Labo n° 3
Microprocessor Architectures [ELEC-H-473]
RiSC16: internal processor behaviour 3/3

2012–2013

Introduction

This last lab about the RiSC16 will highlight limitations of the RiSC16's 8-instruction instruction set. The simulator you will use in this lab will enable you to define a new architecture with additional instructions and additional registers.

Processor performances can be characterised by the time (t) required to execute a program and depends on:

- the number of instruction executed CI
- the average number of cycles per instruction CPI
- the clock frequency f

Thus the equation is:

$$\begin{aligned} t &= \frac{1}{\text{performance}} \\ &= (\text{instruction number}) \cdot (\text{cycles per instruction}) \cdot (\textit{clock period}) \\ &= \frac{\text{CI} \cdot \text{CPI}}{f} \end{aligned}$$

Possibilities to improve performance are:

- increase clock frequency (f)
- change processor internal design to lower CPI
- optimise the compiler to reduce instruction number (CI) or decrease CPI
- extend the instruction set to decrease CI

In this lab, we will explore the last possibility and thus will extend the RiSC16 instruction set to 16 instructions.

1 New instructions

The 8-instruction set will be extended with 8 new instructions. As there are now 16 instructions, 4 bits must be used to code the opcode instead of 3. Instructions must be coded on 17 bits. This instruction set is detailed in Table 1 on the following page.

The number of registers and the number of bits used for immediate constants can be adapted to make several flavours of the same processor. A variant using 24-bit instructions is defined in the simulator as in the Table 2 on the next page.

Instr \ bit	16–13	12–10	9–7	6–3	2–0
ADD	0000	reg A	reg B	(-8 to 7)	reg C
SUB	0001	reg A	reg B	(-8 to 7)	reg C
NAND	0010	reg A	reg B	0000	reg C
LUI	0011	reg A	immediate 0 to 0x3FF		
SHL	0100	reg A	reg B	(-8 to 7)	reg C
SHA	0101	reg A	reg B	(-8 to 7)	reg C
NOR	0110	reg A	reg B	0000	reg C
ADDI	1000	reg A	reg B	signed immediate (-64 to 63)	
SHIFTI	1001	reg A	reg B	signed immediate (-64 to 63)	
BL	1010	reg A	reg B	signed immediate (-64 to 63)	
BG	1011	reg A	reg B	signed immediate (-64 to 63)	
LW	1100	reg A	reg B	signed immediate (-64 to 63)	
SW	1101	reg A	reg B	signed immediate (-64 to 63)	
BEQ	1110	reg A	reg B	signed immediate (-64 to 63)	
JALR	1111	reg A	reg B	signed immediate (-64 to 63)	

Table 1: Special IS[1]

Instr \ bit	23–20	19–16	15–12	11–4	3–0
ADD	0000	reg A	reg B	(-128 to 127)	reg C
SUB	0001	reg A	reg B	(-128 to 127)	reg C
NAND	0010	reg A	reg B	00000000	reg C
LUI	0011	reg A	immediate 0 to 0xFFFF		
SHL	0100	reg A	reg B	(-128 to 127)	reg C
SHA	0101	reg A	reg B	(-128 to 127)	reg C
NOR	0110	reg A	reg B	00000000	reg C
ADDI	1000	reg A	reg B	signed immediate (-2048 to 2047)	
SHIFTI	1001	reg A	reg B	signed immediate (-2048 to 2047)	
BL	1010	reg A	reg B	signed immediate (-2048 to 2047)	
BG	1011	reg A	reg B	signed immediate (-2048 to 2047)	
LW	1100	reg A	reg B	signed immediate (-2048 to 2047)	
SW	1101	reg A	reg B	signed immediate (-2048 to 2047)	
BEQ	1110	reg A	reg B	signed immediate (-2048 to 2047)	
JALR	1111	reg A	reg B	signed immediate (-2048 to 2047)	

Table 2: Special IS[2] 16 reg 24-bit instructions

1.1 Arithmetic instructions

Defined in IS[1] et IS[2]

1.1.1 SUB (Substraction) : $R\,[\text{regA}] \longleftarrow R\,[\text{regB}] - R\,[\text{regC}]$

Substract content of `regC` from `regB` and write the result in `regA`.

Defined in IS[2] only

1.1.2 Mul (Multiplication) :
$R\,[\text{regA-1}] \longleftarrow (R\,[\text{regB}] * R\,[\text{regC}]) \gg 16,$
$R\,[\text{regA}] \longleftarrow (R\,[\text{regB}] * R\,[\text{regC}])\,\%\,2^{16}$

Multiply the content of `regB` with content of `regC` and write the 16 LSB[1] to `regA` and the 16 MSB to `regA-1`. Its a big endian[2] representation: most significant bits are at the lowest address. Only present in IS[2].

1.2 Logic instructions

Defined in IS[1] et IS[2]:

1.2.1 NOR : $R\,[\text{regA}] \longleftarrow \text{NOT}\,(R\,[\text{regB}] \mid R\,[\text{regC}])$

Bitwise NOR between content of `regB` and content of `regC`. Result is written in `regA`.

1.2.2 XOR (eXclusive OR) : $R\,[\text{regA}] \longleftarrow (R\,[\text{regB}] \wedge R\,[\text{regC}])$

Bitwise XOR between content of `regB` and content of `regC`. Result is written in `regA`.

Available using "Architecture > Instruction Set > Other":

1.2.3 OR : $R\,[\text{regA}] \longleftarrow R\,[\text{regB}] \mid R\,[\text{regC}]$

Bitwise OR between content of `regB` and content of `regC`. Result is written in `regA`.

1.2.4 XNOR (eXclusive NOR) : $R\,[\text{regA}] \longleftarrow \text{NOT}\,(R\,[\text{regB}] \wedge R\,[\text{regC}])$

Bitwise XOR between content of `regB` and content of `regC`. Result is written in `regA`.

1.2.5 AND : $R\,[\text{regA}] \longleftarrow R\,[\text{regB}]\,\&R\,[\text{regC}]$

Bitwise AND between content of `regB` and content of `regC`. Result is written in `regA`.

1.3 Branch instructions

Defined in IS[1] et IS[2]

1.3.1 BL (Branch if Lower) :
if $(R\,[\text{regA}] < R\,[\text{regB}])\,\{\text{PC} \longleftarrow \text{PC} + 1 + \text{immed}\}$ else $\{\text{PC} \longleftarrow \text{PC} + 1\}$

Compare content of `regA` with content of `regB`, if `regA` lower than `regB` then
$\text{PC} = \text{PB}_{BL} + 1 + \text{imm(extend)}$ else $\text{PC} = \text{PC}_{BL} + 1$.

Defined in IS[1], replaced by `MUL` IS[2]

1.3.2 BG (Branch if Greater) :
if $(R\,[\text{regA}] > R\,[\text{regB}])\,\{\text{PC} \longleftarrow \text{PC} + 1 + \text{immed}\}$ else $\{\text{PC} \longleftarrow \text{PC} + 1\}$

Compare content of `regA` with content of `regB`, if `regA` greater than `regB` then
$\text{PC} = \text{PB}_{BL} + 1 + \text{imm(extend)}$ else $\text{PC} = \text{PC}_{BL} + 1$.

[1]Least Significant Bits

[2]Endianness is a reference to *Johnatan Swift*'s *"Gulliver's Travels"* about a fight between Lilliput and Blefuscu about which end of a soft-boiled egg –big or small– should be cracked.

1.4 Shift instructions

Shift instructions can be used to multiply (shift left) or divide (shift right) very quickly by a 2^n number. The ALU must be modified to implement a Barrel Shifter to provide this feature.

Defined in IS[1] et IS[2]

1.4.1 SHL (Shift Logical) : $R[\text{regA}] \longleftarrow R[\text{regB}] \ll R[\text{regC}]$ or $R[\text{regB}] \gg R[\text{regC}]$

Shift to the left if the content of `regC` is positive, else shift to the right. The content of `regB` is shifted by the content of `regC` bits and the result is written in `regA`.

1.4.2 SHA (Shift Arithmetic) : $R[\text{regA}] \longleftarrow R[\text{regB}] \ll R[\text{regC}]$ or $R[\text{regB}] \gg R[\text{regC}]$

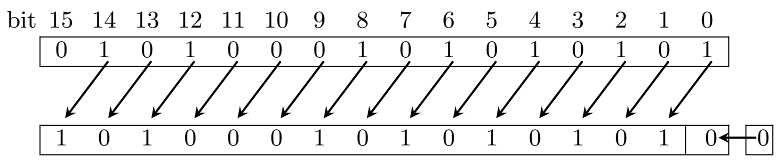
The content of `regB` is shifted by the content of `regC` bits and the result is written in `regA`. Shift to the left if the content of `regC` is positive, else shift to the right. If `regA` is shifted to the **right**, the sign bit is duplicated.

1.4.3 SHIFTI (Shift Immediate) : $R[\text{regA}] \longleftarrow R[\text{regB}] \ll \text{immed}$ or $R[\text{regB}] \gg \text{immed}$

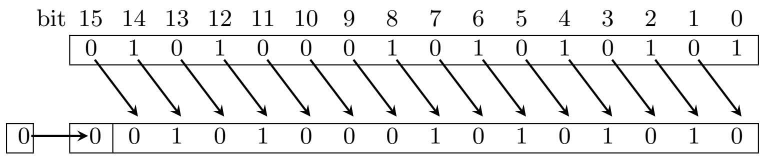
The content of `regB` is shifted by immediate constant value bits and the result is written in `regA`. Shift to the left if the constant is positive, else shift to the right. If `regA` is shifted to the **right**, the sign bit is duplicated. The 7 bit constant uses the least significant 5 bits as the immediate constant, the sixth as the mode (0: logic, 1: arithmetic), the seventh is unused.

bit:	6	5	4	3	2	1	0
	0	M	-16 to 15				

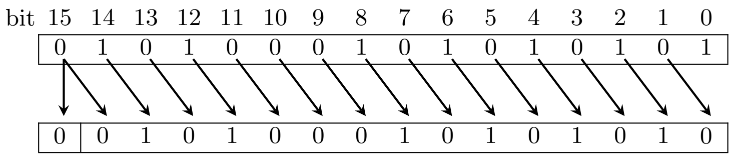
The different shifts are detailed on Figure 1.



(a) 1-bit left shift (arithmetic and logic)



(b) 1-bit right shift, logic



(c) 1-bit right shift, arithmetic

Figure 1: Arithmetic and logic shifts

1.5 Overflow management

In the basic architecture, nothing prevents overflow to happen, and nothing can be used to detect them. In the new architectures, overflows can be detected and processed. The mechanism used adds a meaning to unused bits in `RRR` instructions to branch when an overflow occurs, see Table 1 on page 2, bits 6 to 3 for `ADD`, `SUB`, `SHA`, `SHL`. These 4 bits can be used to make a relative jump between -8 and 7 in the program memory, thus is usually sufficient to write an exception routine.

To use this overflow management, the relative jump offset must be added at the end of the instruction. As for branch instructions, a label can be used. Example :

```
ADD 3, 1, 2, [immed] //add reg1 and reg2, jump to immed in case of overflow
ADD 3, 1, 2, -8      //in case of overflow, PC=PC+1-8
ADD 3, 1, 2, label   //in case of overflow, jump to label
```

Listing 1: Examples

The menu “Architecture > Signed or Unsigned” allows to define if the branch must happen when *overflow* occurs in signed arithmetic or when *carry* happens in unsigned arithmetic.

2 Simulator interface presentation

The simulator window (see Figure 2 on the following page) has a syntax highlighting editor. Labels, addresses, comments and instructions has their how style. Instructions from the original RiSC16 instruction set have a different style from instructions added afterwards. This highlighting is helpful to check code validity.

Once the program is written in assembly code, the “Run” button launch the simulation environment. New windows are added:

- Program memory: ASM code and program memory are visible there
- Data Memory
- Register bank content
- Simulation state: execution, trace and statistics

The third first windows are the same as in the previous simulators.

The last window has controls for:

- PLAY : run the program until a breakpoint or the halt instruction is reached
- STOP: stop the simulation[3]
- NEXT: step by step mode
- RESET: `PC` is reset to 0, registers and data memory remain unchanged
- SAVE: save the trace content into a text file, statistics are also saved.
- CPI: Cycles Per Instruction
- RAW Stall: number of data hazards due to `LW` instructions
- Branch stall: number of control conflicts, *i.e.,* number of jumps
- Speedup: acceleration factor. Shows how many times the pipelined version is faster than an hypothetical version using 70 half clock cycles sequencing (5 stages of 14 half-cycles)

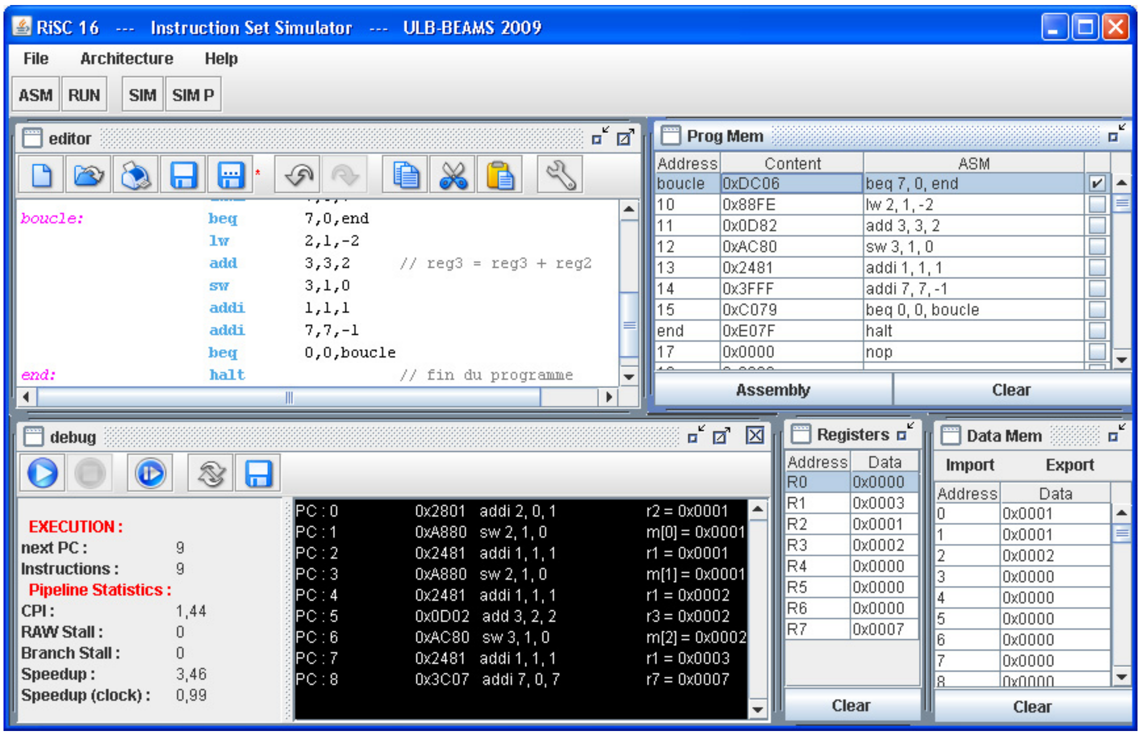
$$\text{speedup} = \frac{\text{pipeline depth}}{\text{CPI}}$$

- Speedup(clock): practical gain including the cycle length in the sequential version (10 clock cycles) *vs* the pipeline version (7 clock cycles)

$$\frac{T_{\text{seq}}}{T_{\text{pipe}} \cdot \text{CPI}} = \frac{10}{7 \cdot \text{CPI}}$$

Last but not least, the window on the right shows the trace.

[3]Captain obvious is back



```
boucle:     beq     7,0,end
            lw      2,1,-2
            add     3,3,2      // reg3 = reg3 + reg2
            sw      3,1,0
            addi    1,1,1
            addi    7,7,-1
            beq     0,0,boucle
end:        halt               // fin du programme
```

Address	Content	ASM	
boucle	0xDC06	beq 7, 0, end	✓
10	0x88FE	lw 2, 1, -2	
11	0x0D82	add 3, 3, 2	
12	0xAC80	sw 3, 1, 0	
13	0x2481	addi 1, 1, 1	
14	0x3FFF	addi 7, 7, -1	
15	0xC079	beq 0, 0, boucle	
end	0xE07F	halt	
17	0x0000	nop	

```
EXECUTION :
next PC :           9
Instructions :      9
Pipeline Statistics :
CPI :               1,44
RAW Stall :         0
Branch Stall :      0
Speedup :           3,46
Speedup (clock) :   0,99
```

```
PC : 0     0x2801   addi 2, 0, 1     r2 = 0x0001
PC : 1     0xA880   sw 2, 1, 0       m[0] = 0x0001
PC : 2     0x2481   addi 1, 1, 1     r1 = 0x0001
PC : 3     0xA880   sw 2, 1, 0       m[1] = 0x0001
PC : 4     0x2481   addi 1, 1, 1     r1 = 0x0002
PC : 5     0x0D02   add 3, 2, 2      r3 = 0x0002
PC : 6     0xAC80   sw 3, 1, 0       m[2] = 0x0002
PC : 7     0x2481   addi 1, 1, 1     r1 = 0x0003
PC : 8     0x3C07   addi 7, 0, 7     r7 = 0x0007
```

Address	Data
R0	0x0000
R1	0x0003
R2	0x0001
R3	0x0002
R4	0x0000
R5	0x0000
R6	0x0000
R7	0x0007

Address	Data
0	0x0001
1	0x0001
2	0x0002
3	0x0000
4	0x0000
5	0x0000
6	0x0000
7	0x0000
8	0x0000

Figure 2: Simulator window

3 Simulator use

All parameters defining an architecture are available in the menu "Architecture". Main parameters are:

- Instruction set ("Architecture > Instruction set"):
  - RiSC16 original: 8 instructions.
  - Special IS[1]: 8 instructions + 8 new instruction. See Table 1 on page 2.
  - Special IS[2]: same as IS[1] but `BG` replaced by `MUL`.
  - Other: custom instruction set. Instructions can be selected in a list.
- 8, 16, 32 or 64 registers ("Architecture > Registers")
- Size of immediate constant fields in instructions can be modified ("Architecture > Imm & Instru size")

These parameters will obviously change the instruction length. It is also possible to specify if instructions use signed or unsigned operands. This choice has an impact on `BL` et `BG` and on the way *overflow* and *carry* are processed for instructions `ADD`, `SUB`, `SHL` and `SHA`.

Several predefined architecture are available in the "Architecture > preset" menu:

- Original RiSC16
- Special IS[1] – 8 reg – 16 17-bit instructions
- Special IS[1] – 16 reg – 16 24-bit instructions, see Table 2 on page 2
- Special IS[2] – 8 reg – 16 17-bit instructions, see Table 1 on page 2, `BG` replaced by `MUL`

Once the chosen architecture is selected, configured and the assembly code written, the "RUN" button compiles the program for the selected architecture. New windows will appear to run the actual simulation.

4 Manipulation

Question 1. Why the pipeline acceleration factor is not equal to the pipeline's depth?

Question 2. Run the example code to instruction #8 and observe the "speedup" and "speedup (clock)". Observe, explain and conclude.

Question 3. Considering a program without any hazard, what is the instruction number to execute to make the pipeline more efficient? $T_{\text{pipe}} = 7$, $T_{\text{seq}} = 10$

Question 4. Compare performances of several algorithm using several instruction sets:

- Write a program which compare ($<$ or $>$) two signed numbers stored is `reg1` and `reg2`. 1 will be written in `reg3` is the comparison is true, else 0.
  1. Write the program using the 8 original instructions
  2. Write the program using the Special IS[1] instruction set.
- Write a program to multiply content of `reg1` and `reg2` and write the result in `reg3` and `reg4`, LSB in `reg3`.
  1. Write the program using the original instruction set
  2. Write the program using the Special IS[1] instruction set
  3. Write the program using the Special IS[2] and using the `MUL` instruction.

  Conclude.