COMPUTER ARCHITECTURE
2020 - 2021

TD n°6 - Correction

Exercise 1:

Let the following assembly code be used:

```
ADD   Im R1 -10
ADD   Im R1 100
JMPZ ET1
JMPN ET2
PUSH Rg1 R1
STOP
ET1:   STORE D R1 100
STOP
ET2:  STORE D R1 50
STOP
```

Where Im indicates that the addressing mode used is.

```
ADD   Im R1 -10          R1 ← R1 - 10
ADD   Im R1 100          R1 ← R1 + 100
JMPZ ET1                 If R1 = 0 then goto ET1
JMPN ET2                 If R1 < 0 then goto ET1
PUSH Rg1 R1              R1 is written in the stack
STOP
ET1:   STORE D R1 100    R1 is written to address 100
STOP
ET2:  STORE D R1 50      R1 is written to address 100
STOP
```

Signed numbers are represented in complement to 2 on 8 bits.

1. At the beginning of the program, R1 contains **20**. After execution of the latter, where is R1 written? Why?
   After the two additions, R1 = 110. The jumps are not executed and R1 is written to the stack.
2. At the beginning of the program, R1 contains **-90**. After executing the program, where do you write R1? Why?
   After the two additions, R1 = 0. The first jump is made and R1 is written to address 100.

3. At the beginning of the program, R1 contains -120. After executing the program, where do you write R1? Why?
   After the two additions, R1 = -30. The second jump is made and R1 is written to address 50.

Exercise 2:

You write a program that can be either compiled or interpreted.

The compiler can compile ten thousand lines of code per second, then it takes two seconds to edit the links. Your final executable program contains twenty times more processor instruction lines than the advanced language start program and runs at a speed of one hundred thousand instructions per second.

To execute it, you also have the choice of using an interpreter working at a speed of one hundred thousand instructions per second. Unfortunately, the interpretation is complicated and requires two hundred times more instructions than your initial program.

1. Once the program is written (a thousand lines of code), how long does it take to run if it is compiled (including compilation time)? What if it is interpreted?

   The compiler takes 1/10 of a second to compile (one thousand lines of code at ten thousand lines per second), 2 seconds for link editing and generates an executable of twenty thousand instructions. This one is executed in 0.2 seconds, making a total of 2.3 seconds.

   If the program is interpreted, two hundred thousand instructions are required, which takes 2 seconds to execute.

2. What are the two execution times if it contains one million lines of code?

   With a million lines of code, the compiler requires 100 seconds, plus 2 seconds for link editing. The final program has twenty million instructions (i.e. 200 seconds of execution) and the total execution time is therefore 302 seconds.

The interpretation of the program costs two hundred million instructions and is executed in 2,000 seconds.

3. When to choose compilation over interpretation, or vice versa?

The execution of a compiled program is much faster than its interpretation, but the compilation time must be taken into account, especially if the source code must undergo many modifications.

Here, for a program containing N lines of code, the total compilation time is:

$$\frac{N}{10^4} + 2 + \frac{20N}{10^5}$$

while the interpretation time is:

$$\frac{200N}{10^5}$$

Equality is obtained when

$$N = \frac{2.10^5}{170} \approx 1176.$$

If the program is smaller, it is faster to interpret it. If not, it is better to compile it.

Exercise 3:

A processor has registers *r0* to *r9*. We want to calculate the expression:

$$((r1 \times r2 - r3) / (r1 + r4 + r5)) + r6 + r2$$

and put the result in *r0*.

1. The processor has three-operand instructions that are:
   ADD rX,rY,rZ (rX ← rY+rZ)
   SUB rX,rY,rZ (rX ← rY-rZ)
   MUL rX,rY,rZ (rX ← rY×rZ)
   DIV rX,rY,rZ (rX ← rY/rZ)
   MOVE rX,rY (rX ← rY)
   Write the sequence of instructions corresponding to the desired calculation.

The expression is constructed progressively by storing the intermediate calculations in the register *r0*.

```
MUL r0,r1,r2 ;      r0 ← r1×r2
SUB r0,r0,r3 ;      r0 ← (r1×r2)-r3
ADD r1,r1,r4 ;      r1 ← r1+r4
ADD r1,r1,r5 ;      r1 ← (r1+r4)+r5
DIV r0,r0,r1 ;      r0 ← (r1×r2-r3)/(r1+r4+r5)
ADD r0,r0,r6 ;      r0 ← ((r1×r2-r3)/(r1+r4+r5))+r6
ADD r0,r0,r2 ;      r0 ← ((r1×r2-r3)/(r1+r4+r5)+r6)+r2
```

The advantage of a three-data instruction appears from the beginning, when it is possible to multiply *r1* and *r2* without modifying these registers in order to be able to reuse their values.

2. The processor has two-operand instructions that are:

```
ADD rX,rY       (rX ← rX+rY)
SUB rX,rY       (rX ← rX-rY)
MUL rX,rY       (rX ← rX×rY)
DIV rX,rY       (rX ← rX/rY)
MOVE rX,rY      (rX ← rY)
```

Write the sequence of instructions corresponding to the desired calculation.

Here, it is no longer possible to directly multiply r1 and r2 from the beginning because this would change the value of one of the two registers, a value that is essential for the rest of the calculation.

Therefore, one of the values must first be transferred to r0 to initiate the calculation.

```
MOVE r0,r1 ;      r0 ← r1
MUL r0,r2 ;       r0 ← (r1)×r2
SUB r0,r3 ;       r0 ← (r1×r2)-r3
ADD r1,r4 ;       r1 ← r1+r4
ADD r1,r5 ;       r1 ← (r1+r4)+r5
DIV r0,r1 ;       r0 ← (r1×r2-r3)/(r1+r4+r5)
ADD r0,r6 ;       r0 ← ((r1×r2-r3)/(r1+r4+r5))+r6
ADD r0,r2 ;       r0 ← ((r1×r2-r3)/(r1+r4+r5)+r6)+r2
```

Exercise 4:

Here is a series of instructions:

```
        MVI   r1,#0 ; counter equal to 0
ici:    LDB   r2,(r0) ; get next character
        JZ    r2,fin ; end of the chain?
        ADD   r1,r1,#1 ; otherwise increment the counter
        ADD   r0,r0,#1 ; and go to the next character
        JMP   ici ; back in the loop
fin:
```

Based on the following instruction table, can you indicate what this program does?

Finding the length of a string:

- initialize the counter (i.e. r1) to 0 ;
- retrieve a character and test if it is null;
- if it is, it's finished, otherwise you have to increment the counter and the pointer on the string, and start again.

Mnemonic	Operation	Meaning
MVI *rX*,#i	*rX* ← valeur i	The rX register receives on 32 bits the immediate value i indicated (MVI, MoVe Immediate).
LDB *rX*,(*rY*)	*rX* ← Mem[*rY*]$_8$	The register rX receives the memory byte whose address is in rY (LoaD Byte).

ADD *rX*,*rY*,*rZ*
 	
ADD *rX*,*rY*,#i
	rX ← *rY* + *rZ*
	
	rX ← *rY* + valeur I
		The rX register receives the 32-bit sum of the values contained in the rY and rZ registers (or the sum of rY and an immediate value i).

JMP Adr	PC ← Adr	Unconditional jump to the address Adr (JMP, JuMP).
JZ *rX*,Adr	PC ← Adr si *rX* = 0	Conditional jump to address Adr if register rX is zero (JZ, Jump if Zero).