Labo n° 4
Microprocessor Architectures [ELEC-H-473]
RiSC16: back to lab 1, correct your mistakes 4/3

2014–2015

This document/lab is about the code from Lab 1 multiplication evaluation but can be generalised to all exercises where code was required. An automatic verification tool is available to run your code and check results with some input vectors. In 2015, this tool was generalised to other exercises, to help you highlight your mistakes. During this lab about the RISC16, you can ask the assistant about your results from previous labs and you can improve your code to reach the "100% passed" goal. Section 2 will help you maximise your mark. Have fun.

We don't expect you to correct your code and rerun all proposed tests before the lab, but it might be useful to understand and identify any discrepancy between the test report and your own tests.

1 Lab 1 evaluation

Points awarded/signification:

1.1 16b SLL : 5pt

This exercise is removed from the quotation $\Longrightarrow$ 0 pt

- input: `reg5`, output: `reg5`

1: coding style (comments, readability)

-1: error in code, by type of error

4×% of passed tests (see test vectors below)

- `0x0000` $\longrightarrow$ `0x0000`
- `0x0001` $\longrightarrow$ `0x0002`
- `0x8FFF` $\longrightarrow$ `0xFFFE`
- `0x8000` $\longrightarrow$ `0x0000`

1.2 Most Significant Bit extraction : 5pt

- input: `reg1`, output `reg7`

1: coding style (comments, readability)

-1: error in code, by type of error

4×% of passed tests (see test vectors below)

- `0x0000` $\longrightarrow$ `0`
- `0x0001` $\longrightarrow$ `0`
- `0x8000` $\longrightarrow$ `1`
- `0xFFFF` $\longrightarrow$ `1`

1.3 32b SLL : 5pt

- input: `reg6`(MSB),`reg5`, output:`reg6`(MSB),`reg5`, `reg5` initialised from `reg1`

1: coding style (comments, readability)

-1: error in code, by type of error

4×% of passed tests (see test vectors below)

- `0x00000000` $\longrightarrow$ `0x00000000`
- `0x00000001` $\longrightarrow$ `0x00000002`
- `0x00008FFF` $\longrightarrow$ `0x0000FFFE`
- `0x00008000` $\longrightarrow$ `0x00010000`
- `0x80000000` $\longrightarrow$ `0x00000000`
- `0x80000001` $\longrightarrow$ `0x00000002`
- `0x0001FFFF` $\longrightarrow$ `0x0003FFFE`
- `0x00042000` $\longrightarrow$ `0x00084000`

1.4 16b+16b : 5pt

- input: `reg1`,`reg2`, output: `reg3`, carry: `reg4`

1: coding style (comments, readability)

-1: error in code, by type of error

4×% of passed tests (see test vectors below)

- `0x0000+0x0000 = 0, 0x0000`
- `0x0001+0x0000 = 0, 0x0001`
- `0x0001+0x0001 = 0, 0x0002`
- `0x8000+0x4000 = 0, 0xC000`
- `0x7FFF+0x7FFF = 0, 0xFFFE`

- `0xFFFF+0xFFFF = 1, 0xFFFE`
- `0xFFFF+0x0001 = 1, 0x0000`
- `0xFFFF+0x0002 = 1, 0x0001`

1.5 32b+32b : 5pt

- input: `reg4`(MSB), `reg3` and `reg6`(MSB), `reg5`, output: `reg4`(MSB), `reg3`

1: coding style (comments, readability)

-1: error in code, by type of error

4×% of passed tests (see test vectors below)

- `0x00000000+0x00000000 = 0x00000000`
- `0x00000001+0x00000000 = 0x00000001`
- `0x00000001+0x00000001 = 0x00000002`
- `0x00008000+0x00004000 = 0x0000C000`
- `0x00007FFF+0x00007FFF = 0x0000FFFE`
- `0x0000FFFF+0x0000FFFF = 0x0001FFFE`
- `0x0000FFFF+0x00000001 = 0x00010000`
- `0x0000FFFF+0x00000002 = 0x00010001`
- `0x7FFF1000+0x0000F001 = 0x80000001`
- `0xFFFFFFFF+0x00000001 = 0x00000000`
- `0xFFFFFFFE+0x00000001 = 0xFFFFFFFF`
- `0x7FFFFFFF+0x7FFFFFFF = 0xFFFFFFFE`

1.6 Multiplication : 20pt

- Originality removed
- Size of operands $\Longrightarrow$ 8 pt instead of 16.

- input: `reg1`, `reg2`, output: `reg4`(MSB), `reg3`

2: coding style (comments, readability)

-1: error in code, by type of error

2: Algorithm (efficiency)

2: ~~Originality~~ removed

0-4: Special bonus points (report, anything unusual/fancy)

1: Length of code:

$$\frac{21}{\text{number of instructions}}$$

21 is the shortest code giving the right result seen in the assignment so far.

1: Number of instructions to compute 0xff × 0xff:

$$\frac{233}{\text{number of instructions}}$$

0 if result is not correct.

233 is the lowest number of instructions giving the right result seen in the assignment so far.

6×% of passed tests (see test vectors below)

~~16~~ 8: Max size of operands/2 for x^2 (identified using test vectors below)

The mark is then expressed related to 20 using proportionality (result is integer only, rounded down, no floating point).

1.6.1 Test vectors for multiplication

All codes are automatically tested against:

- `0x0000 × 0x00ff =0x00000000`
- `0x00ff × 0x0000 =0x00000000`
- `0x7fff × 0x0007 =0x00037ff9`
- `0xffff × 0x0007 =0x0006fff9`
- `0xa060 × 0x88dc =0x55bcd280`
- `0x0001 × 0x0001 =0x00000001`
- `0x0003 × 0x0003 =0x00000009`
- `0x0007 × 0x0007 =0x00000031`
- `0x000f × 0x000f =0x000000e1`
- `0x001f × 0x001f =0x000003c1`
- `0x003f × 0x003f =0x00000f81`
- `0x007f × 0x007f =0x00003f01`
- `0x00ff × 0x00ff =0x0000fe01`
- `0x01ff × 0x01ff =0x0003fc01`
- `0x03ff × 0x03ff =0x000ff801`

- `0x07ff` $\times$ `0x07ff` `=0x003ff001`
- `0x0fff` $\times$ `0x0fff` `=0x00ffe001`
- `0x1fff` $\times$ `0x1fff` `=0x03ffc001`
- `0x3fff` $\times$ `0x3fff` `=0x0fff8001`
- `0x7fff` $\times$ `0x7fff` `=0x3fff0001`
- `0xffff` $\times$ `0xffff` `=0xfffe0001`

Other vectors can be added, ask the assistant if you need a specific one.

2 Code requirements/recommendations

This section is for the multiplication but you can generalise to the other exercises. Be sure your code:

1. uses `reg1` and `reg2` for operands. Anything else will cause the tests to fail (because the verification program assumes operands are in `reg1` and `reg2` and nowhere else). **Operands in correct registers**

2. uses `reg4` and `reg3` to store the result at the end of the execution. `reg3` is the Least Significant Word. Same note as previous point, result **must** be in `reg4-reg3`. **Result in correct registers**

3. initialises `reg1` and `reg2` using 2 `movi` instructions at the beginning (any other initialisation method will cause the automatic tests to fail). This is necessary to ensure that your code will work with the verification tool if you use `jalr` instructions for absolute jumps. These instructions will be replaced by `nop` instructions during automatic testing. **Use first instructions (2/4/8) to initialise your operands**

4. is writen knowing that the RAM is unaffected by any reset (you might get some bonus points by using this feature and creativity).

5. stops the simulator when the result is stored in `reg4-reg3` using a `halt` instruction. **End your code**

6. works with the test vectors. If the test report marks some tests as failed but your simulator works, check the code at the beginning of the report against your code and **then** ask the assistant. Don't ask the assistant before checking (friendly warning).

7. has some comments.

8. does not use any illegal instruction and/or instruction with too big immediate values.

9. is not too greedy ! (the verification tool stops any test after 10^7 instructions)

3 Test report details

The test report has several sections. An example is used to highlight them below[1].

3.1 Program processing

This part of the file shows how your program has been processed by the verification tool. If you see that your program has been misinterpreted, tell the assistant.

```
@   0 :           : lui 1,511
@   1 :           : addi 1,1,63
@   2 :           : lui 2,0
@   3 :           : addi 2,2,7
label : loop @4
@   4 : loop      : beq 5, 2, end
@   5 :           : add 3, 3, 1
@   6 :           : addi 5, 5, 1
@   7 :           : beq 0, 0, loop
label : end @8
@   8 : end       : halt
Instructions :  8
{'end': 8, 'loop': 4}
test_report.txt
```

3.2 Testing

This part shows the results of the tests using the vectors listed p.1.6.1. Tests are marked `passed` or `FAILED` and the number of instructions executed before the `halt` instruction is listed. When a test fails, the computed value is added in the report so you can check with your simulation.

```
Instructions 0 to 3 changed to 'nop', let's test ...
0x0000 x 0x00ff =0x00000000, passed (instr=   1025)
0x00ff x 0x0000 =0x00000000, passed (instr=      5)
0x7fff x 0x0007!=0x00037ff9(=0x00007ff9), FAILED!!!   (instr=     33)
0xffff x 0x0007!=0x0006fff9(=0x0000fff9), FAILED!!!   (instr=     33)
0xa060 x 0x88dc!=0x55bcd280(=0x0000d280), FAILED!!!   (instr= 140149)
0x0001 x 0x0001 =0x00000001, passed (instr=      9)
0x0003 x 0x0003 =0x00000009, passed (instr=     17)
0x0007 x 0x0007 =0x00000031, passed (instr=     33)
0x000f x 0x000f =0x000000e1, passed (instr=     65)
0x001f x 0x001f =0x000003c1, passed (instr=    129)
0x003f x 0x003f =0x00000f81, passed (instr=    257)
0x007f x 0x007f =0x00003f01, passed (instr=    513)
0x00ff x 0x00ff =0x0000fe01, passed (instr=   1025)
0x01ff x 0x01ff!=0x0003fc01(=0x0000fc01), FAILED!!!   (instr=   2049)
0x03ff x 0x03ff!=0x000ff801(=0x0000f801), FAILED!!!   (instr=   4097)
0x07ff x 0x07ff!=0x003ff001(=0x0000f001), FAILED!!!   (instr=   8193)
0x0fff x 0x0fff!=0x00ffe001(=0x0000e001), FAILED!!!   (instr=  16385)
0x1fff x 0x1fff!=0x03ffc001(=0x0000c001), FAILED!!!   (instr=  32769)
0x3fff x 0x3fff!=0x0fff8001(=0x00008001), FAILED!!!   (instr=  65537)
0x7fff x 0x7fff!=0x3fff0001(=0x00000001), FAILED!!!   (instr= 131073)
0xffff x 0xffff!=0xfffe0001(=0x00000001), FAILED!!!   (instr= 262145)
```

[1]Any similarities with code read in the lab assignments is a pure coincidence

3.3 Errors

The verification tool checks operands sizes and reports any error in the report file with the address causing the trouble. This is an experimental feature.

```
error , immediate too big@ 15 32768
error , immediate too big@ 15 32768
error , immediate too big@ 15 32768
error , immediate too big@ 15 32768
```

Sometimes, a crash log from Python is also present in the script. This means your code made the verification tool crash, depending on what caused the crash, it might be bad for your mark.

3.4 Summary

At the end of the file, the number of passed tests and the size of the biggest possible operand for reliable multiplication using your code are listed.

```
Tests : 21,  10 passed => 47.6% passed
Argument size : 8 bits
```

3.5 Disclaimer

The verification tool has been developed last year[2] and still has some bugs[3]. If your simulation does not match the report:

1. Check the beginning of the report and verify that your code has been correctly interpreted. Especially, if you used `jalr` instructions, identify any unexpected offset.
2. Sometimes, I changed some instructions because of an error (operands too big), well, that means your code had problems to begin with...
3. If there is a python crash log, try to identify if something in your code caused it (mistyped instruction, unexpected literal...).
4. Check with the assistant what went wrong, maybe your code is correct and the verification tool missed something...
5. If you don't understand something, ask the assistant.
6. If you want some feature to be added to the verification tool, ask the assistant.
7. Help the assistant improve the lab: report mistakes...

4 Assignment

Send by email to the assistant any improved code for testing. You can do it 10 times before the lab, 4 times during the lab and 3 times before the deadline. Don't expect to get the test report in less than a day if you send it before/after the lab.

[2] 2014
[3] Any bug report will be appreciated