Ensta Bretagne
2, rue François Verny
29806 Brest cedex
France
Tel +33 (0)2 98 34 88 00
www.ensta-bretagne.fr

Architecture des ordinateurs

SNS 2023

31 octobre 2021

Simulateur de jeu d'instructions SIMJI

C. Goetghebeur
corentin.goetghebeur@ensta-bretagne.org

Résumé

Ce projet a pour but la conception d'un simulateur de jeu d'instruction (ISS). Il est réalisé dans le cadre du cours d'architecture des ordinateurs dispensé à l'ENSTA Bretagne en deuxième année, dans la voie d'approfondissement Systèmes Numériques et Sécurité (SNS).

Table des matières

Table des figures

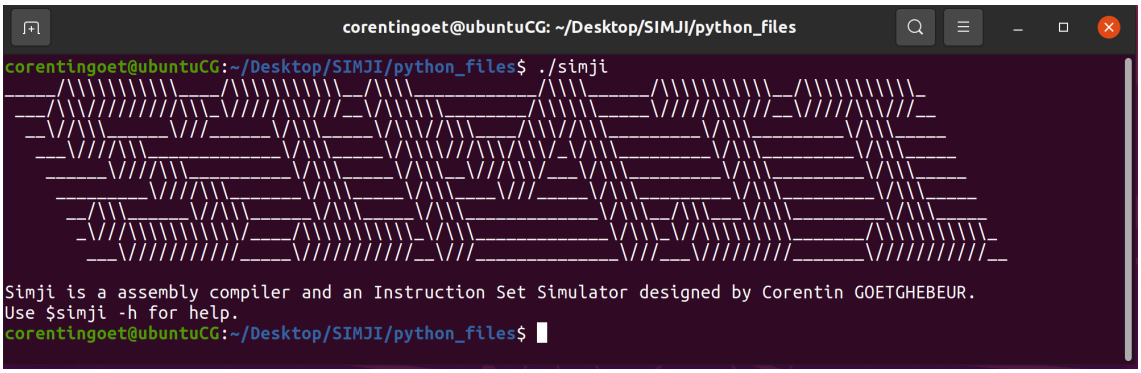

FIGURE 1 – Lancement du programme SIMJI

1 Introduction

Ce projet se sépare en deux parties principales, la traduction de programme assembleur en code machine interprétable par l'ISS, et le simulateur. J'ai choisi de réaliser ce projet en utilisant le langage Python.

2 Téléchargement et installation

Les instructions de ce paragraphe sont également disponibles dans le fichier README.md fournit dans l'archive du projet. Pour obtenir le programme, vous pouvez le télécharger sur GitHub ou cloner le repository en utilisant git:

```
git clone https://github.com/CorentinGoet/SIMJI
```

Pour installer SIMJI, placez-vous dans le dossier du projet, donnez le droit d'exécution au programme d'installation et exécutez ce programme.

```
chmod +x setup.sh
./setup.sh
```

Vous pouvez ensuite vous rendre dans le dossier *python_files* et exécuter le programme simji (voir figure 1) :

```
./simji
```

3 Utilisation

Pour exécuter le programme, déplacez-vous dans le dossier *python_files* et exécutez simji. Pour afficher l'aide de syntaxe, utilisez :

```
./simji -h
```

Pour utiliser l'assembleur, utilisez l'option *assemble* et pour l'iss utilisez l'option *execute*.

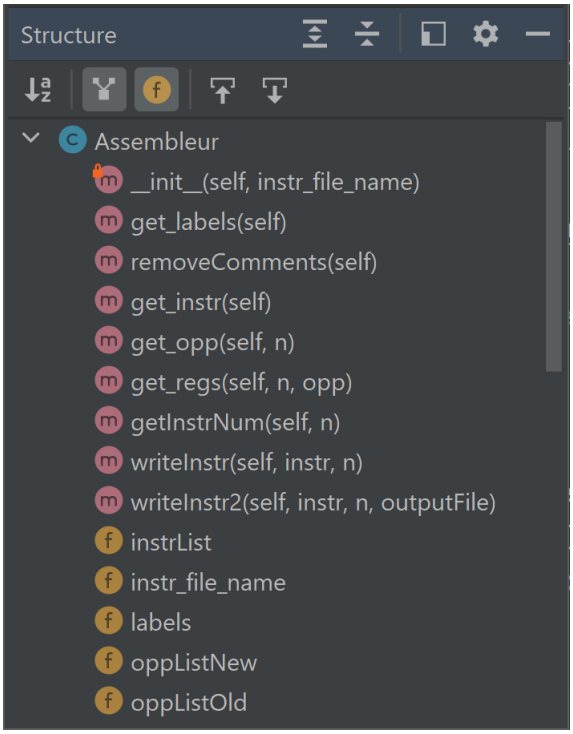

FIGURE 2 – Diagramme de classes de l'assembleur

4 Assembleur

La première partie du projet consiste à la conception d'un programme de traduction de code assembleur vers un code machine exécutable par l'ISS. Cette partie est gérée par le fichier python *assembleur.py* et plus précisément de la classe Assembleur.

4.1 Syntaxe

Les programmes en langage assembleur doivent respecter la syntaxe prévue par le projet :

- Les opérations se limitent à celles présentée par la liste de la figure 2.
- Les commentaires sont signalés par '#' ou ';' tous les caractères se situant à la suite de ceux-ci sur une ligne seront ignorés.
- Les *labels* doivent commencer par un 'L' majuscule et se terminer par ':'.
- Pas d'espace entre les paramètres sur une ligne (Par exemple: *add r0,4,r1* et non *add r0, 4, r1*).
- Utiliser les opérations de saut ou branchement (*jmp, braz, branz*) uniquement avec des labels.

4.2 Fonctionnement

Pour traduire le code assembleur, le programme procède par plusieurs étapes :

- Suppression des commentaires (méthode *removeComments*).

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 3 – Liste des opérations disponibles.

— Récupération des *labels* (méthode *get_labels*)
— Pour chaque ligne,
  — L'opération est associée au nombre correspondant (0 pour stop, 1 pour add ...) (Méthode *get_opp*).
  — Les registres ou nombres sont remplacés en fonction de l'opération (méthodes *get_regs* et *getInstrNum*).
  — L'instruction est écrite au format hexadécimal dans le fichier correspondant aux instructions.

5 Simulateur

Pour le simulateur, on utilisera 32 registres de 32 bits chacun. Cette partie est implémentée dans le fichier VM.py par la classe VM. Ce programme fonctionne de la manière suivante :

— Chargement du programme hexadécimal (méthode *progLoad*).
— Tant que le simulateur ne rencontre pas d'instruction stop,
  — Récupération de l'instruction à exécuter correspondant au *program counter* en utilisant la méthode *fetch*.
  — Décodage de l'instruction récupérée par la méthode *decode*.
  — Exécution de l'instruction par la méthode *eval*.
  — Incrémentation du *program counter*.

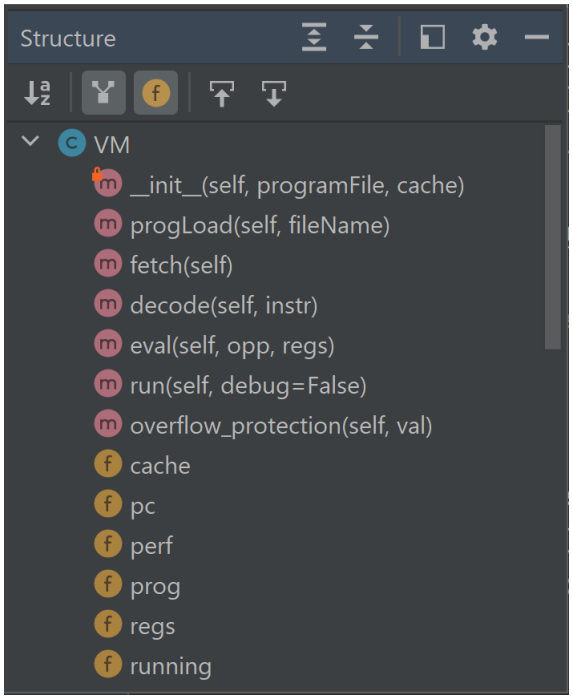

FIGURE 4 – Diagramme de classes du simulateur

6 Stockage des données

Pour modéliser plus fidèlement le fonctionnement d'un processeur classique, le simulateur utilise un cache. Ce cache est modélisé par la classe Cache (du fichier Data.py), qui s'appuie lui-même sur une mémoire de stockage modélisée par la classe Storage.

6.1 Mémoire de stockage

Pour la mémoire de stockage, les données sont stockées dans une liste de 1024 valeurs (*mem*). Pour manipuler ces données, on peut :

- Ecrire à une adresse grâce à la méthode *write*.
- Lire une case mémoire dont on connait l'adresse grâce à la méthode *read*.
- Lire en rafale grâce à la méthode *burst_read*.
- Charger un fichier hexadécimal comme mémoire grâce à la méthode *loadMem*.
- Ecrire la mémoire dans un fichier grâce à la méthode *writeMem*.

6.2 Cache

Le cache est constitué de 16 lignes contenant 8 blocs par lignes. Chaque ligne contient également une adresse, un tag, et un bit de validation.

Lorsque le simulateur demande à accéder aux données par l'opération *load*, le cache vérifie si les données demandées sont stockées dans le cache, et si elles ne le sont pas, il lit en rafale les informations proches de celle demandée et les stocke (méthode *read*.

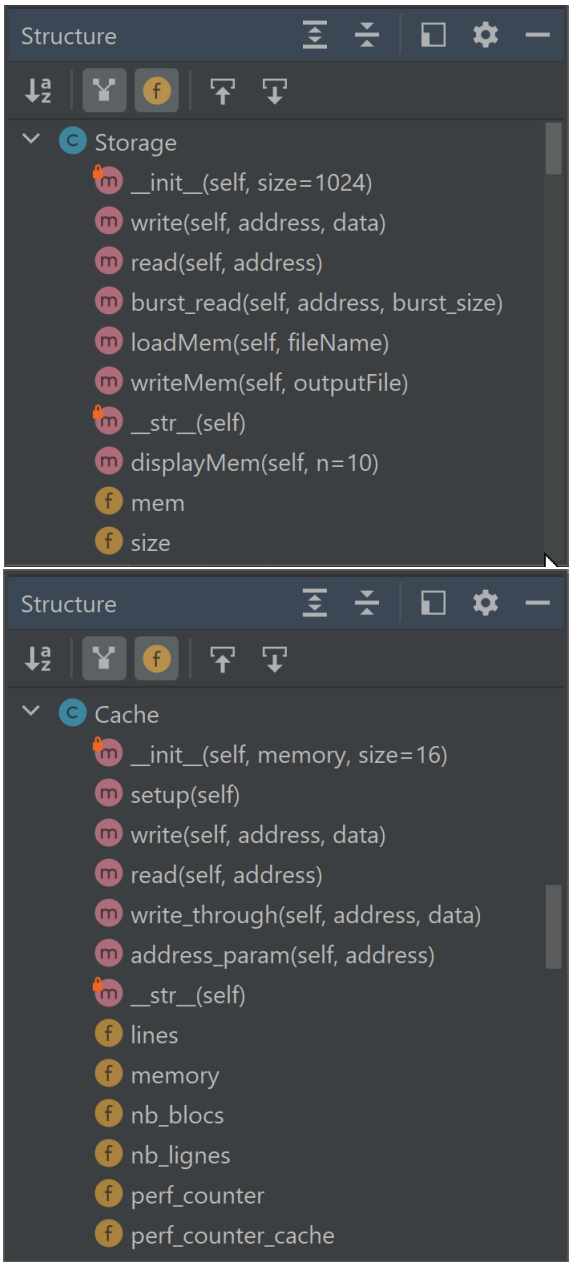

FIGURE 5 – diagrammes de classes du stockage de données

```
################################################
 ---------  CACHE  ---------
adresse tag  valid Bloc 0      Bloc 1      Bloc 2      Bloc 3      Bloc 4      Bloc 5      Bloc 6      Bloc 7
0x0     0b0  0     0xf         0x262       0x0         0x0         0x0         0x0         0x0         0x0
0x1     0b0  0     0x0         0x0         0x0         0x0         0x0         0x0         0x0         0x0
0x2     0b0  0     0x0         0x0         0x0         0x0         0x0         0x0         0x0         0x0
0x3     0b0  0     0x0         0x0         0x0         0x0         0x0         0x0         0x0         0x0
0x4     0b0  0     0x0         0x0         0x0         0x0         0x0         0x0         0x0         0x0
0x5     0b0  0     0x0         0x0         0x0         0x0         0x0         0x0         0x0         0x0
0x6     0b0  0     0x0         0x0         0x0         0x0         0x0         0x0         0x0         0x0
0x7     0b0  0     0x0         0x0         0x0         0x0         0x0         0x0         0x0         0x0
0x8     0b0  0     0x0         0x0         0x0         0x0         0x0         0x0         0x0         0x0
0x9     0b0  0     0x0         0x0         0x0         0x0         0x0         0x0         0x0         0x0
0xa     0b0  0     0x0         0x0         0x0         0x0         0x0         0x0         0x0         0x0
0xb     0b0  0     0x0         0x0         0x0         0x0         0x0         0x0         0x0         0x0
0xc     0b0  0     0x0         0x0         0x0         0x0         0x0         0x0         0x0         0x0
0xd     0b0  0     0x0         0x0         0x0         0x0         0x0         0x0         0x0         0x0
0xe     0b0  0     0x0         0x0         0x0         0x0         0x0         0x0         0x0         0x0
0xf     0b0  0     0x0         0x0         0x0         0x0         0x0         0x0         0x0         0x0
################################################
```

FIGURE 6 – Affichage du cache

Lorsque le simulateur doit stocker des données en mémoire, elles sont écrites à la fois dans le cache et dans la mémoire de stockage (méthode *write_through*).

6.3 Mesure de performance

En utilisant l'option *-p*, le programme affiche à la fois les performances de calcul mais aussi celles des accès aux données. La mesure de performances en calcul compte les opérations effectuées sans distinctions, elle est donnée en opérations par secondes. La mesure des performances d'accès à la mémoire a pour but d'évaluer l'intérêt du cache. Elle est donnée en nombre d'opérations équivalentes en termes de temps. Un accès aux données dans le cache est considéré 10 fois plus long à exécuter qu'une opération telle que l'addition, et l'accès aux données situées en mémoire de stockage est considéré 100 fois plus long.

7 Tests

7.1 Tests unitaires

Chaque opération a été testée séparément. Les programes de tests sont disponibles dans le dossier *Tests/tests_op*.

7.2 Suite de Fibonacci

Le premier programme de test est celui de la suite de Fibonacci :

$$\begin{cases} u_{n+1} = 3u_n + 1 & \text{si n est impair} \\ u_{n+1} = \frac{u_n}{2} & \text{si m est pair} \end{cases}$$

Le *control-flow graph* du calcul de cette suite peut se représenter comme sur la figure 7.

Un programme permettant de mesurer la durée de vol de cette suite en python est :

```
n = 10
r1 = 0
r2 = 1
for i in range(n):
    r1, r2 = r2, r1 + r2
print(r1)
```

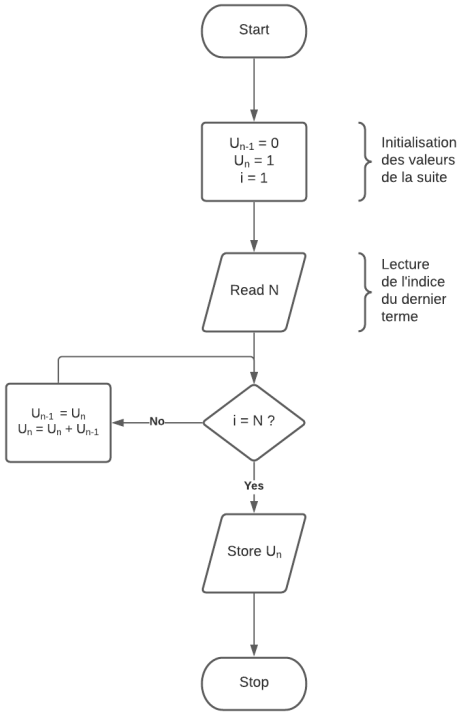

FIGURE 7 – Flowchart du calcul de la suite de Fibonacci

En langage assembleur, ce programme devient:

```
# initialisation
    add r0,0,r1;stockage du terme précédent
    add r0,1,r2;stockage du terme actuel
    load r0,0,r3;nombre cible
    add r0,0,r4; compteur de boucle

# Boucle
L1:
    slt r4,r3,r5
    braz r5,Label_end; si i>n, on stop
    add r1,r2,r6; r6 (tmp) recoit r1 + r2
```

```
    add r2,0,r1; r1 recoit r2
    add r6,0,r2; r2 recoit r6 (tmp)
    add r4,1,r4; i++
    jmp L1,r10

#fin
Label_end:
    store r0,1,r1; on stocke la valeur de r1 dans la case mémoire 1
    stop
```