Lab 1–2
Microprocessor Architectures [ELEC-H-473]
RiSC16: internal processor behaviour 1–2/4

2015–2016

Introduction

This lab is about the internal behaviour of a RISC processor and goals are to:

- understand the internal behaviour of a simple processor
- write and test some programs in assembly code for this specific RISC processor
- watch this code running in simulation

You will use a RiSC16 processor which is a RISC processor developed for teaching by Bruce Jacob (University of Maryland) for a course about microelectronics. This processor based on a Harvard architecture works on 16-bit data and instructions. It has 8 instructions, one 8×16b-register bank, 256-word RAM and ROM are available.

The simplicity of the instruction set allows a quick learning curve and is enough to address complex problems. The internal structure of the processor is simple enough to be represented in a didactic way like on Figure 1.

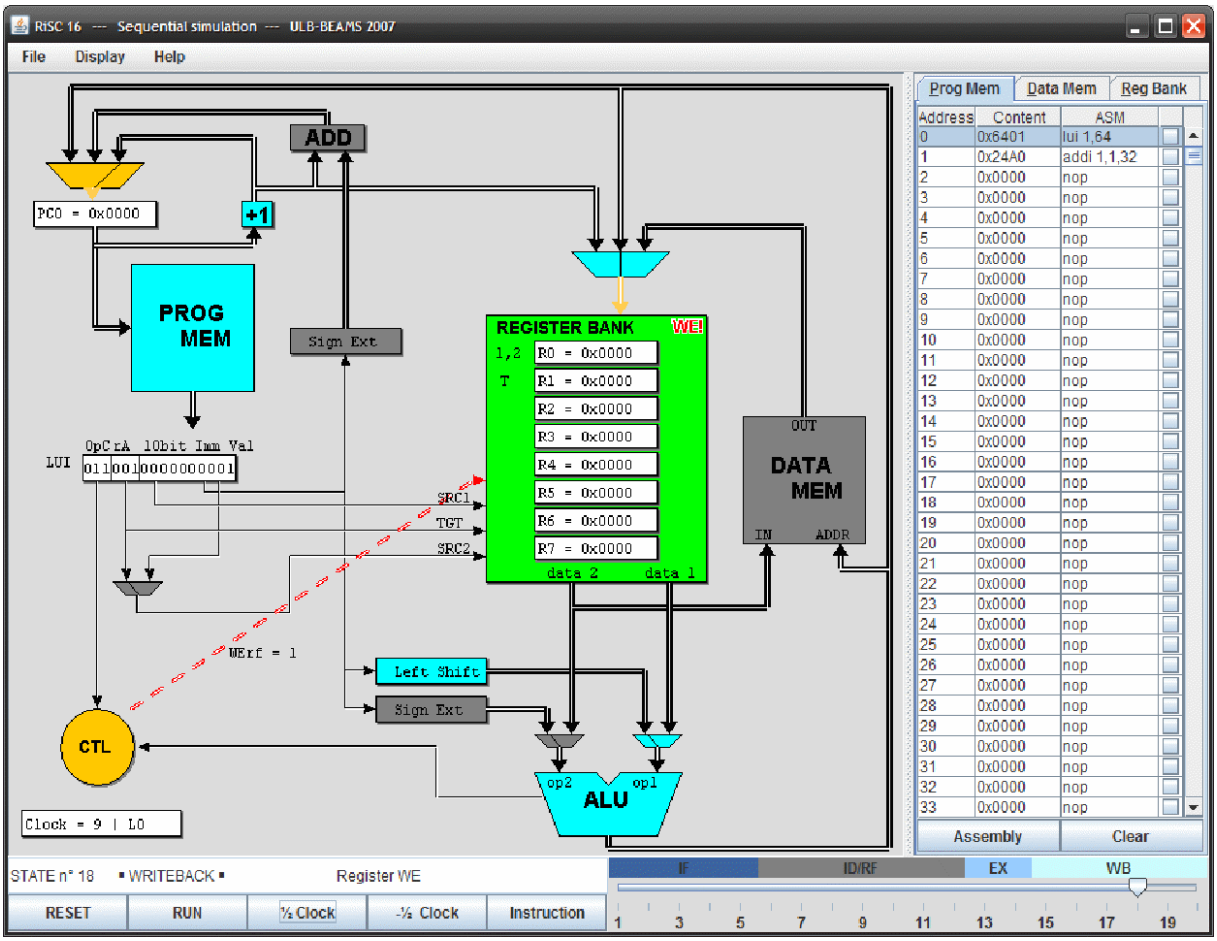


Figure 1: RiSC16 – Simulation view

The user interface has 4 zones:
- the lower part groups the interaction possibilities: buttons provide *reinitialisation* (reset), *execution* (run), *half-cycle execution*, *instruction execution* and *undo last instruction* capabilities.
- The text zone above displays general informations about instruction execution. The cursor on the right can be used to follow each step of instruction execution and can be used to jump at any step of this instruction.
- The zone on the right shows and allows editing program and data memory. A third tab shows register, they can also be modified.
- The central zone shows the microprocessor representation

1 Processor description

Several blocks are used in this microprocessor:
- Program memory (`PROG MEM`) and Data memory (`DATA MEM`) – Harvard architecture
- Internal register bank (`Register Bank`) to store operands and computation results. Register 0 (`reg0`) has a special meaning: it is a read only register containing the `0` value. A write in `reg0` has no effect.
- The Arithmetic and Logic Unit (ALU) is used to execute the operations needed to execute instructions. Complement to 2 arithmetic is used. 4 operations are possible:
  1. add two operands, used during `ADD, ADDI, LW, SW`
  2. bitwise NAND of two operands, used obviously during `NAND`
  3. no modification, result is `OP1`, used during `LUI, JALR`
  4. operand comparison, used during `BEQ`
- The Control Unit (`CTL`) decodes the opcode and configure other blocks accordingly. It's synchronised on the clock and can be used for rising edge or falling edge operation.
- The program counter `PC0`
- The instruction register
- The incrementer `+1`
- The adder (`ADD`) to compute the address for jumps (Branch with `BEQ`)
- The immediate value converter (left shift and sign extension)

Blocks are linked together using buses with width of 1, 3, 6, 10 or 16 bits. Control signals are:
- `WE_rf`: write in register bank
- `FUNC_alu`: operation to use
- `Mux_rf`, `Mux_alu` (1, 2), `Mux_tgt`, `Mux_pc` : select multiplexer input
- `WE_dmem`: write in data memory
- `PSEN` : read from program memory.
- `WE_PC0` : write data into `PC0`.

The instruction set has 8 elementary instructions. None of them can be replaced by any combination of other instructions and complex problems can be solved using only these 8 instructions. This architecture is RISC at its fate.

1.1 Instructions detailed description

Each instruction and how to code it is described below.

1.1.1 ADD : $R[\text{regA}] \longleftarrow R[\text{regB}] + R[\text{regC}]$

bits	3	3	3	4	3
ADD	000	regA	regB	0000	regC

Adds content from `regB` with content from `regC` and writes result in `regA`.

1.1.2 ADDI (ADD immediate) : $R[\text{regA}] \longleftarrow R[\text{regB}] + \text{immed}$

bits	3	3	3	7
ADDI	001	regA	regB	signed immediate

Adds content from `regB` with an immediate 7-bit signed constant (-64 to 63) extended to 16-bits. Writes result in `regA`.

1.1.3 NAND : $R[\text{regA}] \longleftarrow \text{NOT}(R[\text{regB}] \,\&\, R[\text{regC}])$

bits	3	3	3	4	3
NAND	010	regA	regB	0000	regC

Bitwise NAND using content from `regB` and `regC`. Writes result in `regA`.

1.1.4 LUI (LoadUpperImmediate) : $R[\text{regA}] \longleftarrow \text{immed} << 6 \,\&\, 0\text{xFFC0}$

bits	3	3	10
LUI	011	regA	immediate (0 to 0x3FF)

Immediate 10-bit constant sign extended and written in `regA`.

1.1.5 LW (Load Word) : $R[\text{regA}] \longleftarrow \text{Mem}[R[\text{regB}] + \text{immed}]$

bits	3	3	3	7
LW	100	regA	regB	signed immediate

Reads a word in memory at address (`regB`+immediate). Immediate 7-bit constant is sign extended to 16-bit before the addition. This is an indirect addressing with offset.

1.1.6 SW (Store Word) : $R[\text{regA}] \longrightarrow \text{Mem}[R[\text{regB}] + \text{immed}]$

bits	3	3	3	7
SW	101	regA	regB	signed immediate

Writes the content of `regA` to memory at address (`regB`+immediate). Immediate 7bit constant is sign extended to 16bit before the addition. This is an indirect addressing with offset.

1.1.7 BEQ (Branch if EQual) :

if $R[\text{regA}] == R[\text{regB}]$ $\{\text{PC} \longleftarrow \text{PC} + 1 + \text{immed}\}$ else $\{\text{PC} \longleftarrow \text{PC} + 1\}$

bits	3	3	3	7
BEQ	110	regA	regB	signed immediate

Compares content of `regA` and `regB`. If equal, PC is updated to $\text{PC}_{\text{BEQ}} + 1 + \text{immed(extended)}$ else `PC` is just incremented by 1.

1.1.8 JALR (Jump And Link using Register) : $\text{PC} \longleftarrow R[\text{regB}], R[\text{regA}] \longleftarrow \text{PC} + 1$

bits	3	3	3	7
JALR	111	regA	regB	0

Jumps to address stored in `regB`. Writes `PC+1` in `regA`. This is a call with saved return address.

1.2 Pseudo Instructions

Other (pseudo-)instructions can be used:

1.2.1 `NOP`

```
NOP = ADD 0,0,0
```

This instruction does nothing because writing to `reg0` has no effect (`reg0` is *read only*).

1.2.2 `RESET`

```
RESET = JALR 0,0
```

This pseudo instruction changes `PC` to `0` and thus resets the processor. Registers and data memory remain unchanged.

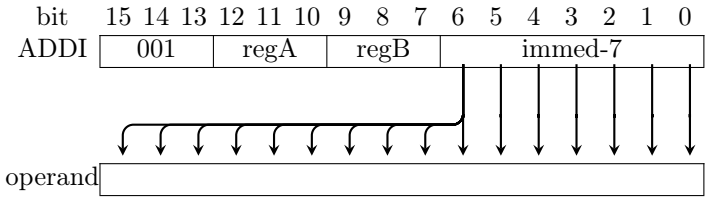
1.2.3 `MOVI`

`MOVI rx,imm(16bits) = LUI rx, immH(10bits); ADDI rx,rx,immL(6bits)`

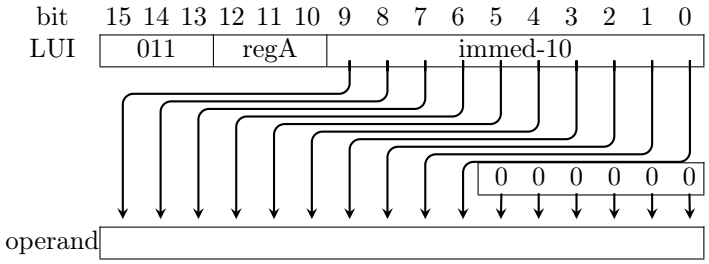
This pseudo instruction can be used to load a 16-bit immediate constant to `reg(rx)`. Two instructions are used to perform this pseudo instruction. `LUI` loads the upper 10 bits and `ADDI` loads the 6 lower bits. To avoid any unexpected side effect during assembly, be sure to add a `NOP` after this instruction. The `NOP` will be replaced by the `ADDI` instruction.

Depending on their size, immediate constants are extended to 16 bits:

– Immediate 7bit signed constants (from -64 to 63) are sign-extended to 16 bits before they are used.



– Immediate 10bit constants are unsigned numbers from 0 to 1023.



1.2.4 `HALT`

This pseudo instruction stops the simulator.

1.3 Instruction execution

All instructions are executed in 4 stages:

1. `IF` *Instruction Fetch*: instruction is copied from the program memory to the instruction register
2. `ID/RF` *Instruction Decode/Register Fetch*: instruction decoding and operand extraction
3. `EX` *Execute*: instruction execution in the ALU
4. `WR` *Write Back*: result saving or data memory access

As the RiSC-16 is a RISC architecture[1], all instructions have the same execution time, which is 20 half clock-cycles. Control signals are activated considering the slowest instruction, which is `BEQ`. The details of a machine cycle (execution of one instruction) are detailed in Table 1 on the next page.

[1] Incredible, isn't it?

Stage:		IF		ID/RF				EX	WB				
Half-cycle:		1–2–3	4–5	6–7	8	9	10–11–12	13–14	15–16	17	18	19	20
ADD	000	ROM	IR	CTL+RF(src1)	CTL+Mux_RF+RF(src1)	Mux_RF	RF(src2)	ALU		Mux_PC	RF+Mux_PC	RF+PC	RF+PC
ADDI	001	ROM	IR	CTL+RF(src1)+Sign_ext	CTL+RF(src1)			ALU		Mux_PC	RF+Mux_PC	RF+PC	RF+PC
NAND	010	ROM	IR	CTL+RF(src1)	CTL+Mux_RF+RF(src1)	Mux_RF	RF(src2)	ALU		Mux_PC	RF+Mux_PC	RF+PC	RF+PC
LUI	011	ROM	IR	CTL+Left_Shift	CTL			ALU		Mux_PC	RF+Mux_PC	RF+PC	RF+PC
LW	100	ROM	IR	CTL+RF(src1)+Sign_ext	CTL+RF(src1)			ALU	datam	datam+Mux_PC	RF+Mux_PC	RF+PC	RF+PC
SW	101	ROM	IR	CTL+RF(src1)+Sign_ext	CTL+Mux_RF+RF(src1)	Mux_RF	RF(src2)	ALU	datam	datam+Mux_PC	Mux_PC	PC	PC
BEQ	110	ROM	IR	CTL+RF(src1)+Sign_ext	CTL+Mux_RF+RF(src1)+ADD	Mux_RF+ADD	RF(src2)	ALU	CTL	CTL+Mux_PC	Mux_PC	PC	PC
JALR	111	ROM	IR	CTL+RF(src1)	CTL+RF(src1)			ALU		Mux_PC	RF+Mux_PC	RF+PC	RF+PC

Table 1: Detailed instruction processing

2 Simulation dynamics

Most elements are asynchronous, which means that no register buffers data between blocks but the control unit:

- is synchronous to the clock
- provides control signal `WE` just after clock rising edges for write operations to the `PC`, to the register bank and to the data memory.
- provides a read in ROM signal `PSEN` just after a clock rising edge, which provides a synchronisation on instruction reading. An instruction is extracted each 20 $\frac{1}{2}$cycles precisely.

The simulator aims at exposing in detail the internal processor behaviour during execution. To highlight and ease understanding, three colors are used for different states:

- Green: used to show that the block is busy: input data is stable and the block is processing them. This transient state ends after the propagating time in the block. This state is irrelevant for busses.
- Orange: this states follows the previous one. The block has processed data and the result is available. Output signals also take that stable state.
- Default color: this color is used for "has been used" and for default state. Default state means that the block has not yet received data for the current instruction. State "has been used" means that the block has provided its result to the other blocks and will not do anything until the next instruction.

Figure 2 shows the different states and the associated colors. Lines represent output buses for a specific block.

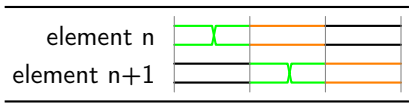


Figure 2: Transitional, stable and "Has been used" states.

After opcode decoding, unused blocks in the instruction will be greyed. See `ADD` and DATA MEM on Figure 1 on page 1. It's not a specific state but it is used to de-emphasize unused blocks to ease user thoughts by highlighting only relevant blocks.

Any greyed block will provide a signal computed from its input but the actual value is useless, and so is not displayed.

The simulation might show a synchronous behaviour because of instantaneous color change and aligned on clock edges to ease the simulator design and thus does not represent the behaviour of an actual implementation of the RiSC16. Most blocks have a propagating time lower than a half-period, as shown on Figure 2.

3 The Simulator

The assembly program can be loaded into the simulator from two sources:

- Writing instruction directly in the ASM column in "Program memory" window. To update the program memory with the modified instructions, use the "assembly" button. The pseudo instruction `MOVI` will be replaced by two instructions so an empty line must follow this instruction.
- Importing an external file using the "File import ROM" menu.

Several features are available when editing your program in an external file:

- Comments: anything after "//" on a single line
- Any hexadecimal constant must begin with the "`0x`" notation.
- Place instruction at specific address: "@" followed by constant value must be added before the instructions
- Labels: Labels are defined using a name+":" at the beginning of a line

Syntax for instructions is defined as:

label:<space>opcode<space>field1, field2, filed3<space>// comments

Where:

- label: optional label
- // comments: optional comment
- opcode: opcode of the instruction (mandatory)
- **field1**, 2, 3: fields that can be register numbers, constants (instruction dependant)

The program is assembled when the file is imported (menu "File-Import ROM"). A program modified inside the program memory can also be saved to a file. Several files can be imported. Overlapping memory segments have the value of the last imported file. Importing and exporting RAM is also possible.

An example is given in Listing 1.

```
1                   addi    2,0,1
                    sw      2,1,0
                    addi    1,1,1
                    sw      2,1,0
                    addi    1,1,1
6                   add     3,2,2
                    sw      3,1,0
                    addi    1,1,1
                    addi    7,0,7
loop:               beq     7,0,end
11                  lw      2,1,-2
                    add     3,3,2
                    sw      3,1,0
                    addi    1,1,1
                    addi    7,7,-1
16                  beq     0,0,loop
end:                halt
```

Listing 1: Example code

Known bugs: To update labels, save and import your file. Don't forget to "assembly" after code changes.

4 Manipulation

Question 1. Explain what is the example on listing 1 on the preceding page doing? Detail the state of registers and the state of the `PC` after each instruction.

Question 2. Load `exemple1.txt` and run the simulation. Explain the internal behaviour for each instruction.

Question 3. Using the graph in annexe 3 on page 13, draw the chronogram for the `BEQ` instruction. Signals on the graph are output of blocks of the processor.

For each of the exercises 4–9, you can automagically test your code against a set of *test vectors* (listed in section 6) using the "**Online Verification Tool®©™**" at address:

http://164.15.75.4:8080

Read this handout to the end to get a glimpse on its capabilities and to understand how to use it.

Question 4. Write[2] a program which shifts to the left the content of `reg5`.

Question 5. Write[2] a program which extracts the most significant bit from `reg1` and stores the value (0/1) in `reg7`.

Question 6. Write[2] a program which shifts to the left a 32-bit value stored in `reg6`(MSB), `reg5`.

Question 7. Write[2] a program which adds the unsigned content of `reg1` and `reg2` and writes the result in `reg3` and the carry bit in `reg4`.

Question 8. Write[2] a program which adds the unsigned content of 32-bit numbers stored in `reg4`(MSB), `reg3` and `reg6`(MSB), `reg5` and writes the result in `reg4`(MSB), `reg3`.

Question 9. Write[2] a program which multiplies the (unsigned) content of `reg1` and `reg2` and writes the result into `reg4`(MSB), `reg3`.

5 Code requirements/recommendations

This section is for the multiplication but you can generalise to the other exercises according to the question formulation. 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)

[2] And test it using the "Online Verification Tool®©™", some results will be included in the quotation of these labs, read further for more details.

6 Test vectors

This sections lists some usual test vectors to test the code you are writing for the different exercises. They are all integrated in the "Online Verification Tool®©™".

6.1 16b SLL

input: `reg5`, output: `reg5`
- `0x0000` ⟶ `0x0000`
- `0x0001` ⟶ `0x0002`
- `0x8FFF` ⟶ `0xFFFE`
- `0x8000` ⟶ `0x0000`

6.2 Most Significant Bit extraction

input: `reg1`, output `reg7`
- `0x0000` ⟶ `0`
- `0x0001` ⟶ `0`
- `0x8000` ⟶ `1`
- `0xFFFF` ⟶ `1`

6.3 32b SLL

input: `reg6`(MSB), `reg5`, output: `reg6`(MSB), `reg5`
- `0x00000000` ⟶ `0x00000000`
- `0x00000001` ⟶ `0x00000002`
- `0x00008FFF` ⟶ `0x0000FFFE`
- `0x00008000` ⟶ `0x00010000`
- `0x80000000` ⟶ `0x00000000`
- `0x80000001` ⟶ `0x00000002`
- `0x0001FFFF` ⟶ `0x0003FFFE`
- `0x00042000` ⟶ `0x00084000`

6.4 16b+16b

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

6.5 32b+32b

This code will be evaluated using these elements:
- input: `reg4`(MSB), `reg3` and `reg6`(MSB), `reg5`; output: `reg4`(MSB), `reg3`
- coding style (comments, readability)
- errors in code
- % of passed tests (see test vectors below)

The mark will be expressed related to 2 (result is integer only, rounded down).

6.5.1 Test vectors for addition

- `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`

6.6 Multiplication

This code will be evaluated using these elements:
- input: `reg1`, `reg2`; output: `reg4`(MSB), `reg3`
- coding style (comments, readability)
- errors in code
- Algorithm (efficiency)
- Length of code
- number of instructions to compute 0xff × 0xff
- % of passed tests (see test vectors below)
- Max size of operands for x^2 (identified using test vectors below)

The mark will be expressed related to 8 (result is integer only, rounded down).

6.6.1 Test vectors for multiplication

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

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

7 Test report details

The test report has several sections. An example is used to highlight them below[3].

7.1 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
```

7.2 Testing

This part shows the results of the tests using the vectors listed p.10. 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)
```

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

7.4 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
```

7.5 Disclaimer

The verification tool has still some bugs[4]. 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. If there is a python crash log, try to identify if something in your code caused it (mistyped instruction, unexpected literal...).
3. Check with the assistant what went wrong, maybe your code is correct and the verification tool missed something...
4. If you don't understand something, ask the assistant.
5. If you want some feature to be added to the verification tool, ask the assistant.
6. Help the assistants improve the lab: report mistakes, bugs...

8 Assignment

Send by email to your assistant codes for 32b addition and multiplication.

[4] Any bug report will be appreciated

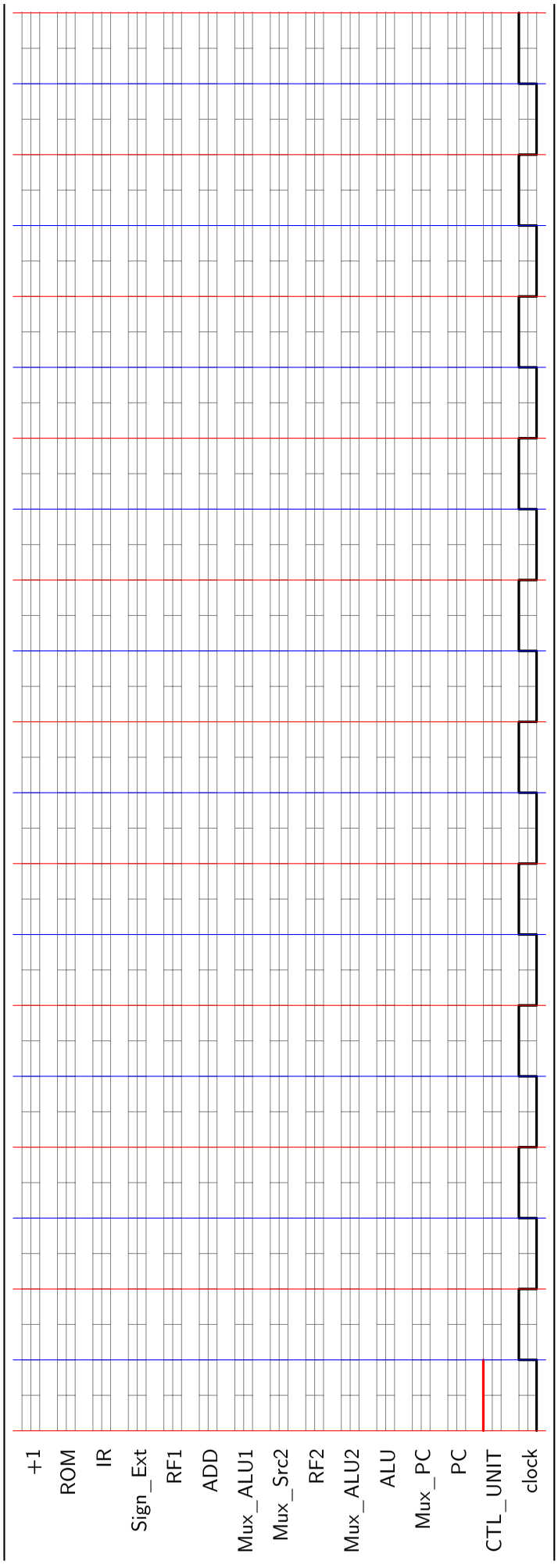


Figure 3: BEQ chronogram