CMSC 216 Project 5 Fall 2021

MIPS Assembly Programming **Due date posted on Piazza**

1 Overview

In this project you write MIPS assembly code that is equivalent to provided C functions.

The objectives are to gain an understanding of how a program in a high-level language is translated to assembly and executed by hardware.

IMPORTANT: Implement this project individually. Do not work with others.

IMPORTANT: Do not use a compiler to generate MIPS code.

2 Academic Integrity

Please **carefully read** the academic honesty section of the course syllabus. We take academic integrity matters seriously. Please do not post assignment solutions online (eg, Chegg, github) where others can see your work. This can lead to you being reported to the Office of Student Conduct.

3 Grading Criteria

Your project grade will be determined as follows:

Results of public tests	40%
Results of secret tests	40%
Satisfying MIPS conventions	10%
Code style	10%

The only files we grade are your .s source files.

The good faith attempt information will be posted on the class web page.

3.1 Project Files

This project description can be found in the 216public project_descriptions directory. Copy the folder project5 in the 216public projects directory to your 216 directory. The files in the distribution, other than `.submit`, are:

1. C "function" files which you have to translate to assembly:
   `fibonacci.c`, `is_palindrome.c`, `isqrt.c`, `reverse_prefix_sum.c`.
2. C driver files to run the above C function files:
   `fibonacci_driver.c`, `is_palindrome_driver.c`, `isqrt_driver.c`, `reverse_prefix_sum_driver.c`.
3. Assembly driver files corresponding to the C driver files:
   `fibonacci_driver.s`, `is_palindrome_driver.s`, `isqrt_driver.s`, `reverse_prefix_sum_driver.s`.
4. Test scripts to run your assembly code:
   `fibonacci_test1`, `is_palindrome_test1`, `isqrt_test1`, `reverse_prefix_sum_test1`.
5. Output files of the tests (these are generated from the C files):
   `fibonacci_test1.output`, `is_palindrome_test1.output`, `isqrt_test1.output`, `reverse_prefix_sum_test1.out`

4 Specifications

Below, we use *f* to stand for `fibonacci`, `is_palindrome`, `isqrt`, or `reverse_prefix_sum`. Each C function file *f*.c contains a C function of the same base name, *f*. For each function *f*, you must submit an assembly file *f*.s containing an assembly function *f* **equivalent** to the C function *f*, which means the assembly function

- the **same input-output behavior** (ie, produces the same output for any input), and
- the **same internal operation** (ie, for each C statement there is corresponding assembly code).

For example, the C function file `isqrt.c` has the function `isqrt()`. You must create an assembly file `isqrt.s` that contains an assembly function `isqrt` equivalent to the C function. Similarly, you must create assembly files `is_palindrome.s`, `fibonacci.s`, and `reverse_prefix_sum.s`.

To generate the test output of C function *f*, compile *f*.c and *f*`_driver.c` and run the executable. For example,

```
% gcc isqrt_driver.c isqrt.c
% a.out
```

would generate the contents of file `isqrt_test1.output`.

After you write your assembly function *f* in file *f*.s, you execute it by appending the file to the driver file *f*`_driver.s` (eg, "`cat` *f*`_driver.s` *f*.s `>` *f*`_prog.s`") and executing file *f*`_prog.s` in qtspim or in spim. Script *f*.`_test1` does exactly this. Hence, if your `isqrt.s` is correct, then the following should print nothing.

```
% isqrt_test1 | diff - isqrt_test1.output
```

5 Requirements

1. Your assembly function must be equivalent, in the sense defined above, to the C function. In particular, your `fibonacci` and `reverse_prefix_sum` functions **must** be recursive, otherwise you get **no credit**. (So the body of your `fibonacci` function must execute a `jal fibonacci` instruction. Similarly for your function `reverse_prefix_sum`.)
2. Your `is_palindrome.s` file must also have a `strlen` function. Use the one from your own assembly exercise.
3. Don't change the C code or assembly drivers. **The public tests are in the drivers**.
4. Your functions should not have any labels that begin with `main`; the drivers use such labels.
5. Obey the callee-save and caller-save conventions. The secret tests in the submit server may check for this.
6. Do not use global variables except for string literals.
7. Your code must have a comment at the beginning with your name, university ID number, and UMD Directory ID (ie, your username on Grace).
8. (Style) Your code must be thoroughly and carefully commented. Assembly language can easily become unreadable without proper documentation.

   A "per-line comment", which is a comment at the right of each line of code, is usually necessary. Your comments should be descriptive of the intent, not of the action; eg, "stash n for later use" is an ok comment, whereas "move t0 into a0" is not. See our examples.
9. (Style) A description of the variable that each register represents is useful. See our examples.
10. (Style) Use appropriate whitespace, especially between blocks of instructions that perform different tasks.
11. (Style) Reasonable and consistent indentation is required. The convention is to indent all instructions by a few spaces or one "tab", to not indent labels, and to align per-line comments starting at the same column. Our examples are generally formatted this way.

    You can have lines that exceed have 80 characters, typically because of per-line comments.
12. (MIPS conventions) Follow the MIPS conventions for accessing arguments and variables on the stack (eg, offset from frame pointer).