INF2C Computer Systems

Coursework 1

MIPS Assembly Language Programming

Deadline: Wed, 24 Oct (Week 6), 16:00

Instructor: Boris Grot

TA: Siavash Katebzadeh

The aim of this assignment is to introduce you to writing simple programs in MIPS assembly and C languages. The assignment asks you to write three MIPS programs and test them using the MARS IDE. For details on MARS, see Lab Exercise 1. The assignment also asks you to write C code and use C code versions of the programs as models for the MIPS versions.

This is the first of two assignments for the INF2C Computer Systems course. It is worth 50% of the coursework mark for INF2C-CS and 20% of the overall course mark.

Please bear in mind the guidelines on academic misconduct from Undergraduate year 2 student handbook available at http://web.inf.ed.ac.uk/infweb/student-services/ito/students/year2.

1 Preliminaries

One of the most common and simplest tools that can assist the user with writing is a spell checker. A spell checker is a program or function of a program which determines the correctness of the spelling of a given word based on the language set being used.

The main focus of this exercise is to implement a spell checker for a subset of the English language in C and MIPS.

1.1 Task A: Tokenizer

The first task is a warm-up exercise. It helps you get familiar with the basic structure of MIPS and C programs, and with using the MARS IDE.

In this task, an input file is used. The input file is a text document, consisting of alphabetic characters (`a-z` and `A-Z`), punctuation marks `,.!?` (i.e., comma, period, exclamation, question mark) and spaces in any order and combination (without any newlines). The input file is terminated by End-of-File (EOF). A valid input file does not include any other characters.

You are given a MIPS file named `tokenizer.s` which contains a skeleton of your program. This skeleton reads an input file named `input.txt` and stores its content into a `null`-terminated string named `content`. Your task is to break the `content` string into a list of *tokens* and output them, one token per line.

A token consists of one or more consecutive characters of exactly **one** of the following three different character types:

1. *Alphabetic characters*, which includes only alphabetic characters (`a-z` and `A-Z`)

   e.g. 'CompSys'

2. *Punctuation marks*, which includes only punctuation mark characters (`,.!?`)

   e.g. '?!.!?.?'

3. *Spaces*, which includes only space characters (`' '`)

   e.g. '   '

If the content of the input file is:

```
Let go your earthly tether. Enter the void. Empty, and become wind.
```

then the output of your program should look like:

```
Let

go

your

earthly

tether
.

Enter

the
```

```
void
...

Empty
,
and

become

wind
.
```

A good way to go about writing a MIPS program is to first write an equivalent C program. It is much easier and quicker to get all the control flow and data manipulations correct in C than in MIPS. Once the C code for a program is correct, one can translate it to an equivalent MIPS program statement-by-statement. To this end, a C version of the desired MIPS program is provided in the file `tokenizer.c`.

For convenience, the C program includes definitions of functions such as `read_string` and `print_char`, which mimic the behaviour of MARS system calls with the same names. Derive your MIPS program from this `tokenizer.c` C program.

Do not try optimizing your MIPS code. Aim to keep the correspondence with the C code as clear as possible. Use your C code as comments in the MIPS code to explain the structure. Then put additional comments to document the details of the MIPS implementation.

As a model for creating and commenting your MIPS code, have a look at the supplied file `hex.s` and the corresponding C program `hex.c`. These are versions of the `hexOut.s` program from the MIPS lab which converts an entered decimal number into hexadecimal.

1.2 Task B: Spell Checker

The goal of this task is to write a program that detects misspelled words in a string and marks them in the output. A word is not misspelled if it exists in the dictionary. All characters in the dictionary are lowercase. For the purposes of this exercise, spell checking will be case insensitive. Thus, 'tHen' is a correct spelling of a dictionary word 'then'.

This task requires an input file similar to the one in Task A. In addition to this input file, we also use a dictionary file. The dictionary file consists of English words in lowercase alphabetic characters (`a-z`) delimited by a newline (`\n`) and terminated by End-of-File (EOF). A valid dictionary does not include any other characters.

You are given a MIPS file named `spell_checker.s` that contains a skeleton of your program. This skeleton reads an input file named `input.txt` and stores its content into a null-terminated string named `content`. It also reads a dictionary file named `dictionary.txt` and stores its content into a null-terminated string named `dictionary`.

In this task, you will write a program in both C and MIPS, that in the first step, *tokenizes* the `content` string and in the second step, checks whether each *Alphabetic* token is correct; if not, the program marks it by putting underscores (_) at the beginning and end of the token. In the final step, the program outputs all of tokens.

If the content of a given input file is:

```
Everybody is speciall.Everybody.. Everybody is a HeRo, a luver, a fool,
a villain!! Everybody... Everybody has thier story to tell.
```

then the output of your program should look like:

```
Everybody is _speciall_.Everybody.. Everybody is a HeRo, a _luver_, a
fool, a villain!! Everybody... Everybody has _thier_ story to tell.
```

Approach this task by first writing an equivalent C program `spell_checker.c` and then translating this C program into a MIPS program `spell_checker.s`. Before translating, you should compile and test your C program. Ensure it is working correctly before starting on the MIPS code. To help you get started with the C program, an outline `spell_checker.c` is supplied. Furthermore, you can reuse the *Tokenizer* of Task A to extract the tokens from the `content` string. We recommend structuring the code into functions for ease of development and readability. See section 2.2 for further advice and requirements on MIPS code.

As in Task A, use the C code to comment your MIPS code, and put additional comments for MIPS-specific details. In your C program, your comments should focus on explaining the higher-level features of your program, so your comments in the MIPS code can mainly just concern themselves with the MIPS implementation details.

1.3 Task C: Punctuation Checker

The goal of this task is to extend Task B by adding basic punctuation checking to your program. In this task, in addition to spell checking (Task B), your program (`punctuation_checker.s`) also has to detect punctuation errors and inconsistencies in the `content` string and mark them in the output. The rules of correctness of *Punctuation* tokens are shown in Table 1. The rules are sorted according to their priority, with higher priority rules on top. These rules must be evaluated and executed sequentially against each token.

If the content of a given input file is:

```
Everybody is speciall.. Everybody. Everybody is a HeRo ,a luver, a fool,a
villain!! Everybody... Everybody has thier story to tell....
```

then the output of your program should look like:

```
Everybody is _speciall__.._ Everybody. Everybody is a HeRo _,_a _luver_,
a fool_,_a villain_!!_ Everybody... Everybody has _thier_ story
to tell_...._
```

While you are encouraged to write a C program to facilitate the MIPS implementation, you should **not** submit the C program for this task.

Punctuation Token	Correct	Incorrect	Example
has a *Space* token before it		✓	Hello .Is
has an *Alphabetic* token after it		✓	Hello.Is
has ellipsis ("...")	✓		now...
has only one punctuation mark	✓		yours?
has more than one punctuation marks		✓	Really?!
other		✓	

Table 1: Grammar checker rules

1.4 Assumptions and Restrictions on Program Inputs

Your programs in both tasks will be tested against dictionary and input files that adhere to the following rules:

- A dictionary file will contain a maximum of 10,000 words.
- A single word in the dictionary file will contain a maximum of 20 lowercase alphabetic characters.
- An input file will be a maximum of 2048 characters including spaces and without any newline character.
- An input file may contain alphabetic (`a-z`, `A-Z`), punctuation marks (`,.!?`) and space (`' '`) characters.

1.5 Restrictions on the use of C Library Functions

The only C library functions that can be used are `fgets` and `printf`, which are called within the functions `print_char`, `print_int`, `print_string` and `read_string` provided with the C files.

1.6 Output

For your convenience, the `output` function is provided in both C files. You can use it to output a string. Make sure your program does not print anything other than what is printed by the `output` function as it may interfere with automated marking. In MIPS programs, you should output exactly the same way as done in C programs.

2 Program Development and Testing

2.1 C

You can compile a C program written in a file called, for instance, prog.c at the command prompt on the DICE machines with the following command:

```
gcc -o prog prog.c
```

If compilation succeeded without any errors, it creates an executable `prog` which can then run by entering:

```
./prog
```

2.2 MIPS

You will need to choose what kind of storage to use in MIPS for all variables from the C code. In all tasks, we recommend you to store all tokens in the data segment. Use the `.space` directive to reserve space in the data segment for arrays, preceding it with an appropriate `.align` directive if the start of the space needs to be aligned to a word or other boundary.

In the MIPS implementation of Tasks B and C, you must call at least one function written by you. Function calls are optional in the implementation of Task A, and you are allowed to use the same function(s) in all three tasks, if you wish. You are also allowed to have multiple functions and nested function calls. Which functionality to abstract into a separate function(s) is entirely up to you. To reiterate, the only requirement is having at least one call to a user-defined function in Tasks B and C.

You must abide by the MIPS register convention. Remember that values of `$t*` registers are not guaranteed to be preserved across function call invocations (including both `syscall` and user-defined functions), whereas values of `$s*` registers will be preserved. However, do not use only `$s*` registers in your code; make use of `$t*` registers when appropriate.

Ultimately, you must ensure that your MIPS programs assemble and run without errors when using MARS. When testing your programs, we will run MARS from the command line. Please check that your MIPS programs run properly in this way before submitting them. For example, if the MARS JAR file is saved as `mars.jar` in the same directory as a MIPS program `prog.s`, running the following command at the command-line, assembles and then runs prog.s.

```
    java -jar mars.jar sm prog.s
```

Notes:

1) The `sm` option tells MARS to start running at the `main` label rather than with the first instruction in the code in `prog.s`. When running MARS with its IDE, marking the check-box for *Initialize Program Counter to global 'main' if defined* on the *Settings* menu achieves the same effect.

2) MARS supports a variety of pseudo-instructions, more than the ones that are described in the MIPS appendix of the Hennessy and Patterson book. In the past, we have often found errors and misunderstandings in student code relating to the inadvertent use of pseudo-instructions that are not documented in this appendix. For this reason, make sure you only use pseudo-instructions that are explicitly mentioned in the appendix.

3 Submission

Submit your work using the command

```
submit inf2c-cs cw1 tokenizer.s spell_checker.c spell_checker.s punctuation_checker.s
```

at a command-line prompt on a DICE machine. Unless there are special circumstances, **late submissions are not allowed**. Please consult the online undergraduate year 2 student handbook for further information on this.

You can submit more than once up until the submission deadline. All submissions are timestamped automatically. Identically named files will overwrite earlier submitted versions, so we will mark the latest submission that comes in before the deadline.

Warning: Unfortunately the `submit` command will technically allow you to submit late even if you submitted before the deadline. Don't do this! We can only retrieve the latest version, which means you will be penalized for submitting late.

For additional information about late penalties and extension requests, see the School web page below. Do **not** email any course staff directly about extension requests; you must follow the instructions on the web page. http://web.inf.ed.ac.uk/infweb/student-services/ito/admin/coursework-projects/late-coursework-extension-requests

4 Assessment

Your programs will be primarily judged according to correctness, completeness, and correct use of MIPS instructions, registers and function call convention.

In both C and MIPS programming, commenting a program and keeping it tidy is very important. Make sure that you comment the code throughout and format it neatly. A proportion of the marks will be allocated to these aspects of your code.

When editing your code, please make sure you do not use tab characters for indentation. Different editors and printing routines treat tab characters differently, and, if you use tabs, it is likely that your code will not look pretty when we come to mark it. If you use `emacs`, the command `(m-x)untabify` will remove all tab characters from the file in a buffer.

5 Similarity Checking and Academic Misconduct

You must submit your own work. Any code that is not written by you must be clearly identified and explained through comments at the top of your files. Failure to do so is plagiarism. Detailed guidelines on what constitutes plagiarism can be found at http://web.inf.ed.ac.uk/infweb/admin/policies/guidelines-plagiarism.

All submitted code is checked for similarity with other submissions using the MOSS system [1]. MOSS has been effective in the past at finding similarities. It is not fooled by name changes and reordering of code blocks.

[1] http://theory.stanford.edu/~aiken/moss/

6 Questions

If you have any questions about the assignment, please start by checking existing discussions on Piazza. If you can't find the answer to your question, start a new discussion – chances are, others have already encountered (and, possibly, solved) the same problem. The TA and the course instructor will also monitor Piazza and contribute to the discussions. You should also take advantage of the labs and the lab demonstrators who are there to answer your questions.

October 10, 2017