# Inf2C Computer Systems

# Coursework 1

# MIPS Assembly-Language Programming

**Deadline: Tue 22 Oct 2013, 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 the MARS IDE. For details on MARS, see the first lab script, available at:

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

The assignment also gets 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 12.5% of the overall course mark.

Please bear in mind the guidelines on plagiarism which are linked to from the online Undergraduate Year 2 Handbook.

### 1.1 Task A: Printing bit patterns

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

Write a MIPS program `print_bits.s` that has a function for producing a printed representation of a range of bits in a word. The function should expect a word $w$ in register `$a0` and a number of bits $n$ in register `$a1`. It then prints out bits $n-1$ to 0 of $w$ in a row, using a '`#`' character for a 1 bit and a '`.`' character for a 0 bit.

An excellent 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 for a program is correct, one can translate it straightforwardly statement-by-statement into an equivalent MIPS program. To this end, a

C version of the desired MIPS program is provided in file `print_bits.c`. You can compile this program at a command prompt on our DICE machines with the command

```
gcc -o print_bits print_bits.c
```

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

```
./print_bits
```

Give it a try.

The test code has 2 example calls of the `print_bits` function. The parameters for these calls are:

| $w$ | $n$ |
|---|---|
| 0xfff | 10 |
| 0xa5 | 12 |

When run, the program generates output

```
##########
....#.#..#.#
```

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

Derive your MIPS program from this `print_bits.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 Task A and Task B are small enough that all single byte or single word variables can be held just in registers rather than the data segment. However any 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.

If you were following MIPS coding guidelines strictly, your MIPS code for the `print_bits` function would need at its start to save any current values in `$s*` registers it uses, and to restore these values just before returning. The guidelines call this the *callee-save* convention. However, for simplicity just arrange that each of your MIPS functions uses distinct subsets of the `$s*` registers.

Implement calls of the `print_bits` function 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. However, do not assume that values stored in `$ra` are preserved across syscalls. Use the MIPS convention of passing arguments to your function in the `$a*` registers.

In lecture, MIPS shift instructions were introduced that shift by a constant number of places. You might find it useful to exploit one of the alternate instructions which shift by a value contained in a register. See the reference section of the MIPS appendix for details.

Your completed MIPS program for the `print_bits` function and the test code should be under 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.

## 1.2 Task B: Banner printing

Some flavours of Unix provide a *banner* function which takes a string of characters and prints the string using large characters, each made up of a 2D grid of regular-size characters. In this task, you have to write a simple variant on this function in both C and MIPS.

An example run of the C program is the following.

```
> ./banner
input: 12345
....#....##...###.....#...####
...##...#..#.....#...##...#...
....#.....#....##...#.#...###.
....#....#.......#..####.....#
...###..####..###.....#...###.
>
```

Here `>` is the Unix command-line prompt, '`input:  `' is a prompt printed by the program, and `12345` is text input. After this text is keyed in and the return key is pressed, the program prints a banner as shown.

For the purposes of this coursework, to help with debugging, we print '`.`' characters where a real program might just print space characters.

Approach this task by first writing an equivalent C program `banner.c` and then translating this C program into a MIPS program `banner.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 `banner-outline.c` is supplied.

The bit-pattern for each large character has 5 rows and 4 columns and is stored using 20 of the bits of a 32-bit word. For example, the bit-pattern for the number '4' is

```
0010
0110
1010
1111
0010
```

The word for this pattern is derived by concatenating together the bit sequences for each row. Hence, this pattern is stored as a word with hex value `0x26af2`. An array named

char_pats is provided in banner-outline.c that has the patterns for the digits '0'-'9'. For simplicity, patterns are not provided for other printing characters. If an attempt is made to print any character other than '0'-'9', the bit pattern

```
1111
1111
1111
1111
1111
```

should be used. For example:

```
> ./banner
input: 678/:90
...###..####...##...####..####...##....##.
..#........#..#..#..####..####..#..#..#..#
..###.....#....##...####..####...###..#..#
..#..#...#....#..#..####..####.....#..#..#
...##...#......##...####..####..###....##.
>
```

The characters / and : are good test characters to use, as they have ASCII codes just outside the range of the ASCII codes for the digits 0-9 and so they check if the detection of this range has been implemented correctly.

As shown in the examples, each character should be preceded by two columns of '.' characters.

The supplied C outline includes an outline

```
unsigned int get_row_pattern(char c, int i) {
    return 0xf;
}
```

for a function for retrieving the rows of bit patterns for characters. Rows are numbered from the top of a pattern, starting with 0. For example, if c is the ASCII code for the character '4', and i is 1, this function is expected to return 0x6. Your C code should complete and make use of this function. Your MIPS code should introduce a corresponding MIPS function. As with the print_bits function, pass arguments in the $a* registers. Also pass the return value in one of the $v* registers.

For printing the row of a character pattern, make use of the provided print_bits C function in Task A and the corresponding MIPS function you write for Task A.

To aid you in testing your get_row_pattern C function, the C outline includes some test code:

```
        unsigned int pat;

        pat = get_row_pattern('4', 0);
        print_bits(pat, NUM_COLS+2);
        print_char('\n');
```

```
pat = get_row_pattern('4', 1);
print_bits(pat, NUM_COLS+2);
print_char('\n');
```

You are strongly recommended to ensure that your implementation of `get_row_pattern` works correctly before starting on completing the code for the main function, both when coding in C and when coding in MIPS. A translation to MIPS of the row 0 test above is

```
                        # pat = get_row_pattern('4',0);
li   $a0, 0x34          # // 0x34 = '4'
li   $a1, 0
jal  get_row_pattern
move $a0, $v0           # // $v0 holds value of pat
li   $a1, 6             # print_bits(pat, NUM_COLS + 2);
jal  print_bits

li   $v0, 11            # print_char('\n');
li   $a0, 0x0a          # // 0x0a == '\n'
syscall
```

Complete the translation of both row tests and check you get the same behaviour as with the C code. The main function of both your C and MIPS code should be set up so that this test code rather than the normal main code is enabled if any optional program argument is provided. In C, the provided outline shows how to do this. In MIPS with MARS, the presence of a program argument can be detected by checking for register `$a0` being non-zero at the start of the main block.

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.

Your completed MIPS program might involve up to 100 MIPS instructions, including the `print_bits` code you bring forward from Task A and the code for testing your `get_row_pattern` MIPS function. 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

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. We recommend you check 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`, the command

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

at a command-line prompt will assemble and run prog.s. 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, checking the setting *Initialize Program Counter to global 'main' if defined* on the *Settings* menu achieves the same effect.

As remarked above, the MIPS program you complete for Task B should run in a special test mode when a program argument is provided. With the MARS GUI, one of the options on the *Settings* menu enables a box for entering one or more program arguments. Enter say the text `test` in this box to enable running your code in test mode. If MARS is run in command-line mode, then the Task B program `banner.s` can be run with a program argument `test` with the command:

```
java -jar mars.jar sm banner.s pa test
```

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

MARS supports a variety of pseudo-instructions, more than 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, only make use of pseudo-instructions explicitly mentioned in the appendix.

# 3 Submission

Submit your work using the command

```
submit inf2c-cs cw1 print_bits.s banner.c banner.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 Handbook 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 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

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).

8th October 2013

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