Inf2C Computer Systems

Coursework 1

Deadline: Thu 25 Oct 2012, 12:00

Paul Jackson

1 Description

The aim of this assignment is to introduce you to writing MIPS assembly-code programs. The assignment asks you to write two MIPS programs and test them using one or more of the SPIM simulators. For details on SPIM, see the first lab script, available at:

`http://www.inf.ed.ac.uk/teaching/courses/inf2c-cs/labs/lab1.html`

and the Patterson and Hennessy text book.

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 12.5% of the overall course mark.

Please bear in mind the guidelines on plagiarism which are linked to from the Informatics 2 Course Guide.

1.1 Task A: ROT13 encoding

This first task is a warm-up exercise. It helps you get familiar with the basic structure of MIPS programs and with using one of the SPIM simulators.

Write a MIPS program `rot13.s` that encodes a string of lower-case characters and spaces using the ROT13 scheme. With ROT13, the characters a-z are numbered in order 0-25, and a character numbered n is encoded by the character numbered $(n + 13) \bmod 26$.

A sample interaction with your program should look like:

```
 input: the quick brown fox

output: gur dhvpx oebja sbk
```

Please use the same wording as above in the text printed by your program. This will simplify testing and marking your program.

An excellent way to create a MIPS program is first to 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 is correct, one can translate it straightforwardly statement-by-statement into an equivalent MIPS program. To this end, a C program for ROT13 `rot13.c` is provided. You can compile this program at a command prompt on our DICE machines with the command

```
gcc -o rot13 rot13.c
```

This creates an executable `rot13` which you can run by entering

```
./rot13
```

Give it a try.

For convenience, this C program includes definitions of functions such as `read_string` and `print_string` which mimic the behaviour of the SPIM system calls with the same names.

Derive your MIPS program from this `rot13.c` C program.

Don't try optimising your MIPS code, just 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 add extra C-style comments within MIPS comments documenting 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.

To look up decimal and hexadecimal codes for ASCII characters, type `man ascii` at a command prompt.

You will need to choose what kind of storage to use for each of the variables in the C code. The programs for this coursework are small enough that all single byte or single word variables can be held just in registers rather than the data segment. However arrays will need to go into the data segment. Use the `.space` directive to reserve space in the data segment for arrays, preceeding it with an appropriate `.align` directive if the start of the space needs to be aligned to a word or other boundary.

Be careful about when you choose to use `$t*` registers and when `$s*` registers. Assume that values of `$t*` registers are not guaranteed to be preserved across `syscall` invocations, whereas values of `$s*` registers will be preserved. However, do not just use `$s*` registers in your code: also make use of `$t*` registers when appropriate.

Your completed MIPS program should use 30-40 MIPS instructions. If you are going over 40, review your code with one of the demonstrators to check you are on the right track.

To ease debugging your MIPS program, write it in stages. For example, you could first write a program that leaves out the encoding and just prints out the same string as entered.

1.2 Task B: A Reverse Polish Notation Calculator

In Reverse Polish Notation ((RPN) for arithmetic, operators are always placed after their arguments. For example, the expression $(1 + 2) \times (4 + 5)$ is written as $1\ 2 + 4\ 5 + \times$. RPN expressions are evaluated left to right with the help of a stack data structure for holding operands and results. The following table shows the state of the stack after evaluating successive prefixes of $1\ 2 + 4\ 5 + \times$.

Input	Stack
1	$\underline{1}$
1 2	1 $\underline{2}$
1 2 +	$\underline{3}$
1 2 + 4	3 $\underline{4}$
1 2 + 4 5	3 4 $\underline{5}$
1 2 + 4 5 +	3 $\underline{9}$
1 2 + 4 5 + $\times$	$\underline{27}$

Here, the stacks are written horizontally, with the number on the top of each stack underlined. For further details on RPN, see the Wikipedia page on it.

For this task, you should write a MIPS program `rpn.s` for an RPN calculator that takes in a string of digit and operator characters and prints out the state of the stack after evaluating the input string. The characters that should be recognised in the input string and the corresponding actions are as follows:

`e`: (*Enter new number*) Push 0 on the stack

`0-9`: For digit d input, pop the value v off the stack and push $10v + d$.

`n`: (*Negate*) Pop the value v off the stack and push $-v$

`+`: Pop value w off the stack, then pop v off the stack and push $v + w$

`*`: Pop value w off the stack, then pop v off the stack and push $v \times w$

`-`: Pop value w off the stack, then pop v off the stack and push $v - w$

All other characters should be ignored. Initially the stack is empty. An example input and corresponding output are as follows

```
> e12e3e4e5+n

0: 12
1: 3
2: -9
```

Here `>` is a prompt you program should generate for the input string, and the stack is printed from bottom to top with its contents numbered from 0. Your program should use the same format as in this example, to ease the automatic testing carried out during marking.

Approach this task by first writing an equivalent C program `rpn.c` and then translating this C program into MIPS. Before translating, you should compile and test your C program. Ensure it is working correctly before starting on the MIPS code. It should have the same input and output formatting as the MIPS program. To help you get started with the C program, an outline `rpn-outline.c` is supplied.

As in Task A, use the C code to comment your MIPS code, and add implementation details as extra C-style comments within the MIPS comments. In your C program, your comments can 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.

In implementing the stack data structure, you have a number of design choices. For example whether to use an array index or use a pointer for the top of the stack, whether this index or pointer identifies the top element or the first free position beyond the top element the stack grows towards higher or lower addresses

The supplied C outline gives distinct functions for the push and pop operations. When implementing these in MIPS, use the `$a*` registers for arguments, and the `$v*` registers for result values. Implement calls of these functions using the `jal` *jump-and-link* instruction and returns using the `jr` *jump-register* instruction. As there are no nested function calls, there is no need to save return addresses on a program stack. When choosing when to use `$t*` or `$s*` registers, calls of your push and pop functions should assume that values in any `$t*` registers are not preserved through the calls.

When writing the C code, be careful about writing expressions such as `pop() - pop()`. C offers no guarantees as to the order of evaluation in expressions. Here, the left `pop()` might be evaluated first, then the right `pop()`, or it might be the other way round.

You might well want to make use of the `read_char` syscall to read in typed characters. If you do so, bear in mind that the documentation for it in the SPIM Appendix linked to from the Lab web page has a typo. The table on page A-44 claims that the result character is returned in register `$a0`. This should be `$v0`. This typo is corrected in recent editions (e.g. 4th) of the Patterson and Hennessy book.

You will find that the exact nature of the interaction will depend on the tool you use. For example, with `spim`, input characters are only sent to your program when *Enter* is keyed, whereas with `xspim`, input is seen immediately by system calls such as `read_char`.

Your completed MIPS program might involve up to 100 MIPS instructions. If you are going significantly over this, review your code with one of the demonstrators to check you are on the right track.

2 Program Development and Testing

You are free to develop your programs using different versions of the SPIM simulator, e.g. the linux command line version `spim`, the Linux X-Windows version `xspim` or the Windows PC version `pcspim`. Further information on these versions and the new `qtspim` is available from `http://spimsimulator.sourceforge.net/`. The new `qtspim` version is not currently installed on DICE, as we have had problems building it.

Ultimately, you must ensure that your MIPS programs are read without errors and run using the command line `spim` tool, as currently installed on all DICE machines. This is the tool that will be used to check your programs. To run a MIPS program on DICE using `spim`, type

```
spim -f filename.s
```

at a command-line prompt. Our tests will involve running your programs with input data from files. We will do this by running

```
spim -f filename.s < datafile
```

For each of your MIPS programs, you should create some test data files and use them in this way to test the program. When you do this, you will notice the output formatting is slightly different as it does not include the echo'ed input. Ensure your programs insert appropriate new-line characters in the output so the output is still well formatted in this case. For example it should not have input prompts and output text on the same line.

Ensure too your C program for Task B compiles without errors and runs properly.

3 Submission

Submit your work using the command

```
submit inf2c-cs cw1 rot13.s rpn.c rpn.s
```

at a command-line prompt on a DICE machine. Unless there are special circumstances, late submissions are not allowed. Please consult the online Informatics 2 Course Guide for further information on this.

4 Assessment

Your programs will be primarily judged according to correctness, completeness, code size, and the correct use of registers.

In assembly 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.

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

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. If you do choose to collaborate with others, you must explain how you are doing so in comments at the top of your files. If you do not, you are plagiarising.

6 Questions

If you have questions about this assignment, ask for help from the lab demonstrators, your Inf2C-CS tutor or the course organiser. Alternatively, ask your questions on the course Discussion Forum (linked to from the course home page).

11th October 2012

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