Assignment 2: smips, Simple MIPS

version: 1.1 last updated: 2020-08-02 10:00:00

Aims

- Understanding encoding of MIPS instructions
- Understanding semantics of MIPS instructions
- Generally building a concrete understanding of an example CPU
- Practising C, including bit operations

Getting Started

Create a new directory for this assignment called `smips`, change to this directory , and fetch the provided examples: by running these commands:

```
$ mkdir smips
$ cd smips
$ unzip /web/cs1521/20T2/activities/smips/examples.zip
```

There is no starting code for this assignment.

The Assignment

Your task in this assignment is to write `smips.c` an emulator for small simple subset of the MIPS .

The input `smips.c` will be the 32-bit instruction codes for MIPS instructions as hexadecimal numbers.

The command `1521 spim2hex` will give you the hex codes for MIPS instructions.

```
$ cat examples/42.s
    li   $a0, 42          # printf("%d", 42);
    li   $v0, 1
    syscall
    li   $a0, '\n'        # printf("%c", '\n');
    li   $v0, 11
    syscall
$ 1521 spim2hex examples/42.s
3404002a
34020001
c
3404000a
3402000b
c
```

The output from `smips.c` should be

- The instruction (assempler) corresponding to each instruction code
- The output produced by syscalls when executing the MIPS instructions..
- The register values when the program terminates.

For example:

```
$ cat examples/42.hex
3404002a
34020001
c
3404000a
3402000b
c
$ dcc smips.c -o smips
$ ./smips examples/42.hex
Program
  0: ori  $4, $0, 42
  1: ori  $2, $0, 1
  2: syscall
  3: ori  $4, $0, 10
  4: ori  $2, $0, 11
  5: syscall
Output
42
Registers After Execution
$2  = 11
$4  = 10
```

Reference implementation

A reference implementation is available as `1521 smips`,for example:

```
$ cat examples/triangle.hex
34080005
34090001
3402000b
340a0000
3404002a
c
214a0001
3404000a
149582a
140bfffb
3404000a
c
21290001
128582a
140bfff5
$ 1521 smips examples/triangle.hex
Program
  0: ori  $8, $0, 5
  1: ori  $9, $0, 1
  2: ori  $2, $0, 11
  3: ori  $10, $0, 0
  4: ori  $4, $0, 42
  5: syscall
  6: addi $10, $10, 1
  7: ori  $4, $0, 10
  8: slt  $11, $10, $9
  9: bne  $0, $11, -5
 10: ori  $4, $0, 10
 11: syscall
 12: addi $9, $9, 1
 13: slt  $11, $9, $8
 14: bne  $0, $11, -11
Output
*
**
***
****
Registers After Execution
$2  = 11
$4  = 10
$8  = 5
$9  = 5
$10 = 4
```

Provision of a reference implementation is a common, efficient and effective method to provide or define an operational specification, and it's something you will likely need to work with after you leave UNSW.

Where any aspect of this assignment is undefined in this specification you should match the reference implementation's behaviour.

Discovering and matching the reference implementation's behaviour is deliberately part of the assignment.

If you discover what you believe to be a bug in the reference implementation, report it in the class forum. If it is a bug, we may fix the bug, or indicate that you do not need to match the reference implementation's behaviour in this case.

MIPS Instruction Subset

You need to implement only this subset of MIPS instructions:

Assembler	C	Bit Pattern
add $d, $s, $t	`d = s + t`	`000000ssssstttttddddd00000100000`
sub $d, $s, $t	`d = s - t`	`000000ssssstttttddddd00000100010`
and $d, $s, $t	`d = s & t`	`000000ssssstttttddddd00000100100`
or $d, $s, $t	`d = s | t`	`000000ssssstttttddddd00000100101`
slt $d, $s, $t	`d = 1 if s < t else 0`	`000000ssssstttttddddd00000101010`
mul $d, $s, $t	`d = s * t`	`011100ssssstttttddddd00000000010`
beq $s, $t, I	`if (s == t) PC += I`	`000100ssssstttttIIIIIIIIIIIIIIII`
bne $s, $t, I	`if (s != t) PC += I`	`000101ssssstttttIIIIIIIIIIIIIIII`
addi $t, $s, I	`t = s + I`	`001000ssssstttttIIIIIIIIIIIIIIII`
slti $t, $s, I	`t = (s < I)`	`001010ssssstttttIIIIIIIIIIIIIIII`
andi $t, $s, I	`t = s & I`	`001100ssssstttttIIIIIIIIIIIIIIII`
ori $t, $s, I	`t = s | I`	`001101ssssstttttIIIIIIIIIIIIIIII`
lui $t, I	`t = I << 16`	`00111100000tttttIIIIIIIIIIIIIIII`
syscall	`syscall`	`00000000000000000000000000001100`

The instruction 'Bit Pattern' uniquely identifies each instruction:

- **0**: Literal bit zero
- **1**: Literal bit one
- **I**: Immediate (16-bit signed number)
- **d**, **r**, **s**, **t**: five-bit register number

System Calls

You only need to implement this subset of system calls.

Request ($v0)	Description	Arguments	C
1	print integer	$a0 = integer to print	`printf("%d")`
10	exit		`exit(0)`
11	print character	$a0 = character to print	`printf("%c")`

Execution

Execution halts silently if it would beyond the range of specified instructions,so:

- Execution halts if a branch would take execution beyond the range of specified instructions.
- Execution halts when the finish of the instructions is reached.

Registers

All 32 registers are set to be zero when execution begins.

The value of register 0 is always 0. Instructions which attempt to change it have no effect.

The other 31 registers have no special meaning and can be used for any purpose (unlike SPIM where some registerscan not be used).

Match the reference implementation when printing registers.

Note only registers with non-zero values are printed.

Examples

Some example MIPS programs are available as a zip file

You will also need to do your own testing and constuct your own examples using `1521 spim2hex`.

Note the assembler for the example programs contains pseudo-instructions such as **li**. These are translated by `1521 spim2hex` to instructions in the subset for this assignment.

`1521 spim2hex` also make some other minor translations if given instructions which are not part of the subset for this assignment. For example it the last instruction is **jr** it is deleted.

Assumptions and Clarifications

Like all good programmers, you should make as few assumptions as possible.

If in doubt, match the output of the reference implementation.

You can assume `smips.c` is given a single file as a command line argument

You can assume this file contains only 32-bit hexadecimal numbers one per line.

You can assume this file contains at most 1000 numbers.

You do not have to implement MIPS instructions, system calls, or features which are not explicitly mentioned in the tables above.

Your program should print an error message if given a hexadecimal number which does not correspond to an instruction in the above MIPS subset.

```
$ echo 0 >0.hex
$ cat 0.hex
0
$ 1521 smips 0.hex
0.hex:1: invalid instruction code: 00000000
```

The reference implementation uses `%08X` to print invalid instruction codes
Invalid instruction codes can be upper or lower case, and may or may not be padded with leading 0's

However, you will not be penalized if you implement extra MIPS instructions beyond the subset above and do nnot print an error message for them.

You can assume overflow does not occur during arithmetic or other operations.

You do not need to handle instructions which access memory such as **lw**). They are not in the subset for this assignment.

There are no instructions which access memory (e.g. **lw**) in the subset for this assignment.

You do not need to handle branch labels. **1521 spim2hex** translates these into the relative offset which are part of the branch instruction codes.

Some of the example assembler (`.s`) files contain pseudo-instructions, for example **li** You do not need to handle these. The corresponding `.hex` files contains only instructions in the assignment subset.

Some of the example assembler (.s) contain instructions outside the assignment subset, for example **jr** You do not need to handle these. The corresponding `.hex` files contains only instructions in the assignment subset.

If an invalid syscall number is supplied an error message should be printed an

```
$ 1521 smips examples/bad_syscall.hex
Program
  0: ori  $2, $0, 4242
  1: syscall
  2: ori  $4, $0, 42
  3: ori  $2, $0, 1
  4: syscall
  5: ori  $4, $0, 10
  6: ori  $2, $0, 11
  7: syscall
Output
Unknown system call: 4242
Registers After Execution
$2  = 4242
```

Syscall 11 should print the low byte (lowest 8 bits) of `$a0`.

Your submitted code must be C only. You may call functions from the standard C library (e.g., functions from `stdio.h`, `stdlib.h`, `string.h`, etc.) and the mathematics library (`math.h`). You may use `assert.h`.

You may not submit code in other languages. You may not use `system` or other C functions to run external programs. You may not use functions from other libraries; in other words, you cannot use dcc's `-l` flag.

If you need clarification on what you can and cannot use or do for this assignment, ask in the class forum.

You are required to submit intermediate versions of your assignment. See below for details.

Your program must not require extra compile options. It must compile with `dcc *.c -o smips`, and it will be run with `dcc` when marking. Run-time errors from illegal C will cause your code to fail automarking.

If your program writes out debugging output, it will fail automarking tests: make sure you disable debugging output before submission.

Change Log

Version 1.0
(2020-07-26 12:00:00)
- Initial release.

Version 1.1
(2020-08-02 10:00:00)
- Update reference implementation
- Clarify how to print invalid instruction codes

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest smips smips.c [other .c or .h files]
```

Assessment

Submission

When you are finished working on the assignment, you must submit your work by running `give`:

```
$ give cs1521 ass2_smips smips.c [other .c or .h files]
```

You must run `give` before **Tuesday December 31 23:59** to obtain the marks for this assignment. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

You can run `give` multiple times. Only your last submission will be marked.

If you are working at home, you may find it more convenient to upload your work via give's web interface.

You cannot obtain marks by e-mailing your code to tutors or lecturers.

You check the files you have submitted here.

Automarking will be run by the lecturer after the submission deadline, using test cases different to those `autotest` runs for you. (Hint: do your own testing as well as running `autotest`.)

Manual marking will be done by your tutor, who will mark for style and readability, as described in the **Assessment** section below. After your tutor has assessed your work, you can view your results here; The resulting mark will also be available via give's web interface.

Due Date

This assignment is tentatively due **Tuesday December 31 23:59**.

If your assignment is submitted after this date, each hour it is late reduces the maximum mark it can achieve by 2%. For example, if an assignment worth 74% was submitted 10 hours late, the late submission would have no effect. If the same assignment was submitted 15 hours late, it would be awarded 70%, the maximum mark it can achieve at that time.

Assessment Scheme

This assignment will contribute 15 marks to your final COMP1521 mark.

80% of the marks for assignment 2 will come from the performance of your code on a large series of tests.

20% of the marks for assignment 2 will come from hand marking. These marks will be awarded on the basis of clarity, commenting, elegance and style. In other words, you will be assessed on how easy it is for a human to read and understand your program.

An indicative assessment scheme follows. The lecturer may vary the assessment scheme after inspecting the assignment submissions, but it is likely to be broadly similar to the following:

HD (90+)	beautiful documented code, prints instructions correctly, output and registers correctly for all programs
CR/DN (70+)	very readable code, prints all instructions correctly, prints output and registers correctly for some programs
PS/CR (60+)	readable code, prints all instructions correctly
PS (50+)	gprints prints some instructions correctly

0%	knowingly providing your work to anyone and it is subsequently submitted (by anyone).
0 FL for COMP1521	submitting any other person's work; this includes joint work.
academic misconduct	submitting another person's work without their consent; paying another person to do work for you.

Intermediate Versions of Work

You are required to submit intermediate versions of your assignment.

Every time you work on the assignment and make some progress you should copy your work to your CSE account and submit it using the `give` command below. It is fine if intermediate versions do not compile or otherwise fail submission tests. Only the final submitted version of your assignment will be marked.

All these intermediate versions of your work will be placed in a Git repository and made available to you via a web interface at `https://gitlab.cse.unsw.edu.au/`*`z5555555`*`/20T2-comp1521-ass2_smips` (replacing *z5555555* with your own zID). This will allow you to retrieve earlier versions of your code if needed.

Attribution of Work

This is an individual assignment.

The work you submit must be entirely your own work, apart from any exceptions explicitly included in the assignment specification above. Submission of work partially or completely derived from any other person or jointly written with any other person is not permitted.

You are only permitted to request help with the assignment in the course forum, help sessions, or from the teaching staff (the lecturer(s) and tutors) of COMP1521.

Do not provide or show your assignment work to any other person (including by posting it on the forum), apart from the teaching staff of COMP1521. If you knowingly provide or show your assignment work to another person for any reason, and work derived from it is submitted, you may be penalized, even if that work was submitted without your knowledge or consent; this may apply even if your work is submitted by a third party unknown to you. You will not be penalized if your work is taken without your consent or knowledge.

Submissions that violate these conditions will be penalised. Penalties may include negative marks, automatic failure of the course, and possibly other academic discipline. We are also required to report acts of plagiarism or other student misconduct: if students involved hold scholarships, this may result in a loss of the scholarship. This may also result in the loss of a student visa.

Assignment submissions will be examined, both automatically and manually, for such submissions.
