De l'assembleur vers le code machine : InstructionToMachineCode.py

L'intérêt de ce premier programme est de convertir un programme en assembleur en une suite d'instructions interprétables par une machine virtuelle. Les instructions concernées sont les suivantes :

Notations : r nom de registre (`r 0`, `r 1`, …, `r 31`)
o nom de registre ou constante entière (`12`, `-34`, …)
a constante entière

Syntaxe	Instruction	Effet
`add`(r_1, o, r_2)	Addition entière	r_2 reçoit $r_1 + o$
`sub`(r_1, o, r_2)	Soustraction entière	r_2 reçoit $r_1 - o$
`mult`(r_1, o, r_2)	Multiplication entière	r_2 reçoit $r_1 * o$
`div`(r_1, o, r_2)	Quotient entier	r_2 reçoit r_1 / o
`and`(r_1, o, r_2)	« Et » bit à bit	r_2 reçoit r_1 « et » o
`or`(r_1, o, r_2)	« Ou » bit à bit	r_2 reçoit r_1 « ou » o
`xor`(r_1, o, r_2)	« Ou exclusif » bit à bit	r_2 reçoit r_1 « ou exclusif » o
`shl`(r_1, o, r_2)	Décalage arithmétique logique à gauche	r_2 reçoit r_1 décalé à gauche de o bits
`shr`(r_1, o, r_2)	Décalage arithmétique logique à droite	r_2 reçoit r_1 décalé à droite de o bits
`slt`(r_1, o, r_2)	Test « inférieur »	r_2 reçoit 1 si $r_1 < o$, 0 sinon
`sle`(r_1, o, r_2)	Test « inférieur ou égal »	r_2 reçoit 1 si $r_1 \leq o$, 0 sinon
`seq`(r_1, o, r_2)	Test « égal »	r_2 reçoit 1 si $r_1 = o$, 0 sinon
`load`(r_1, o, r_2)	Lecture mémoire	r_2 reçoit le contenu de l'adresse $r_1 + o$
`store`(r_1, o, r_2)	Écriture mémoire	le contenu de r_2 est écrit à l'adresse $r_1 + o$
`jmp`(o, r)	Branchement	saute à l'adresse o et stocke l'adresse de l'instruction suivant le `jmp` dans r
`braz`(r, a)	Branchement si zéro	saute à l'adresse a si $r = 0$
`branz`(r, a)	Branchement si pas zéro	saute à l'adresse a si $r \neq 0$
`scall`(n)	Appel système	n est le numéro de l'appel
`stop`	Arrêt de la machine	fin du programme

Figure 1: Instructions traduisibles par ce programme

Le programme InstructionToMachineCode.py peut être lancé sur Linux, Windows et MacOS. Pour fonctionner, le programme prend en entré un jeu d'instruction assembleur (.asm) et fourni en sortie le jeu d'instruction en code machine (.bin).

```
add r0, 5, r1
add r0, 2, r2
add r1, r2, r3
stop
```

Figure 2 : Programme assembleur à traduire

```
0x00000000 0x080000a1
0x00000001 0x08000042
0x00000002 0x08600043
0x00000003 0x00000000
```

Figure 4 : Programme rendu par le convertisseur

Le programme doit être lancé de la manière suivante :

```
python InstructionToMachineCode.py -i asm_file.asm -o output_file.bin
```

Avec :

- -i asm_file.asm, le fichier assembleur contenant un programme à traduire
- -o output_file.bin, le fichier binaire où sont les instructions traduites en langage machine

Si vous ne parvenez pas à lancer le programme, plus d'informations sont disponible avec la commande suivante :

```
python InstructionToMachineCode.py -h
```

```
usage: InstructionsToMachineCode.py [-h] -i INPUT_FILE -o OUTPUT_FILE

This program transforms your assembly code into a machine language that can be simulated by the corresponding instruction set simulator.

optional arguments:
  -h, --help            show this help message and exit
  -i INPUT_FILE, --input_file INPUT_FILE
                        .asm input file
  -o OUTPUT_FILE, --output_file OUTPUT_FILE
                        .bin output file

Program realized in the microprocessor course by Mallory LEHUAULT-PARC
```

Simulation du programme assembleur traduit : VirtualMachineProgram.py

La simulation du programme assembleur précédemment traduit se fait au travers du programme python VirtualMachineProgram.py.

Le programme VirtualMachineProgram.py peut être lancé sur Linux, Windows et MacOS. Pour fonctionner, le programme prend en entré au minimum un jeu d'instruction de code machine (.bin), le nombre de registres et le nombre de cases mémoire nécessaires pour faire fonctionner le programme à simuler. Le programme rend en sortie de simulation un bilan de la simulation dans un fichier (.txt) horodaté.

Le programme doit être lancé de la manière suivante :

```
python VirtualMachineProgram.py -i machinecode.bin -r 32 -m 64
```

Avec :

- -i machinecode.bin, le programme en langage machine
- -r 32, le nombre de registre utilisés (ici, 32) sur 32 bits signés initialisés à 0
- -m 64, le nombre de slots mémoire (ici, 64) sur 32 bits signés initialisés à 0

Si vous ne parvenez pas à lancer le programme, plus d'informations sont disponible avec la commande suivante :

```
python VirtualMachineProgram.py -h
```

```
usage: VirtualMachineProgram.py [-h] -i INPUT_FILE [-o OUTPUT_ROOT] -r NB_REGISTERS -m MEMORY_SIZE [-ri REGISTERS_INIT_FILE] [-s STEP_BY_STEP]

This program simulates the instruction set of the input machine code, corresponding to the initial assembly program.

optional arguments:
  -h, --help            show this help message and exit
  -i INPUT_FILE, --input_file INPUT_FILE
                        .bin input file
  -o OUTPUT_ROOT, --output_root OUTPUT_ROOT
                        output directory (make sure it already exist)
  -r NB_REGISTERS, --nb_registers NB_REGISTERS
                        number of registers available
  -m MEMORY_SIZE, --memory_size MEMORY_SIZE
                        number of memory slots available
  -ri REGISTERS_INIT_FILE, --registers_init_file REGISTERS_INIT_FILE
                        .txt file to init registers value
  -s STEP_BY_STEP, --step_by_step STEP_BY_STEP
                        (True) press RIGHT KEY to simulate the next instruction

Program realized in the microprocessor course by Mallory LEHUAULT-PARC
```

VirtualMachineProgram.py propose plusieurs options de lancement en plus des options obligatoires. On y retrouve :

- -i OUTPUT_ROOT : renseigner un répertoire où les fichiers bilan (.txt) doivent être enregistrés (utile lors d'une série de test).
- -ri REGISTERS_INIT_FILE : renseigner un fichier (.txt) pour initialiser les registres utilisés lors de la simulation.
- -s STEP_BY_STEP : renseigner cette option avec la valeur « True » signifie que vous pouvez simuler instructions par instructions votre programme en appuyant sur la flèche droite directionnelle.

Exemple de lancement :

```
python VirtualMachineProgram.py -i machinecode.bin -ri regs.txt -o logs -r 32 -m 64 -s True
```

Avec :

- -i machinecode.bin, le fichier binaire correspondant au programme à simuler
- -ri regs.txt, le fichier texte pour initialiser les registres avec une certaine valeur au démarrage
- -o logs, le répertoire où sera enregistré le fichier bilan de la simulation
- -r 32, le nombre de registre utilisés sur 32 bits signés
- -m 64, le nombre de slots mémoire sur 32 bits signés
- -s True, il faudra appuyer sur la flèche de droite pour passer à l'instruction suivante

Exemple de mise en place de l'environnement de travail :

```
    Répertoire : D:\ENSTA BRETAGNE\SA5\MICROPROCESSEURS\simulateur_jeu_instructions


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        30/03/2022     17:14                logs
d-----        30/03/2022     17:19                programmes
-a----        30/03/2022     16:31          10584 InstructionsToMachineCode.py
-a----        30/03/2022     17:20            458 machineCodeOutput.bin
-a----        30/03/2022     17:15             94 regs.txt
-a----        30/03/2022     16:31          21839 VirtualMachineProgram.py
```

Figure 3 : répertoire de travail

```
    Répertoire : D:\ENSTA BRETAGNE\SA5\MICROPROCESSEURS\simulateur_jeu_instructions\logs


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        28/03/2022     10:00           1825 vm_2022_03_28-10_00_08.txt
-a----        28/03/2022     10:05           1825 vm_2022_03_28-10_05_20.txt
-a----        28/03/2022     10:06           1826 vm_2022_03_28-10_06_12.txt
-a----        28/03/2022     10:12           1826 vm_2022_03_28-10_12_22.txt
-a----        28/03/2022     10:21           1827 vm_2022_03_28-10_21_27.txt
-a----        28/03/2022     10:22           1827 vm_2022_03_28-10_22_34.txt
-a----        28/03/2022     10:34           1828 vm_2022_03_28-10_32_22.txt
-a----        28/03/2022     10:36           1825 vm_2022_03_28-10_36_03.txt
-a----        28/03/2022     10:38           1826 vm_2022_03_28-10_38_02.txt
-a----        28/03/2022     10:38           1826 vm_2022_03_28-10_38_46.txt
-a----        30/03/2022     16:31            118 vm_2022_03_30-16_31_17.txt
-a----        30/03/2022     16:35           1828 vm_2022_03_30-16_31_56.txt
```

Figure 4 : plusieurs simulations réalisées et sauvegardées dans logs/

```
    Répertoire : D:\ENSTA BRETAGNE\SA5\MICROPROCESSEURS\simulateur_jeu_instructions\programmes


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        28/03/2022     10:38            342 basicTests.asm
-a----        22/03/2022     23:40           1433 chenillard.asm
-a----        22/03/2022     23:08            313 facto.asm
-a----        22/03/2022     23:08           1106 matrix.asm
```

Figure 5 : plusieurs programmes de test dans programmes/

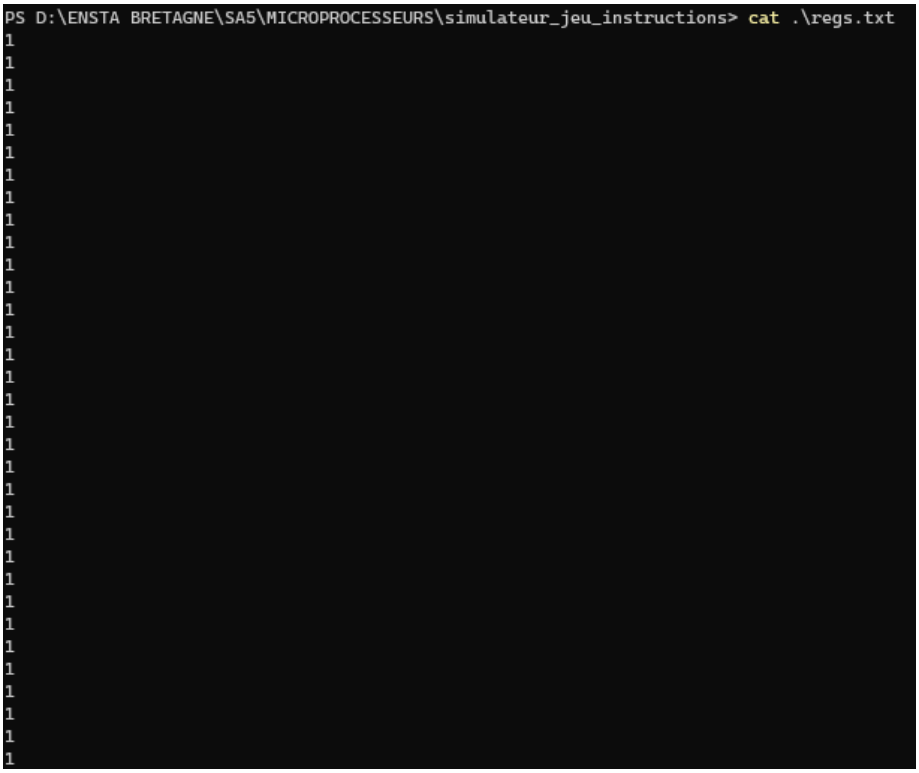

Figure 6 : fichier regs.txt avec tous les registres initialisés à 1

Classes UML correspondantes

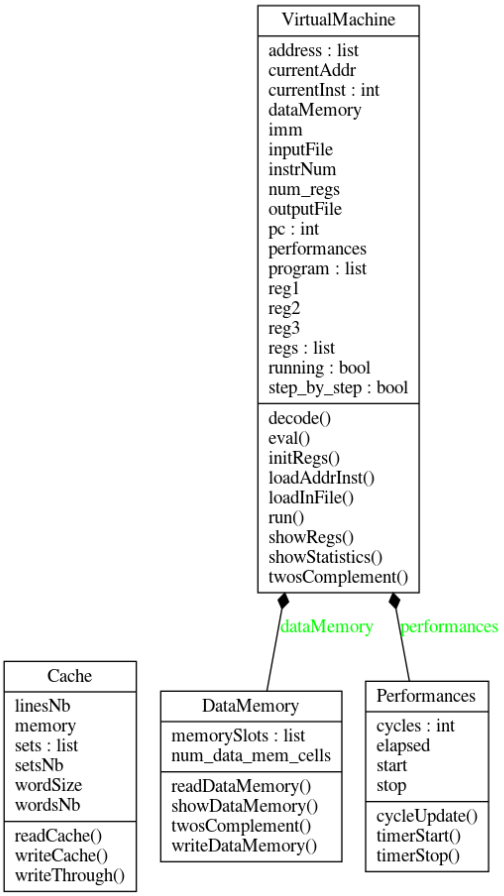

Figure 7 : VirtualMachine.py

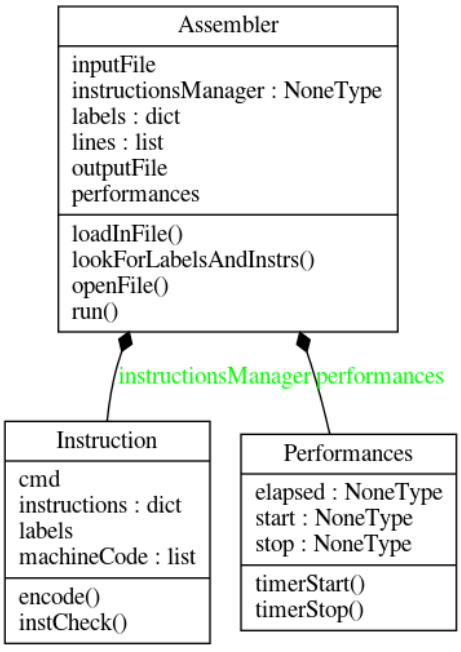

Figure 8 : InstructionToMachineCode.py