INF2C Computer Systems

Coursework 1

MIPS Assembly Language Programming

Deadline: Wed, 23 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 the details on MARS, see Lab Exercise 1. The assignment also asks you to write three C programs 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 Computer Systems and 20% of the overall course mark.
Please bear in mind that the guidelines on academic misconduct from the Undergraduate year 2 student handbook are available on the following link `http://web.inf.ed.ac.uk/infweb/student-services/ito/students/year2`.

1 Overview

Figure 1 shows a typical word search puzzle, where the objective is to find all words within the grid along horizontal, vertical and diagonal lines. The goal of this assignment is to develop a word search puzzle solver. The assignment consists of six tasks that progressively extend the word search capability. For all tasks, you will use a provided dictionary of valid words, which are the only words you need to search for.

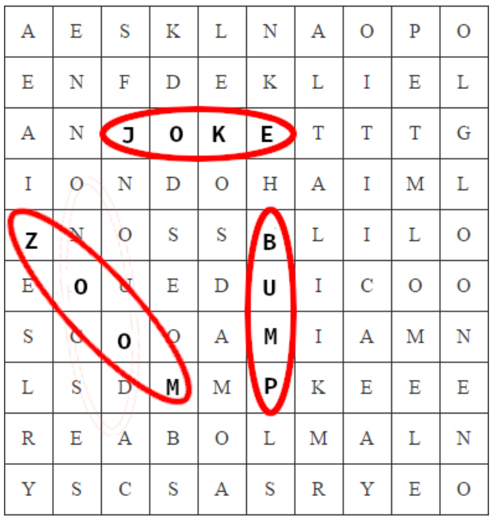

Figure 1: 2D grid puzzle

1.1 Task 1: Find the first match

The first task is to find a word in a given text, which is defined as follows: given a set of strings named *dictionary* (each string is a *word*) and a 1-dimensional string named *grid*, find the *first* word that matches in the grid.

Two input files are provided for this task (as well as for all the subsequent tasks). The first input is a dictionary file, which 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. The second input is the puzzle grid, which in this task is just a single-line string of alphabetic characters (`a-z`) in any order and combination. The grid file is terminated by a newline (`\n`) followed by End-Of-File (EOF). A valid grid file does not include any other characters. You do not need to handle invalid dictionary and grid inputs.

You are given a MIPS file named `1dstrfind.s` which contains a skeleton of your program. This skeleton reads a grid file named `1dgrid.txt` and stores its content into a null-terminated string named `grid`. It also reads a dictionary file named `dictionary.txt` and stores its content into a null-terminated string named `dictionary`.

Your task is to find the first matching word in the `grid` string that exists in the `dictionary` string and output the index of the occurrence in the `grid` followed by the matching word or `-1` if no word in the `dictionary` is present in `grid`.
If the dictionary consists of following words:

```
lose
this
just
happen
part
not
```

```
have
first
nothing
thing
reason
```

and the content of the grid file is:

```
maybenothinginthisworldhappensby
```

then the output of your program should be:

```
5 not
```

because `not` is the first word that matches in the grid string.

Note:

The order of words in the dictionary does **not** matter. In this example, even though the word `this` comes before the word `not` in the dictionary, the output considers the word `not` being the match, as it comes first in the grid string.
In another example, if the content of the grid file is:

```
foolswhodontorespectthepastareli
```

then the output of your program should look like:

```
-1
```

which indicates that no valid word is found.

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 `1dstrfind.c` file.

For convenience, `1dstrfind.c` includes the definitions of helper functions such as `read_string` and `print_char` that mimic the behaviour of MARS system calls with the same names. Derive your MIPS program from the `1dstrfind.c` file.

Do not try optimizing your MIPS code. We suggest 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 1 which converts an entered decimal number into hexadecimal.

1.2 Task 2: Find all matches

In this task, you need to extend code from task 1 (both C and MIPS) to find all matching dictionary words, including fully- or partially-overlapping matches, in the `grid` string. You may find examples of overlapping matches below. The output of this task is the index of an occurrence of a dictionary word in the `grid` string followed by the matching word on each line of the output or `-1` if no word in `dictionary` matches.
Assuming the `dictionary` is the same as the example in task 1, if the content of grid file is:

```
maybenothinginthisworldhappensby
```

then the output of your program would be:

```
5 not
5 nothing
7 thing
14 this
23 happen
```

Notes:

1. The word `nothing` has full overlap with the word `not`. Since both words have the same indexes in the `grid`, the output includes index `5` twice.

2. The word `thing` has full overlap with the word `nothing` and partial overlap with the word `not`.

3. You do not need to write a separate function for this task. Instead, you must extend your code in the `1dstrfind.s` and `1dstrfind.c` files to output all matches, including overlapping.

1.3 Task 3: Find all horizontal matches in a 2D grid

To level up the game, the grid file now consists of a 2D grid of alphabetic characters (`a-z`) instead of a single-line string. Rows in this grid are delimited by a newline (`\n`) and the file is terminated by End-Of-File (EOF). All rows have the same length.

You are given a MIPS file named `2dstrfind.s` which contains a skeleton of your program. This skeleton reads a grid file named `2dgrid.txt` and stores its content into a null-terminated string named `grid`. It also reads a dictionary file named `dictionary.txt` and stores its content into a null-terminated string named `dictionary`. You are also given a C file named `2dstrfind.c` which is the C version of the MIPS skeleton.

A word is said to be horizontally matched if all characters match inside a row without wrap-around. The direction of a horizontal match must be from *left* to *right*. Your task is to write MIPS and C programs, which find all horizontally matching words (including overlapping) in the `grid` string that exist in the `dictionary` string and output the coordinates x,y (where x is the row number and y is the column number) of the first letter of an occurrence of a dictionary word in the `grid` followed by the letter 'H'(for

horizontal) and followed the matching word per line of the output or `-1` if no word in the `dictionary` is present in `grid`.
If the dictionary consists of following words:

```
area
eden
this
happen
less
not
have
first
nothing
thing
ear
reason
```

and the content of the grid is:

```
tahbenotljnginthis
soaodoneppahapacci
uevtetssarythingha
jpeansfdareasbetak
```

then the output of your program should look like:

```
0,5 H not
0,14 H this
2,11 H thing
3,8 H area
```

Notes:

1. Row and column numbers start from zero.
2. The grid does not need to be square, but all rows must have the same length.
3. A 1-row grid is valid.

1.4 Task 4: Find all horizontal/vertical matches in a 2D grid

In this task, your job is to extend your task 3 (both C and MIPS) code to support vertical matches as well. A word is said to be vertically matched if all characters match within a column in the *downward* direction and without wrap-around. Similar to task 3, your program must output x,y coordinates of the first letter of an occurrence of a dictionary word in `grid` followed by letter 'H'or 'V'(for horizontal or vertical match, respectively) and followed by the matching word per line of the output or `-1` if no dictionary word is present in `grid`.
Assuming the dictionary is the same as the example in task 3 and the content of the grid is:

```
tahbenotljnginthis
soaodoneppahapacci
uevtetssarythingha
jpeansfdareasbetak
```

then the output of your program should look like:

```
0,2 V have
0,4 V eden
0,5 H not
0,5 V not
0,14 H this
2,11 H thing
3,8 H area
```

Notes:

1. In this example, the word `not` is matched in two directions, hence the output includes its index twice.
2. Order of the matches and their directions does not matter.
3. You do not need to write a separate function for this task. Instead you have to extend your code in the `2dstrfind.c` and `2dstrfind.s` files to output all horizontal and vertical matches.

1.5 Task 5: Find all matches in a 2D grid

In this task, your are required to extend task 4 (both C and MIPS) to support matches in all directions, including diagonals. A word is said to be diagonally matched if all characters match along a diagonal. A valid diagonal match extends from left to right in the downward direction. Similar to the previous two tasks, your program must output x,y coordinates of the first letter of an occurrence of a dictionary word in the `grid` followed by letter 'H', 'V'or 'D'(for horizontal, vertical and diagonal matches, respectively) and followed by the matching word per line of the output or `-1` if no dictionary word is present in `grid`.
Assuming the dictionary is the same as the example in task 3 and the content of the grid is:

```
tahbenotljnginthis
soaodoneppahapacci
uevtetssarythingha
jpeansfdareasbetak
```

then the output of your program should look like:

```
0,2 V have
0,4 V eden
```

```
0,5 H not
0,5 V not
0,14 H this
1,7 D ear
2,11 H thing
3,8 H area
```

Notes:

1. Order of the matches and their directions does not matter.

2. You do not need to write a separate function for this task. Instead you have to extend your code in the `2dstrfind.c` and `2dstrfind.s` files to output matches in all directions.

1.6 Task 6: Find all matches in a 2D grid with wrap-around

This is the ultimate grid solver. Your job is to extend the grid solver in task 5 to support matches that wrap around the grid, as shown in figure 2.

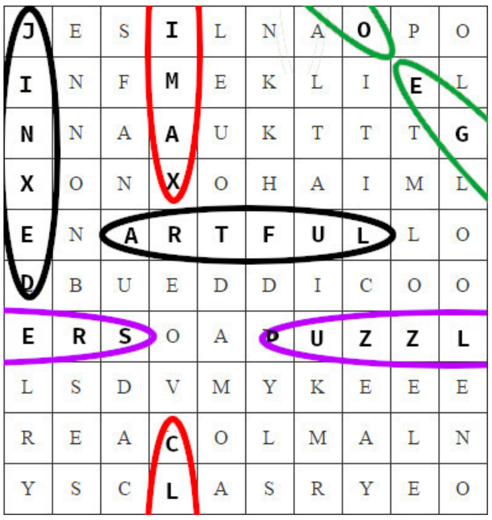

Figure 2: Black matches are the same as task 5. Purple, red and green lines show horizontal, vertical and diagonal matches wrapped around the grid.

You are given a MIPS file named `wraparound.s` which contains a skeleton of your program. The same as previous tasks, this skeleton reads a grid file named `2dgrid.txt` and stores its content into a null-terminated string named `grid`. It also reads a dictionary file named `dictionary.txt` and stores its content into a null-terminated string named `dictionary`. You are also given a C file named `wraparound.c` which is the C version of the MIPS skeleton.

Your task is to write MIPS and C programs, which find all matching words, including matches that wrap around the grid horizontally, vertically or diagonally. Your program must output x,y coordinates of the first letter of an occurrence of a dictionary word in

the `grid` followed by letter 'H', 'V'or 'D'(for horizontal, vertical and diagonal matches, respectively) and followed by the matching word per line of the output or `-1` if no dictionary word is present in `grid`.
Assuming the dictionary is the same as the example in task 3 and the content of the grid is:

tah~~o~~enotljngin**v**his
soa~~t~~doneppahapa**e**ci
vevtetssarything**h**a
jpe~~n~~nsfdareasbeta**a**

then the output of your program should look like:

```
0,2 V have
0,4 V eden
0,5 H not
0,5 V not
1,7 D ear
2,11 H thing
2,16 H have
2,16 D have
3,8 H area
3,3 V not
```

Notes:
1. The matched words `have` (underlined), `not` (struck-through) and `have` (bolded) are wrapped around the grid, horizontally, vertically and diagonally respectively.
2. The grid does not contain multiple wrap-around matches.

1.7 Specifications for Inputs

Your programs for all tasks will be tested against dictionary and grid files that adhere to the following rules:

- A dictionary file may contain a maximum of 1000 words.
- A single word in the dictionary file may contain a maximum of 10 alphabetic characters. All characters are lowercase.
- All grid coordinates start at 0. The maximum grid coordinate in any dimension (x, y) is 31.
- A grid file may contain only lower-case alphabetic characters (`a-z`).
- You do not need to handle invalid inputs in either dictionary or grid files.
- You are not allowed to change the *names* of dictionary and grid input files. You are welcome to modify the content of these files to facilitate development and debug.

1.8 Specifications for Outputs

The outputs of your program must follow the following set of rules:

- For tasks 1 and 2, each line of the output containing a matching word should consist of the index of the first matching character in the grid followed by a space character (‘ ’) followed by a word. Example:

```
    12 joke
```

- For tasks 3 to 6, each line of the output containing a matching word should consist of coordinates of the first matching character in the grid delimited by a comma character (‘,’) followed by a space character (‘ ’) followed by one letter indicating the direction (‘H ’, ‘V ’or ‘D ’) followed by a space character (‘ ’) followed by a word. Example:

```
    0,12 H joke
```

- Matches may be printed in any order.
- Each line of the output should contain only one match. If no matches are found in the entire grid, the output should be one line containing -1.

Note that the outputs of your programs will be auto-marked, and failing to follow the correct format will likely result in a mark of 0.

1.9 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. *Use of additional library functions beyond the ones specified here will result in a* 0 *mark for the C implementation.*

2 Advice on Program Development and Testing

The assignment is structured in a way that encourages you to gradually extend your code base. As such, keeping your code properly structured, readable and tidy will make developing and debugging easier and save your time. In practical terms, a program is well-structured when the program logic is clear, and when it is written in small, easy-to-read computational blocks. Meaningful names for labels and data go a long way toward enhancing readability and reducing the chance of programmer error.

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 nice when opened in a different editor.

If you use `Emacs`, the command `(m-x)untabify` will remove all tab characters from the file in a buffer.

Always make sure your code compiles/assembles without warnings and errors. Below, you may find language-specific guidelines.

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 -Wall -o prog prog.c
```

If compilation succeeds without any errors, it creates an executable `prog` which can then be 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 strings 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.

Ultimately, you must ensure that your MIPS programs assemble and run without errors when using MARS *on the command line on DICE*, which is how we will be running your programs for marking purposes.

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 on the command line will assemble and runs prog.s.

```
    java -jar mars.jar sm me prog.s 2>/dev/null
```

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) The `me` option tells MARS to display assembler messages to standard error instead of standard output. This allows you to separate MARS messages from MIPS program output using redirection (e.g. `2>/dev/null`, which prevents standard error from showing up on console).

3) 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 1dstrfind.c 1dstrfind.s 2dstrfind.c 2dstrfind.s wraparound.c wraparound.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

Each task is worth 5% for the C implementations. Since the C implementation for Task 1 is provided, it will not be tested or counted toward the mark for this coursework. For each of tasks 1, 2 and 3, the MIPS implementation is worth 10%; for tasks 4, 5 and 6, the MIPS implementation is worth 15% each.

Because multiple tasks are covered by each file, we will test these by varying the grid input. We will NOT be changing your code. Your solutions will be evaluated with a number of test inputs. For each task, your mark will be proportional to the pass rate of the tests.

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 – chances are, others have already encountered (and, possibly, solved) the same problem. If you can't find the answer to your question, start a new discussion. You should also take advantage of the drop-in labs and the lab demonstrators who are there to answer your questions.

October 8, 2019