Anthony Debruyn
Quentin Delhaye
Alexis Lefebvre
Aurélien Plisnier

Labo 3 : Asté

RiSC16

le gaulois

Dragomir Milojevic & Yannick Allard ELEC-H-473

1 Exercise 4.1.1

```
// R1 (a) >? R2 (b)
// Result of the test in R3
// Masks in R4 & R5

movi 1, 0xFFFE
movi 2, 0xFFFF // The two signed numbers to compare

movi 4, 0xFFFF // Mask filled with 1s only
movi 5, 0x8000 // Mask 100...0

nand 2, 2, 4 // Flips all the bits of R2
addi 2, 2, 1 // Adds 1 to R2, inversion of R2 over

add 2, 1, 2 // b = a + (-b)

nand 2, 2, 5 // Keeps the MSB
beq 2, 4, positive
addi 3, 0, 0 // R1 is lower than R2 (negative test), puts 0 in R3
beq 0, 0, end
positive: addi 3, 0, 1
beq 0, 0, end
end: halt
```

2 Exercise 4.1.2

```
// R1 >? R2
// Result of the test in R3
// To make the program work correctly, check
//      Architecture > Preset > Special IS[1] - 8 reg - Instruction 17 bits
//      Architecture > Signed or UnSigned > signed

movi 1, 0xFFFE
movi 2, 0xFFFF // The two signed numbers to compare
bg 1, 2, true
addi 3, 0, 0
beq 0, 0, end
true: addi 3, 0, 1
beq 0, 0, end
end: halt
```

3 Exercise 4.2.2

```
// M0 A
// M1 B LSB
// M2 B MSB
// M3 mask
// M4 result LSB
// M5 result MSB
// M6 counter

movi 1, 0x7FFF
movi 2, 0x7FFF // The two unsigned numbers to multiply
addi 3, 0, 1
sw 1, 0, 0
sw 2, 0, 1
sw 3, 0, 3
```

```
15
16 loop_prim: lw 2, 0, 3 // Load mask
17 lw 1, 0, 0 // Load A into R1
18 nand 3, 1, 2 // NAND between mask & A
19 movi 4, 0xFFFF // NAND == 11...1 if A.curBit == 0, 011...1 otherwise
20 beq 3, 4, not_one
21
22 //If curBit == 1, do R = R + B (if curBit == 0, we do nothing)
23 lw 1, 0, 1 // B LSB in R1
24 lw 2, 0, 4 // R LSB in R2
25 addi 3, 0, 49 // Prepare the call to addition_report
26 jalr 7, 3 // Jump with save of return address to R7
27 sw 3, 0, 4 // New result LSB from R3 stored in M4
28 lw 5, 0, 5 // Result MSB loaded in R5
29 add 5, 4, 5 // Add R4 (carry bit) & R5 in R5
30 lw 6, 0, 2 // Load B MSB to R6
31 add 5, 5, 6 // Add B MSB & result MSB
32 sw 5, 0, 5 // Save new R MSB in M5
33
34 // end
35
36 // Do B *= 2 with shifti
37 not_one: lw 1, 0, 1 // B LSB in R1
38 lw 2, 0, 2 // B MSB in R2
39 shifti 2, 2, 1 // Shift B MSB by 1 bit
40 movi 3, 0x8000 // Mask 100...0
41 movi 4, 0xFFFF // Mask filled with 1s only
42 nand 3, 1, 3
43 beq 3, 4, zero // NAND == 11...1 => B LSB leftmost bit == 0
44 addi 2, 2, 1
45 zero: shifti 1, 1, 1
46 sw 1, 0, 1 // Save new B LSB to M1
47 sw 2, 0, 2 // Save new B MSB to M2
48
49 // End of B *= 2
50 // Load counter and increment
51 lw 1, 0, 6
52 addi 1, 1, 1
53
54 // Check if counter == 16
55 addi 2, 0, 16 // Load 16 into R2
56 beq 1, 2, write_res // If last bit, go to write result
57 // Else
58 sw 1, 0, 6 // Save counter
59 lw 1, 0, 3 // Load mask
60 shifti 1, 1, 1 // Mask x2
61 sw 1, 0, 3 // Save mask
62
63 beq 0, 0, loop_prim // Unconditional jump to beginning of loop
64
65 write_res: lw 3, 0, 4 // Load result LSB into R3
66 lw 4, 0, 5 // Load result MSB into R4
67 beq 0, 0, end
68
69 // Put the LSB of R1 + R2 in R3 with carry bit in R4
70 addition_report: movi 6, 0x8000 // Mask 100...0 in R6
71
72 add 3, 1, 2 // Addition of R1 & R2 in R3 (don't care about the carry for the
    moment)
```

```
73
74 nand 4, 1, 6 // NOT (R1 & R6) in R4
75 nand 5, 2, 6 // NOT (R2 & R6) in R5
76
77 movi 1, 0xFFFF // Mask 11...1 in R1
78
79 beq 4, 1, op1_leftmost_zero
80 beq 5, 1, op1_leftmost_one_op2_leftmost_zero
81 // Leftmost bits of op1 & op2 are 1 => always carry
82 addi 4, 0, 1
83 beq 0, 0, return
84 op1_leftmost_one_op2_leftmost_zero: nand 5, 3, 6 // NOT (R3 & R6) in R5
85 beq 5, 1, need_carry_bit // If the leftmost bit of the sum of op1 & op2 is 0, a
       carry bit is needed (the sum cannot be smaller than one of the operands)
86 addi 4, 0, 0
87 beq 0, 0, return
88 op1_leftmost_zero: beq 5, 1, op1_and_op2_leftmost_zero
89 // Leftmost bit of op1 is 0 & leftmost bit of op2 is 1
90  nand 5, 3, 6 // NOT (R3 & R6) in R5
91 beq 5, 1, need_carry_bit // If the leftmost bit of the sum of op1 & op2 is 0, a
       carry bit is needed (the sum cannot be smaller than one of the operands)
92 addi 4, 0, 0
93 beq 0, 0, return
94 op1_and_op2_leftmost_zero: addi 4, 0, 0
95 beq 0, 0, return
96 need_carry_bit: addi 4, 0, 1
97 beq 0, 0, return
98 return: JALR 0,7
99 end: halt
```

4 Exercise 4.2.3

```
1 // Operands in R1 & R2
2 // Result LSB in R3, MSB in R4
3
4 movi 1, 0x7FFF
5 movi 2, 0x7FFF // The two unsigned numbers to multiply
6 mul 5, 1, 2 // Here, MSB is in R4 and LSB in R5
7 add 3, 5, 0 // Move the LSB in R3
8 halt
```

5 Conclusion

One could say that using the 8 new instructions increases the performances for the algorithms we've seen.
As an exemple, we need 16 instructions for the first algorithm when using the 8 basic instructions (4.1.1), and only 8 ones when using the "bl" instruction. The results are the same for the second algorithm : on testing with the same operation, we need 953 instructions with the basic instructions (4.2.1), 812 with the use of the "shifti" instruction (4.2.2) and only 7 instruction with the "mul" instruction (4.2.3).