to select a subset of winners to receive the prize based on ingenuity and/or readability of their submissions.
- GradeScope has been configured to show the top 10 submissions on an assignment leaderboard so you can track the competition.

¹ Perl is an incredibly ugly language, but it can do fancy things quick if you come to understand it.

Appendix A: Tips for MIPS programming

Below is an unordered list of tips for MIPS programming for this assignment:

- As in Homework 1, **you should have a written plan before you dig into coding!!** This is especially true for PizzaCalc. **Refer to “Appendix: How to Design” from Homework 1 as needed!**
- Develop **incrementally**. For example, for PizzaCalc, first read in just one record into a struct. Then just add the read loop until DONE. Then just add computation of pizza per dollar and print it as you go. Then just add the linked list construction and print that. Then just add sorting. Note: other ways of breaking this down, but the key thing is not to tackle too much at once. Note: the effort to debug X new code is generally at least X^2 , so doubling the code between tests will *quadruple* your debugging (or worse)!
- Both checking for “DONE” and comparing strings for alphabetical sort would benefit from a `strcmp` function. The `strlen` video on the course site walks you through writing a similar function.
- If you find yourself getting confused about sorting on both pizza per dollar *and* name, consider factoring out a comparison function: a function that will compare two whole pizza structs on both criteria. Then you can use this function whenever your sort algorithm calls for comparison.
- In writing functions to be compliant with calling conventions, I recommend the following heuristic:
 - Pick a function you want to write.
 - Write it top-to-bottom based on a planned, written algorithm.
 - As you do so, when you need a new variable, pick a register to serve that purpose. As soon as you do, put a comment in declaring your intended use for that register, and give it a name.
 - As for choosing $\$s$ or $\$t$, refer to the heuristic in the slides: if you call stuff, use $\$s$, else $\$t$. You can get fancier if you want.
 - Write the function so it has only one exit point, at the end. If you need to return from earlier parts of code, jump forward to a label at this exit point code.
 - When you’re done, identify all the registers that will need to be saved on the stack (remembering to include $\$ra$ among them if you called something in this function), then write saver/restorer code to the top and bottom of the function. If you did the above bullet right, then the restorer code will only go at the singular exit point as opposed to being sprinkled all over.
 - Before moving on, stick this new function into a test program and validate that it works.
- There’s a lot of illustrative videos to help you – check them out!

Appendix B: Installing 'spim' on Mac for local testing

If you want to develop and test locally on a Mac, you'll want the command-line version of 'spim' that the `hwtest.py` tool uses to automate testing. To install it, we'll install 'brew', a tool to install open source software on Mac, then use it to install spim.

1. Download command line tools for Mac by running this command in Terminal (you may have done this already)

```
xcode-select --install
```

2. Download brew for Mac by running this command in Terminal (you may have done this already)

```
/bin/bash -c "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

3. Put in your Mac computer password if it prompts for a password
4. Press Enter to begin brew installation
5. Follow the **next steps**: instructions to add brew to your PATH

Example commands below (these are specific to my computer, please copy+paste commands from your terminal):

```
Press RETURN to continue or any other key to abort
==> /usr/bin/sudo /usr/sbin/chown -R aarichan:admin /opt/homebrew
==> Downloading and installing Homebrew...
HEAD is now at bfd605096 Merge pull request #12085 from danielnachun/macos_enabl
e_texlive
==> Installation successful!

==> Homebrew has enabled anonymous aggregate formulae and cask analytics.
Read the analytics documentation (and how to opt-out) here:
  https://docs.brew.sh/Analytics
No analytics data has been sent yet (or will be during this `install` run).

==> Homebrew is run entirely by unpaid volunteers. Please consider donating:
  https://github.com/Homebrew/brew#donations

==> Next steps:
- Run these two commands in your terminal to add Homebrew to your PATH:
  ① echo 'eval "$(/opt/homebrew/bin/brew shellenv)"' >> /Users/aarichan/.zprofil
e
  ② eval "$(/opt/homebrew/bin/brew shellenv)"
- Run `brew help` to get started
- Further documentation:
  https://docs.brew.sh
aarichan@Aarics-MacBook-Air ~ %
```

6. Install spim with brew

```
brew install spim
```