INF2C Computer Systems

Coursework 1

MIPS Assembly Language Programming

Deadline: Fri, 23 Oct (Week 5), 16:00

Instructors: Boris Grot, Aaron Smith

TA: Dmitrii Ustiugov

This assignment will introduce you to writing programs in MIPS assembly. You will write five MIPS programs and test them using the MARS IDE introduced in Lab 1. This is the first of three assignments for the INF2C Computer Systems course and is worth 20% of the overall class grade.

1 Overview

In Coursework 1 you will develop MIPS programs that work with encrypted texts. The assignment contains five tasks that let you develop different ciphers in increasing order of complexity. All tasks use a number of input files with encrypted and unencrypted text and may have other supplementary files. Your solutions are graded automatically and it is very important they follow the input(§1.6) and output(§1.7) formats described in this document. Note that there is no need to optimize your MIPS code as this is not the goal of this assignment.

1.1 Task 1: Find all words in a text (15 pts)

The first task is a warm up exercise that will help you implement the ciphers in the following tasks. In this task you will find all words in a text file and list each word found in the program's output.

You are provided with an example input file (`input_words.txt`) that follows the rules described in (§1.6) and a MIPS assembly file (`find_words.s`) which contains a skeleton

program to get you started. This code reads the input file and stores its content into a null-terminated string (`input_text`).

Your task is to find all words in `input_text` and list them in the order of occurrence one per line. For example, if `input_text` contains:

```
this is
my plain text
i will not encrypt it
```

then the output from your program would be:

```
this
is
my
plain
text
i
will
not
encrypt
it
```

For this task, we provide an output file (`output.txt`) that corresponds to the given input file (`input_words.txt`) to check your program. Make sure the format of the input files your program uses and the format of the output of your program corresponds **exactly** to the format of the provided example files. Your output should follow the rules described in (§1.7).

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` included in the git repository. 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: Steganography (15 pts)

Steganography hides a secret message inside a larger text and is one of the oldest methods used to conceal information dating back to around 550 B.C. Unlike cryptography, the message is not encrypted and is visible to anyone but does not attract attention to itself. In this task, you will write a program to *retrieve* a concealed message containing one or more words.

Each word of the concealed message is located at the position in the line that corresponds to the line number, counting from the top line (i.e., first) to the bottom line (i.e., last). The first word of the message is the first word of the first line, the second word of the message is the second word of the second line, and so on. If a line has fewer words than its line number that signifies a newline character. For example, given the following input:

```
the sun is shining
the cake is ready
the dog is barking
parents play with a little child
and honest people never lie
so far so good
stay healthy sleep walk exercise and run regularly
```

the hidden message your program would output is:

```
the cake is a lie
run
```

You are provided with an input file (`input_steg.txt`) and skeleton code (`steg.s`) that reads the input file's content into a null-terminated string (`input_text`). Your task is to extend the code to read the hidden message from the input text and output it. Please note that the message can span one or more lines.

1.3 Task 3: XOR cipher (20 pts)

In this task you will implement a XOR cipher which encrypts text by applying the bitwise XOR operator to characters (or sets of characters) using a secret key. You will encrypt the input text with a secret key that can be 1, 2, 3, or 4 bytes long. Your program needs to be able to perform encryption with all four key sizes. When encrypting, your program must preserve all whitespaces and newlines in the output.

An ASCII character is encoded with 1 byte. Therefore a 1 byte key is XOR-ed with one character at a time, a 2 byte key is XOR-ed with 2 characters, a 3 byte key with 3 characters and a 4 byte key with 4 characters.

The cipher encrypts the text from the first (left-most) character in the first line to the last (right-most) character in the last line. Given the key size of N bytes, the cipher operates as following. First, the cipher reads first N characters, including the whitespace and newline characters, and XOR-s the first (left-most) character with the first (left-most) byte of the key, the second character with the second byte of the key, ..., the Nth character with the N-th byte of the key. After that, the cipher reads the next N characters and continues by XOR-ing the (N+1)-th character with the first byte of the key, and so on. Note that the cipher needs to recognize the whitespace and newline characters and leave them unchanged in the output. For example, for the input text with a single line '`ab c`' (terminated by '`\n`'), a 3 byte cipher XOR-s `a` with the first byte of the key, it XOR-s `b` with the second byte of the key, it would XOR ' ' with the 3rd byte of the key but leaves ' ' unchanged in the output, it XOR-s `c` with the first byte of the key, it would XOR '`\n`' with the second byte of the key but leaves '`\n`' unchanged.

Consider another example with the following input text:

```
ab cd
e fg
```

has the following binary encoding:[1]

```
|01000001|01000010|whitespace|01000011|01000100|newline|
|01100101|whitespace|01100110|01100111|newline|
```

Applying a XOR cipher with the 2 byte secret key `|00001000|00010000|` we obtain:

```
|01001001|01010010|whitespace|01010011|01001100|newline|
|01101101|whitespace|01101110|01110111|newline|
```

As ASCII characters the output looks like the following:

```
IR sl
m nw
```

On the first line `a`, `whitespace`, `d` are XOR-ed using the first byte of the 2 byte key whereas `b`, `c`, `newline` are XOR-ed using the second byte of the key. Then, in the second line, `e` is XOR-ed using the left byte of the key, and so on.

We provide two input files: `input_xor.txt` contains the text to encrypt and `key_xor.txt` contains the secret key. We provide skeleton code `xor.s` that reads `input_xor.txt` and `key_xor.txt`, storing their contents into null-terminated strings named `input_text` and `key`, accordingly.

1.4 Task 4: Book cipher (20 pts)

In this task you will decode a message using a book cipher, which has been in widespread use for the last 500 years. A book cipher uses the location of words in a book as a key to describe a message.

Each word of the input text is encoded as `X, Y` followed by a newline character. X is the line number in the book and Y is the number of the word. Lines are counted from the top to bottom, starting with line 1. Words are counted from left to right, starting with word 1. If a line has fewer words than specified by the cipher, your program should recognize it as a newline. For example, given the following input text:

```
3, 6
4, 5
1, 6
8, 4
2, 10
8, 5
```

and the following book:

[1]The symbol "|" is used to separate each byte for readability **only** in this document. The input or output files must not have this symbol.

```
the wind behind the window was becoming
stronger and stronger
i woke up quite early this morning
first i cooked my breakfast
after finishing the breakfast
i removed the trash
and went to the university
finally i can ace ciphers
```

your program should output:

```
this breakfast was ace
ciphers
```

This task has two input files: `input_book_cipher.txt` contains the keys for decoding the book, and `book.txt` is the book to decode. We provide skeleton code `book.s` that reads `input_book_cipher.txt` and `book.txt`, and stores their contents into the null-terminated strings `input_text` and `book`, accordingly.

1.5 Task 5: Cracking the XOR cipher (30 pts)

In this final task, you will develop a program to crack the XOR cipher that you developed in §1.3 by finding the secret 1 byte key. One of the methods that cryptanalysts can use is deciphering certain words and phrases that usually appear in specific locations of the encrypted text, e.g., "Dear Madam" in the beginning of the message. In this task, we will give your program a hint that contains a phrase (i.e., one or more words) that appears in the original unencrypted text.

The hint phrase is a single line that contains one or more words separated by whitespace and null-terminated. The text contains the phrase if any part of the text contains all the words in the same order as they appear in the phrase. However, in the text some of the phrase's words can be separated by newline instead of whitespace inside the hint file.

To crack the secret key, your program should try all possible keys, checking if the hint phrase appears in the decrypted text. The suggested algorithm is as follows. First, the program decrypts the input text with a key of `|00000000|`, and after that performs a search for the hint phrase in the decrypted text (note that the phrase can span one or more lines in the input text). If the hint phrase is present, then the secret key, which your program used to decrypt the text, is correct and the program can terminate after outputting the secret key in *binary* format, i.e., `00000000\n`. If the hint is not present, the program repeats the procedure with `|00000001|`, and so on, checking all the keys up to `|11111111|`. If no keys worked, i.e., the phrase was not found with any key, the program should output `-1` and terminate.

We provide two input files: `input_xor_crack.txt` contains the encrypted text to decrypt and `hint.txt` contains the hint phrase. We provide skeleton code `xor_crack.s` that reads `input_xor_crack.txt` and `hint.txt`, storing their contents into null-terminated strings `input_text` and `hint`, accordingly.

1.6 Specifications for Inputs

Your programs will be tested with input files that adhere to the following rules:

- Input files are ASCII text.[2]
- A file contains one or more lines and each line consists of one or more words of lowercase ASCII characters (`a-z`).
- Words are delimited by a whitespace (‘ ’).
- All lines, including the last line, are terminated by a newline (`\n`).
- The input file is terminated by End-of-File (EOF).
- A valid input file does not include any other characters and you do not need to handle invalid inputs in any of the files.
- You are not allowed to change the *names* of the input files. You are welcome to modify the content of these files to facilitate development and debug. Note, that when changing the input files, e.g., during testing your program, you need to maintain the exact same format, namely the newline character at the end of each line.
- Do not change the format of the input files that we provide. If your program uses a different input file format, the automatic marker is likely to fail and give you zero points for the particular task.

1.7 Specifications for Outputs

The output of your programs must follow the set of rules that we specify for each task. Note that the output of your programs will be auto-marked, and failing to follow the correct format will likely result in a mark of zero.

- Output is printed to STDOUT.
- The output may contain lowercase and uppercase ASCII characters (`a-z` and `A-Z`) and for Task 5 it may contain `1`, `0`, and `-1`.
- All lines, including the last line, are terminated by a newline (`\n`).
- There is no whitespace allowed between the last character of a line and the newline character.
- Since the output of your programs is to STDOUT and not to a file, the output is not terminated by End-of-File (EOF).

[2] One can find ASCII encoding at this link: http://sticksandstones.kstrom.com/appen.html

2 Advice on Program Development and Testing

A good way to go about writing a MIPS assembly 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 assembly. Once the C code for a program is correct, one can translate it to an equivalent MIPS assembly program statement-by-statement.

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.

In all tasks, we recommend you 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 *from 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. Place all necessary input files in the same directory as the MARS JAR file.

```
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

The skeleton code and example inputs are contained in the course work Git repository on GitLab: https://git.ecdf.ed.ac.uk/asmith47/inf2c-cs-20

You will modify the skeleton files provided for the tasks and push your changes to your fork of the coursework Git repo on GitLab. You can push changes to GitLab any time up until the submission deadline. We will not mark any changes pushed after the submission deadline has passed.

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

Task 1 and 2 are worth 15% each; tasks 3 and 4 are worth 20% each; tasks 5 is worth 30% each. 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. We will NOT be changing your code.

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 [3]. MOSS has been effective in the past at finding similarities. It is not fooled by name changes and reordering of code blocks.

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

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 9, 2020

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