INF2C-CS Coursework 1

Instructors: Boris Grot (Course Organiser), Jianyi Cheng, Björn Franke
TAs: Kim Stonehouse, Dilina Wickramasinghe Dehigama, Shengda Zhu, Shyam Jesalpura

Important Dates

- Coursework 1 Release Date: October 11, 2024
- Coursework 1 Submission Deadline: October 25, 2024 (4pm)
- Coursework 1 Feedback Release Date: November 11, 2024

Overview

The aim of this assignment is to introduce you to writing simple programs in MIPS assembly and the C programming language. 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**.

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 using the C and MIPS assembly languages, respectively.

Learning Outcomes

This coursework is closely aligned with the learning outcomes of the INF2C-CS course as specified in its course descriptor (http://www.drps.ed.ac.uk/24-25/dpt/cxinfr08027.htm).

On completion of this course, the student will be able to:

1. *Describe the trade-offs in different binary representation systems.*
2. *Explain the principles of: instruction set architecture, digital logic design, cache hierarchy, virtual memory, I/O devices, exceptions and processor management.*
3. ***Demonstrate an understanding of how a high-level programming language (C) maps to the assembly code by converting a simple C program to MIPS assembly.***
4. *Sketch the design of a simple single- and multi-cycle processor and explain how it operates by combining the knowledge of the logic design basics with that of the MIPS instruction set architecture.*

Task Descriptions

The coursework comprises three tasks: As part of the first two tasks you will develop some useful “helper” functions such as a comparison function for string and a tokeniser, which divides a longer string into individual words (tokens). These routines will be reused in task 3, which is the actual spell checker.

Task 1: String Comparison

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

comparison function `strcmp` in MIPS assembly language. The `strcmp` will be invoked from test harness within a simple test program (comparator), enabling you to test your function with different strings passed on the command line.

A LITTLE BIT OF BACKGROUND

To compare two strings in C, you can use the `strcmp` function. This function becomes available after you include the `<string.h>` file. `strcmp` takes two strings and compares each character in sequence one by one.

Once it finds the first differing character, it returns:
- 1 if the character in the first string is greater than the one in the second.
- -1 if this first differing character is lesser in the first string.
- 0 if the strings are equal (no differing characters).

If it doesn't find any differing characters but one string ends before the other, that string is considered the lesser of the two. We'll see how this works with an example.

`strcmp` compares the corresponding characters from both strings based on their ASCII values, which are numeric codes that represent each character.

Therefore, if the ASCII value of the first differing character in the first string is less than the ASCII value of the corresponding character in the second string, `strcmp` returns a negative number. This indicates that the first string comes before the second string in the dictionary.

If the ASCII value of the first differing character in the first string is greater than the ASCII value of the corresponding character in the second string, `strcmp` returns a positive number. This indicates that the first string comes after the second string in the dictionary.

The prototype of the `strcmp` function looks like:

```
int strcmp (const char * s1, const char * s2);
```

- The first argument should be a pointer to a C-style string (C-style strings end with a '\0' character).
- The second argument should be a pointer to the string you want to compare with.
- The function returns 1, -1, or 0 depending on which string is "greater".

Let's see how this function works with an example:

```
#include <stdio.h>
#include <string.h>

void compare(const char* a, const char* b) {
  printf("strcmp(\"%s\", \"%s\") = %d\n", a, b, strcmp(a, b));
}

int main() {
  compare("abc", "abc");
  compare("abc", "ab");
  compare("ab", "abc");
  compare("abc", "bcd");
  compare("bc", "abc");

  return 0;
}
```

Program output:

```
strcmp("abc", "abc") = 0
strcmp("abc", "ab") = 1
```

```
strcmp("ab", "abc") = -1
strcmp("abc", "bcd") = -1
strcmp("bc", "abc") = 1
```

NULL-TERMINATED BYTE STRINGS

Character strings in C are represented as null-terminated byte strings (NTBS). A NTBS is a sequence of nonzero bytes followed by a byte with value zero (the terminating `null` character). Each byte in a byte string encodes one character of some character set. For example, the character array {'\x63','\x61','\x74','\0'} is an NTBS holding the string "cat" in ASCII encoding.

WRITING YOUR OWN VERSION IN C

Let's craft our own version of the `strcmp` function without using any built-in libraries to get a better grip on its inner workings. A possible unoptimised implementation of the `my_strcmp` function in C could look like this:

```
/*
 * Compare strings.
 */
int my_strcmp(const char *s1, const char *s2)
{
  int result = 0;

  /*
  Search for the first differing character until we reach the end of one
  of the strings. We don't need to check *s2 != '\0` since the first condition
  for the  loop to continue is that *s1 equals *s2. If *s1 equals *s2 and *s1
  isn't '\0`, that means *s2 isn't '\0` either
  */
  while (*s1 == *s2 && *s1 != '\0') {
    s1++;
    s2++;
  }

  /* if both strings have ended, they are equal */
  if (*s1 == '\0' && *s2 == '\0') {
    return 0;
  }

  /* here, we compare the first differing character */
  if (*s1 > *s2) {
    return 1;
  } else {
    return -1;
  }
}
```

To this end, a C version of the desired MIPS program is provided in the file `comparator.c`.

After compiling the comparator.c program, you can invoke the program like this

```
❯ ./comparator this that
my_strcmp("this", "that") = 1
```

Try this out with strings different to "this" and "that"!

For convenience you can also use the provided Makefile, i.e. a build script using the Make build system, for building and running your code.

```
❯ make run
gcc -c -o comparator.o comparator.c -I.
gcc -o comparator comparator.o -I.
./comparator this that
my_strcmp("this", "that") = 1
```

HOW TO GET STARTED

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.

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

Do not try optimising 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.

You are given a MIPS file named `comparator.s` which contains a skeleton of your program. This skeleton takes two command line arguments (like the `comparator.c` program), invokes the `my_strcmp` function from the `compare` function and prints out the numerical result of the string comparison to the screen.

YOUR TASK

Complete the `my_strcmp` function in the `comparator.s` file, provided in the `MIPS` directory of the `task1` folder. Write an implementation of the `my_strcmp` function which behaves exactly like the equivalent `my_strcmp` function in the provided C file. Make sure that you make good and correct use of MIPS registers and instructions when you are translating the `my_strcmp` function statement by statement from C to MIPS assembly. Follow the MIPS calling conventions.

You can the code from the command line using the Make build system like this:

```
❯ make run
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

my_strcmp("this", "that") = 1
```

As part of your code development and debugging process you might also want to use the MARS IDE, possibly single stepping over your code and observe its behaviour.

Test your code with different inputs like this:

```
❯ make run ARGS="car train"
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

my_strcmp("car", "train") = -1
```

For this task you will have to complete MIPS assembly functions marked with “TODO”! Make sure that the output format of your MIPS string comparison program follows the example shown above, and is identical to what the provided C program shows.

Task 2: Tokenizer

In this second task, you are being asked to write a simple tokenizer, i.e. a function that separates a longer string comprising multiple words separated by spaces and punctuation symbols into a

sequence of words (without spaces and punctuation). This tokenizer will play a crucial role in the spell checker application of task 3.

Let's have a look at what the tokenizer will do. Go to the C directory in the task2 folder, where you find an implementation written in C. We can build and run our tokenizer like this:

```
❯ make run
gcc -c -o tokenizer.o tokenizer.c -I.
gcc -c -o libio.o libio.c -I.
gcc -c -o libstring.o libstring.c -I.
gcc -o tokenizer tokenizer.o libio.o libstring.o -I.
Input = Grammar is a system of rules which governs the production and use
of utterances in a given language. These rules apply to sound as well as
meaning, and include componential subsets of rules, such as those
pertaining to phonology (the organisation of phonetic sound systems),
morphology (the formation and composition of words), and syntax (the
formation and composition of phrases and sentences). Many modern theories
that deal with the principles of grammar are based on Noam Chomsky's
framework of generative linguistics.
Token = Grammar
Token = is
Token = a
Token = system
Token = of
Token = rules
Token = which
Token = governs
Token = the
Token = production
Token = and
Token = use
Token = of
Token = utterances
Token = in
Token = a
Token = given
Token = language
Token = These
Token = rules
Token = apply
Token = to
…
```

After compiling the `tokenizer.c` file and a couple of auxiliary libraries, `libio.c` and `libstring.c`, which contain helper functions for I/O and string handling, the tokenizer application is launched with a single long argument string. The application prints the input string, followed by each token (word). Note that separating spaces (' ') and other punctuation have been eliminated!

For this coursework the symbols acting as separators for words are: `Space` and `()[].,;:"#` , any digits, and also the special string null terminator `\000`.

YOUR TASK

You need to write three functions using MIPS assembly for this task: `tokenize` (in `tokenizer.s`), `strncpy` (in `libstring.s`) and `strcspn` (also in `libstring.s`). As before, a good approach to this task is to look at the provided functions written in C, and then translate the C functions to MIPS statement by statement. Your functions should faithfully model the behaviour of the C equivalents.

Explanations and specifications of the string handling functions are provided in the libstring.h C header file. The header file contains the signatures of the functions and "declare" the functions for the C compiler, i.e. make the compiler aware of the names, parameters, and return types. Your MIPS implementations of the `strncpy` and `strcspn` go into the `libstring.s` file where you already find an implementation for the `strlen` function.

When launched with make you should see the same output for your complete tokenizer as for the provided C implementation:

```
❯ make run
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

Input = Grammar is a system of rules which governs the production and use
of utterances in a given language. These rules apply to sound as well as
meaning, and include componential subsets of rules, such as those
pertaining to phonology (the organisation of phonetic sound systems),
morphology (the formation and composition of words), and syntax (the
formation and composition of phrases and sentences). Many modern theories
that deal with the principles of grammar are based on Noam Chomsky's
framework of generative linguistics.
Token = Grammar
Token = is
Token = a
Token = system
Token = of
Token = rules
Token = which
…
```

For this task you should reuse your string comparison function from task 1. Copy your code of the function body `my_strcmp` into the `strcmp` function in the string handling library `libstring.s`.

For this task you will have to complete MIPS assembly functions marked with "TODO"! As before, make sure that your MIPS program follows exactly the output format of the provided C program.

Task 3: Spell Checker

Now it's time to put all components together and develop a 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. For the purposes of this exercise, spell checking will be case insensitive. Thus, 'tHen' is a correct spelling of a dictionary word 'then'.

Our INF2C-CS spell checker reads in a dictionary from a file (`words.txt`), builds an internal data structure to store the dictionary, before it reads in a text to be spell checked from another file (e.g. `exampletext.txt`), which is also stored word-by-word in an internal data structure. It then checks each word of the input against the dictionary, providing output indicating whether the word is contained in the dictionary or not.

Given the provided dictionary and the same input text as in task 2, we would run the spell checker like this and expect the following output:

```
❯ make run
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

INF2C-CS Spellchecker
Loading dictionary...done.
Loading text...done.
Spellchecking...
Grammar - not ok
```

```
is - not ok
a - ok
system - ok
of - ok
rules - not ok
which - ok
governs - not ok
the - ok
production - ok
and - ok
…
```

“Ok” here means that the spell checker has found the word in the dictionary, whereas “not ok” means the word might have been misspelled as it doesn’t exist in the dictionary. Please note that the provided dictionary is very small and does not contain many words even if they are commonly used in many English texts. For example, the word “grammar” as the first word of the example input is marked as “not ok” as it’s not contained in the dictionary.

YOUR TASK

For this task you first need to write the spellchecker in C. There is a framework provided in the C directory of task 3. You will need to implement both data structures and functions for loading the dictionary, the input text, and the spellchecker itself. Auxiliary string handling functions e.g. from tasks 1 and 2 can and should be reused! Functionality is distributed over a number of files, which contain functions concerned with a single aspect of the project (loading of dictionary, loading of input, spell checking).

Once you have completed the C spellchecker, you will need to translate it statement by statement into MIPS assembly. There is a framework for you provided for task 3, which you should be using. The framework provides file and function structures for your C and MIPS projects. This means that your MIPS code is not contained in a single file, but will be placed in different files in line with the C code.

Make sure your spellchecker works on the provided test cases, and produces output in the format shown above!

HINTS

- You might want to use the provided libraries (libstring, libio). You can extend the libraries with additional functions if you need them,
- You might want to include some amount of error checking and handling, e.g. catching non-existing input files. You can use the `error` function in `error.c` for error reporting, which can be suitable extended.
- Test your code with several inputs, including corner cases. The repository contains a number of test cases, which you should use for testing.

OUTPUT FORMAT

As we will use an automated checking framework it is important that you use a standardised output format for your spellchecker. Each line should contain a single word of the input string to be checked, followed by a “ - “ separator and then either “ok” or “not ok” as shown in the example output above.

The template for the output is: WORD - [ok|not ok]

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 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.

Restrictions on the use of C Library Functions

The only C library functions that can be used are basic I/O functions like `scanf` and `printf`, and basic file handling, which are called within the functions `printChar`, `printInt`, `printString` and `readString` etc. provided with the C files. Don't use C library functions directly in your code - there is not MIPS equivalent.

Instead, the provided libraries (libstring, libio, libmem) contain useful functions, which you can use. In fact, for tasks 2 and 3 you can reuse code from the previous tasks and add e.g. your string comparison function to the string handling library.

How to get support

There is plenty of support available to you if you should get stuck or have any questions about this coursework: The INF2C-CS lab demonstrators can provide general and practical support if something doesn't work, or you have general C or MIPS related questions. You can also drop in to InfBase to get additional tutoring and support for your year 1/2 courses. Please take advantage of the labs and the lab demonstrators who are there to answer your questions.

The INF2C-CS Piazza Forum is a great place to find support. 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.

Please be reminded that academic misconduct regulations also apply to Piazza, so you should **not** post coursework solutions or code snippets publicly. If in doubt, make your question private, and an instructor will check if the post is acceptable.

Academic Integrity and the Use of ChatGPT

Please remember the good scholarly practice requirements of the University regarding work for credit. You can find guidance at the School page:

https://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct

This also has links to the relevant University pages.

As mentioned, your code will be subject to further automatic and manual review after the deadline. Submitted code will be checked for similarity with other submissions using the MOSS system. MOSS has been effective at finding similarities in the past and it is not fooled by name changes or reordering of code blocks. Courseworks are **individual**, and we expect everyone to submit their sole, independent work.

Extensions or Extra Time Adjustments (ETA) are permitted up to a maximum of six days, but cannot be combined. If assessed coursework is submitted late without an approved extension or ETA, a penalty of 5% per calendar day will be applied, up to a maximum of six days after the original deadline, after which a mark of zero will be given.

While you must work on this coursework on your own, you can discuss general ideas with other students without sharing actual code or solutions. You are not allowed to use ChatGPT or other Generative AI tools for this coursework (even if this might be tempting and chances are that ChatGPT would be able to create a solution). Instead, if you need help with the coursework, please make use of the support provided to you! It is important for your learning that you develop the knowledge, skills and understanding associated with the learning outcomes of this coursework!

Marking Criteria

Your programs will be primarily judged according to correctness, completeness, and correct use of the C programming language and MIPS instructions, registers and function call conventions. We will make extensive use of an automated testing framework to thoroughly check your submission.

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.

For tasks 1 and 2, you will be graded only for your MIPS programs. For task 3, you will be graded for your C and MIPS programs.

When testing your submitted code we will be using additional test cases that go beyond the ones provided to you in the repository. Even if your programs work correctly on the provided test cases, it is not guaranteed that you will receive full marks for your submission as our test cases used for marking might uncover previously undetected bugs in your code.

CONTRIBUTIONS OF EACH TASK

- Task 1 contributes 15% of this coursework mark
- Task 2 contributes 25% of this coursework mark
- Task 3 contributes 35% of this coursework mark
- Bonus tasks 1 and 2 contribute together 25% of the coursework mark

This means with tasks 1 and 2 together (assuming you get full marks for both of them), you can get a “pass mark” of 40% for this coursework. By completing tasks 1-3 (again, assuming you get full marks for all of these), you can achieve an overall mark of up to 75% (“A”).

Bonus Marks

While you can reach marks up to a standard of an “A” grade by completing all the basic tasks of this coursework, you can earn “bonus marks” for your coursework. Bonus marks are awarded in line with the “exceptional” ranges of the University’s Extended Common Marking Scheme, i.e. the range between 70-100%. Bonus marks are only awarded if your coursework is at least satisfactory across all of the basic marking criteria! If you have struggled with the basic part of this coursework, or have spent significantly longer than 20 hours, please **do not** attempt these additional tasks for bonus marks! It would be better use of your time revising INF2C-CS materials, or engage with coursework for other courses.

There are two additional bonus tasks that you can approach for bonus marks. These extra tasks are hard and/or require significant development effort. Attempt these only if you have plenty of spare time! You don’t need to do any of these to achieve an “A” - this is for bonus marks only!

1. For any word not found in the dictionary, can you determine the “most similar” word that is contained in the dictionary? This would be useful to suggest corrections to users of your spellchecker where a word might be misspelled. For this task you need to research word similarity metrics and choose one to implement.

   Have a look here for inspiration: https://ladal.edu.au/lexsim.html

When attempting this bonus task, provide your implementations in C/MIPS in the bonus1 folder, and make sure that your spellchecker provides output in this format:

WORD - not ok - ANOTHER WORD

2. Can you asymptotically improve the runtime performance of your spellchecker? The MARS IDE has an integrated dynamic instruction counter, which you can use to evaluate the number of instructions needed to execute a program:

   Click *Tools* -> *Instruction Counter* -> *Connect to MIPS*. Then run your program.

   You can also access the instruction counter from the command line by adding the MARS `ic` command line option.

   When attempting this bonus task, provide your implementations in C/MIPS in the bonus2 folder, and provide a README file containing information describing your solution and the instruction counts of your baseline and improved code versions.

Submission Logistics

Once you've pushed all of your changes to GitHub and you're happy with your code, you're finished! We will grade the latest submission on your main branch, so take care not to push any changes after the deadline unless you have an approved extension. Submissions after the deadline without an approved extension will incur penalties as detailed on Learn.