Homework #4 – Assembly Programming

Due Friday, Feb 24, at 5:00pm Eastern Time

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 4 written".
  - Question 3 consists of programming tasks to be submitted via upload to the GradeScope assignment "Homework 4 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 <u>show your work</u> to receive full credit.**

(a) [5 points] What MIPS instruction is represented by this 32-bit string (represented in hex)? 0x8ec80014

(b) [5] What is the hex representation of this instruction? `andi $s7, $t2, 6`

Q2. C compilation and MIPS assembly language

In Homework 3 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 3 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 3, prog_part.c, is available on Sakai. **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 #4.

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

Differences from Homework 3

These programming questions are almost the same as those from Homework 3 with the following key differences:

- In HW3, 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 HW3, 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 3 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 HW3 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 3. To start, in our group on GitLab, find the repository "homework4". 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.
    - All `$s` registers that get modified in a function should be saved at the top/bottom of that function.
    - Any `$t` registers whose values must survive a function call should be saved before/after that call
    - `$ra` should be saved/restored in any function that calls another as if it were an `$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. **Contiguous, well-organized functions:** Functions should be contiguous with a single entry-point and clear return point(s).
5. **The `main` function isn't special:** The calling conventions must be followed in every function, *including* ***`main`****!*
6. **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: by5or6.s

[10] Write a MIPS program called by5or6.s that prints out the first N positive integers that are divisible by 5 or 6, 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 by5or6.s into GradeScope. The following tests cases are provided:

Test #	Input	Expected output file	What is tested
0	1	by5or6_expected_0.txt	input of 1
1	2	by5or6_expected_1.txt	input of 2
2	4	by5or6_expected_2.txt	input of 4
3	7	by5or6_expected_3.txt	input of 7
4	10	by5or6_expected_4.txt	input of 10

Q3b: recurse.s

[20] Write a MIPS program called recurse.s that computes f(N), where N is an integer greater than zero that is input to the program. f(N) = 2*N+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: HoopsRank.s

[50] Write a MIPS program called HoopsRank.s that is similar to the C program you wrote in Homework #3. 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 team stats, where each team entry is 4 input lines long. The first line is a name to identify the team (a string with no spaces), the second line is the average number of points the team scores per game (an int), the third line is the average number of points the team gives up per game (another int), and the fourth line is the average rebounding differential for the team (another int, but it could be negative) After the last team in the list, the last line of the file is the string "DONE". For example:

```
Duke
88
73
5
Carolina
76
77
-3
DONE
```

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

Your program should output a number of lines equal to the number of teams, and each line is the team name and the metric that is computed as:

[(average points scored) – (average points given up)] + (average rebounding differential).

The lines should be sorted in *descending* order based on this metric, and you must write your own sorting function (you can't just use the qsort library function). Teams with equal metrics should be sorted alphabetically (e.g. based on the strcmp function). For example:

```
Duke 20
Carolina -4
OhioState -11
Tech -11
```

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

Empty files and files of the wrong format will not be fed to your program.

IMPORTANT: There is no constraint on the number of teams, so you may not just allocate space for, say, 10 team records; you must accommodate an arbitrary number of teams. You must allocate space on the heap for this data. **Code that does not accommodate an arbitrary number of teams will be penalized (-75% penalty)!** Furthermore, you may NOT first input all names and data into the program to first find

out how many teams there are and *then* do a single dynamic allocation of heap space. Similarly, you may not ask the user at the start how many teams will be typed. Instead, you must dynamically allocate memory on-the-fly as you receive team 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's HoopsRank tests. Your HoopsRank code will need to be able to process roughly 5000 teams 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.

You will upload HoopsRank.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	What is Tested
0	tests/HoopsRank_input_0.txt	One team
1	tests/HoopsRank_input_1.txt	Two teams, in order
2	tests/HoopsRank_input_2.txt	Two teams, out of order
3	tests/HoopsRank_input_3.txt	Six teams
4	tests/HoopsRank_input_4.txt	Ensure we stop reading at "DONE"
5	tests/HoopsRank_input_7.txt	100 teams

Appendix A: Tips for MIPS programming

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

- As in Homework 3, **you should have a written plan before you dig into coding!!** This is especially true for HoopsRank.
- Develop **incrementally**. For example, for HoopsRank, first read in just one record into a struct. Then just add the read loop until DONE. Then just add computation of the team metric 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.
- If you find yourself getting confused about sorting on both team metric *and* name, consider factoring out a comparison function: a function that will compare two whole team 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.

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
   ```