

---

# **SIMJI - Simulateur de Jeu d'Instructions**

U.E. 4.2. Architectures Numériques - ENSTA Bretagne

Alexandre Froehlich

02 Mars 2021

## Table des matières

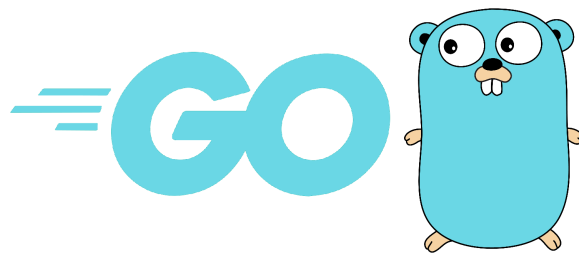
<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Motivation . . . . .	3
1.2	Objectifs . . . . .	3
<b>2</b>	<b>Architecture et langage</b>	<b>4</b>
2.1	Les registres . . . . .	4
2.2	Les instructions . . . . .	4
2.3	La mémoire et le cache . . . . .	5
<b>3</b>	<b>L'assembleur</b>	<b>7</b>
3.1	Manipulation de fichier . . . . .	7
3.2	Adresses des labels . . . . .	8
3.3	Traduction des instructions en codes . . . . .	9
3.4	Traduction des codes en hexadécimal . . . . .	11
3.5	Exemple . . . . .	12
<b>4</b>	<b>La machine virtuelle</b>	<b>13</b>
4.1	Interprétation des instructions . . . . .	14
4.2	Désassemblage . . . . .	16
4.3	Benchmark et profiling . . . . .	16
<b>5</b>	<b>Le logiciel</b>	<b>19</b>
5.1	La ligne de commande (CLI) . . . . .	19
5.2	L'interface utilisateur (GUI) . . . . .	21
<b>6</b>	<b>Conclusion</b>	<b>23</b>

## 1 Introduction

Nous sommes entourés de machines de Turing. Des automates qui agissent et réagissent à la moindre requête qu'on leur soumet. Il serait intéressant de revenir au base et de comprendre comment, à *bas niveau* un ordinateur est capable d'interpréter une série de commandes qu'un programmeur lui donne : le programme.

### 1.1 Motivation

SIMJI est un simulateur de jeu d'instruction développé dans le cadre de mes études à l'ENSTA Bretagne. L'enseignant nous laissant le choix du langage j'ai décidé d'utiliser ce projet pour sortir de ma zone de confort et me tourner vers un langage plus "*moderne*", j'ai donc choisi le Golang.



**FIG. 1 :** Le logo de Golang (prononcé "Go") est une marmotte bleue

### 1.2 Objectifs

Les objectifs principaux de ce projet sont les suivants :

- découvrir l'architecture des processeurs
- découvrir la compilation des langages

A cette fin nous avons réalisé un simulateur de jeu d'instruction se décomposant en plusieurs sous-programmes :

- **un programme d'assemblage** permettant de prendre un programme rédigé en mini-MIPS et sortant des instructions machine
- **un simulateur d'instructions** prenant ces instructions machines et exécutant le stack

## 2 Architecture et langage

### 2.1 Les registres

Le CPU (Compute Processing Unit) est une puce de silicone contenant, aujourd'hui, des milliards de transistors faisant passer ou non un signal électrique. Afin de traiter des calculs plus complexes que ceux effectués par les portes logiques que représentent ces transistors, il est intéressant de stocker les résultats obtenus.

C'est ici qu'interviennent les registres. Les registres sont des espaces mémoires d'accès extrêmement rapide en comparaison des disques durs classiques. Ils sont en nombre limités mais permettent de stocker temporairement des valeurs provenant du processeur.

### 2.2 Les instructions

Pour que le programmeur puisse donner des ordres à la machine il doit écrire une série d'instructions. Nous allons pour ce projet nous baser sur le jeu d'instructions suivant, similaire au jeu MIPS :

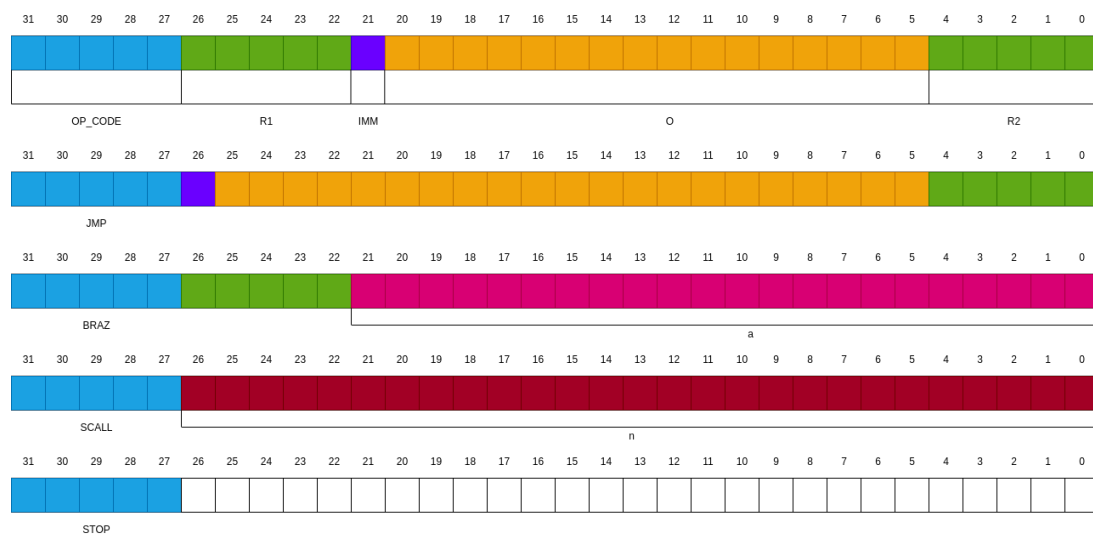
Notations:  $r$  nom de registre ( $r_0, r_1, \dots, r_{31}$ )  
 $o$  nom de registre ou constante entière (12, -34, ...)  
 $a$  constante entière

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

**FIG. 2 :** Jeu d'instructions “mini-MIPS”

Pour se simplifier le travail, on va supposer que l'architecture du processeur que nous allons simuler est de 32 bits. Ainsi chacune des instructions sera elle même encodée sur 32 bits. On va ensuite pouvoir réserver chacun de ces bits pour une fonctionnalité donnée :

- 4 bits pour l'opcode : le type d'instruction
- 5 bits pour chaque adresse de registre
- 1 bit pour indiquer si la valeur qui suit est immédiate ou bien un registre
- 16 bits pour la valeur immédiate ou bien l'adresse de registre



**FIG. 3 :** Structure des instructions du jeu mini-MIPS sur 32 bits

Cependant l'expression en binaire de ce code est bien trop lourde. On lui préférera l'écriture hexadécimale qui permet de ne pas perdre en informations tout en rendant l'écriture plus lisible.

Par exemple si nous souhaitons ajouter 5 au registre `r1` l'instruction correspondante en assembleur est :

```
1 add r1, 5, r1
```

Qui se traduit alors en instruction machine par : 0110 1000 0110 0000 0000 0000 1010 0001 ou bien en hexadécimal `0x686000a1`.

On peut tout de suite mieux comprendre l'intérêt d'une telle écriture.

## 2.3 La mémoire et le cache

J'ai parlé précédemment des registres. Cependant les registres étant en nombre limité, il faut trouver un autre moyen pour stocker l'information.

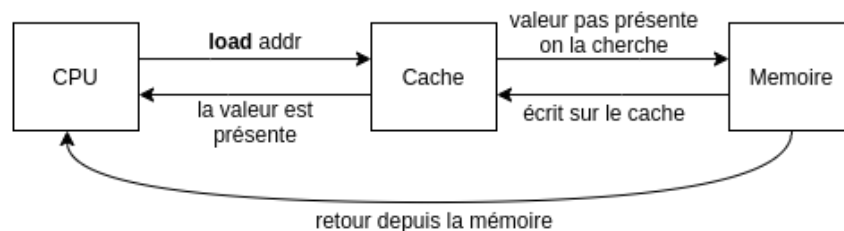
Pour cela le jeu d'instructions mini-MIPS nous fournit des instructions `load` et `store` pour manipuler ces valeurs dans la mémoire. Cependant leur accès reste bien plus lent, en témoigne le tableau suivant donnant le nombre de cycles nécessaires pour accéder à l'information en fonction de l'endroit où elle est stockée :

Type	Taille des données	Nombre de cycles
Registre	4 octets = 32 bits	0
Cache	64 octets	1
Mémoire virtuelle	pages de 4kB	100
Disque dur	secteurs de disque	100 000
Stockage réseau	morceaux de fichiers	10 000 000

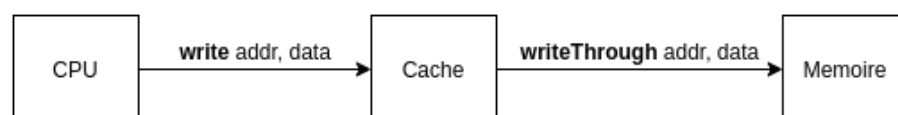
Pour cela on implémente un *cache* qui va enregistrer une donnée dans un espace mémoire plus rapide si elle est utilisée plus fréquemment afin d'améliorer les performances du système.

Dans ce projet on ne va s'intéresser qu'à l'implémentation d'un cache à correspondance directe. Dans ce cas chaque `set` du cache ne contient qu'une ligne. On peut alors résumer le fonctionnement du cache par le schéma suivant :

### Lecture



### Ecriture



**FIG. 4 :** Cycle de vie du cache avec des “hits” et des “misses”. Ce cache fonctionne en mode “write through” c’est à dire en écrivant à la fois dans le cache et dans la mémoire.

### 3 L'assembleur

Un assembleur est un programme d'ordinateur qui traduit un programme écrit en langage assembleur — essentiellement, une représentation mnémonique du langage machine — en code objet.

– Wikipedia

L'assembleur est un outil qui prend en entrée des instructions rédigées en assembleur et qui les traduit en code compréhensible par la machine.

Dans ce projet nous devrons être capable d'interpréter les instructions du jeu mini-MIPS. De plus le langage doit pouvoir supporter les labels qui seront des références dans le programme pour la gestion de tout ce qui est “saut” dans le code.

Pour se faire il faut procéder de la manière suivante :

1. Récupérer le texte correspondant au programme
2. Séparer le texte en ensemble de lignes
3. Parcourir une première fois les lignes et récupérer les adresses des labels
4. Parcourir une seconde fois les lignes :
  1. Remplacer les références de labels par leur adresse
  2. Traduire les arguments par des codes
5. “Assembler” les codes des arguments en une instruction hexadécimale

Nous allons voir chacune des étapes précédentes et expliquer les quelques spécificités ou difficultés rencontrées.

#### 3.1 Manipulation de fichier

En golang, la manipulation des fichiers se réalise à l'aide d'une bibliothèque standard “outil”. Une autre spécificité du langage est la gestion des erreurs. Ces derniers se manipulent comme des variables : on tente une action et on récupère la présence d'erreur ou non dans une variable. Si cette variable est non nulle, il y a eu une erreur et on peut agir en conséquences.

```
1 // OpenFile ouvre un fichier et retourne ses lignes
2 // sous la forme d'un tableau
3 func OpenFile(filename string) string {
4     content, err := ioutil.ReadFile(filename)
5     // si il y a eu une erreur durant l'ouverture
6     if err != nil { panic() }
7
8     // on sépare les lignes obtenues dans le fichier
```

```
9     lines := strings.Split(string(content), "\n")
10    for i := range lines {
11        lines[i] = strings.TrimSpace(lines[i])
12    }
13
14    return lines
15 }
```

### 3.2 Adresses des labels

La prochaine étape est de parcourir l'ensemble des lignes du programme pour trouver les labels et stocker leur adresse dans un dictionnaire. Pour cela golang possède un type standard : la map. On peut ainsi définir le type des clés ainsi que le type des valeurs : `var dictionnaire map[string]int`.

Pour détecter un label on doit d'abord être capable de détecter qu'une ligne n'est pas un commentaire ou bien une ligne vide :

```
1 // EstVideOuCommentaire retourne vrai si la ligne est
2 // un commentaire ou bien est vide
3 func EstVideOuCommentaire(ligne string) bool {
4     // si la ligne est vide on retourne vrai
5     if ligne == "" { return true }
6
7     // on sépare la ligne en arguments
8     args := strings.Split(ligne, " ")
9
10    // on vérifie si la ligne entière est un commentaire
11    return args[0][0] == ";"
12 }
```

Ensuite une ligne définissant un label commence soit par le label, ou bien le label compose la ligne. Il suffit alors de vérifier que le premier mot finisse par : pour s'assurer de la définition d'un label :

```
1 // ExtraitLabel retourne le label si il existe dans une ligne
2 func ExtraitLabel(ligne string) string {
3     // on sépare la ligne en arguments
4     args := strings.Split(ligne, " ")
5
6     // on vérifie si le premier mot est un label
7     if args[0][len(args[0])-1] != ":" { return "" }
8
9     // on extrait le label
10    return args[0][:len(args[0])-1]
11 }
```

Il ne reste alors plus qu'à parcourir le programme pour définir un dictionnaire des adresses des la-



bels :

```

1 // AdressesLabels construit un dictionnaire contenant
2 // les adresses des labels
3 func AdressesLabels(lignes []string) map[string]int {
4     var adresses map[string]int
5     var pc int
6
7     for _, ligne := range lignes {
8         // si la ligne n'est pas viable
9         if !EstVideOuCommentaire(ligne) { continue }
10
11         label := ExtraireLabel(ligne)
12         // si il y a un label on enregistre son adresse
13         if label != "" { adresses[label] = pc }
14         pc++
15     }
16
17     return adresses
18 }

```

```

7:01PM DBG Loading Labels Dictionary
7:01PM DBG Found label addr=2 label=L_LOOP
7:01PM DBG Found label addr=7 label=L_PAIR
7:01PM DBG Found label addr=9 label=L_IMPAIR
7:01PM DBG Found label addr=12 label=L_END

```

**FIG. 5 :** SIMJI affiche cette étape intermédiaire en utilisant le flag `--debug` lors de son exécution.

### 3.3 Traduction des instructions en codes

Cette étape est la plus importante, en terme d'efforts, dans l'assembleur. Il va falloir itérer sur chacune des lignes et traduire chaque argument en code qui sera ensuite traduit en hexadécimal.

On peut commencer en concevant une fonction permettant de dire si un argument est un registre et si oui quel est son numéro de registre. Pour cela golang nous permet de retourner plusieurs valeurs qui ne sont pas du même type, un peu comme python. Cela se déclare dans la signature de la fonction :

```

1 // EstUnRegistre permet de parser un argument et de dire si c'est un
   registre
2 func EstUnRegistre(argument string, labels map[string]int) (int, bool)
   {
3     // on retire le "r" du registre si il est présent
4     if argument[0] == 'r' {
5         // on essaie de parser l'argument
6         value, err := strconv.Atoi(argument[1:])
7         // si il y a une erreur c'est un mauvais registre

```

```
8     if err != nil {
9         panic("Error while parsing register: ", argument)
10    }
11    return value, true
12 }
13
14 // si il s'agit d'un label on retourne son adresse
15 if value, ok := labels[argument]; ok {
16     return value, false
17 }
18
19 // on essaie de parser l'argument
20 value, err := strconv.Atoi(argument)
21 // si il a une erreur c'est qu'on ne sait pas quelle est cette
   valeur
22 if err != nil {
23     panic("Error while parsing immediate: ", argument)
24 }
25
26 // sinon on a réussi à parser la valeur
27 return value, false
28 }
```

On est maintenant capable de comprendre chaque argument séparément en traduisant les adresses des labels et en séparant les registres des valeurs immédiates. Il ne reste plus qu'à traduire les instructions.

Pour information voici les codes associés à chaque type d'instruction :

```
1  var OpCodes = map[string]int{
2      "stop": 0,    "add": 1,
3      "sub": 2,    "mul": 3,
4      "div": 4,    "and": 5,
5      "or": 6,     "xor": 7,
6      "shl": 8,    "shr": 9,
7      "slt": 10,   "sle": 11,
8      "seq": 12,   "load": 13,
9      "store": 14, "jmp": 15,
10     "braz": 16,  "branz": 17,
11     "scall": 18,
12 }
```

```

7:01PM DBG Translating ASM to hex instr
7:01PM DBG scall 0 instr=[18,0] pc=0
7:01PM DBG add r3 r0 r0 instr=[1,3,0,0,0] pc=1
7:01PM DBG scall 1 instr=[18,1] pc=2
7:01PM DBG sle r1 1 r2 instr=[11,1,1,1,2] pc=3
7:01PM DBG branz r2 L_END instr=[17,2,12] pc=4
7:01PM DBG and r1 1 r2 instr=[5,1,1,1,2] pc=5
7:01PM DBG branz r2 L_IMPAIR instr=[17,2,9] pc=6
7:01PM DBG div r1 2 r1 instr=[4,1,1,2,1] pc=7
7:01PM DBG jmp L_LOOP r0 instr=[15,1,2,0] pc=8
7:01PM DBG mul r1 3 r1 instr=[3,1,1,3,1] pc=9
7:01PM DBG add r1 1 r1 instr=[1,1,1,1,1] pc=10
7:01PM DBG jmp L_LOOP r0 instr=[15,1,2,0] pc=11
7:01PM DBG stop instr=[0] pc=12

```

**FIG. 6 :** SIMJI affiche cette étape intermédiaire en utilisant le flag `--debug` lors de son exécution.

### 3.4 Traduction des codes en hexadécimal

C'est la dernière étape de l'assemblage. Il faut maintenant récupérer les codes et les assembler dans une instruction unique sur 32 bits. Pour cela on va s'aider des décalages binaires.

En go lang les décalages binaires sont identique au C. On utilise pour cela les chevrons doubles : `<<` . Enfin pour reconstruire l'instruction on peut créer une fonction qui va respecter les règles que j'ai décrit au chapitre sur les instructions :

```

1 func TraductionHexa(inst []int) int {
2     decInstr := instr[0] << 27
3
4     // on fait un switch sur le nombre d'arguments
5     switch len(instr) {
6     case 1:
7         break
8     case 2:
9         // scall
10        decInstr += instr[1] // num
11        break
12    case 3:
13        // braz
14        decInstr += instr[1] << 22 // reg
15        decInstr += instr[2]      // address
16    case 4:
17        // jmp
18        decInstr += instr[1] << 26 // imm
19        decInstr += BinaryComplement(instr[2], 21) << 5 // o
20        decInstr += instr[3]      // r
21    case 5:

```

```
22      // add, load, store ...
23      decInstr += instr[1] << 22           // reg
24      decInstr += instr[2] << 21           // imm
25      decInstr += BinaryComplement(instr[3], 16) << 5 // o
26      decInstr += instr[4]                 // reg
27      break
28  }
29
30  return decInstr
31 }
```

On a utilisé ici une fonction maison permettant d'écrire un nombre entier signé en nombre binaire en complément à 2. Voici son implémentation :

```
1  // BinaryComplement permet de calculer le complément à 2 d'un entier
2  func BinaryComplement(number int, size int) int {
3      // aucun travail à faire
4      if number >= 0 { return number }
5
6      return (1 << (size - 1)) - number
7  }
```

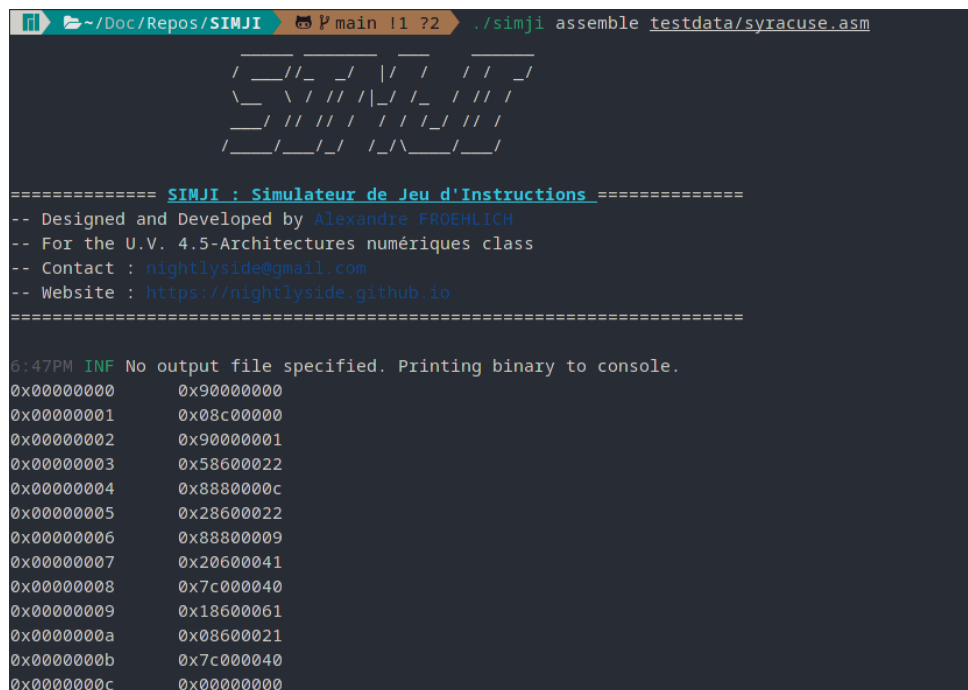
### 3.5 Exemple

Pour le programme suivant, permettant de calculer les termes de la suite de syracuse, on a le programme en assembleur :

```
1      ; initialise valeur de début
2      ; add r0 15 r1
3      scall 0
4      ; compteur pour stockage mémoire
5      add r3, r0, r0
6  L_LOOP:
7      ; on affiche r1
8      scall 1
9      ; r1 <= 1 -> fin du programme
10     sle r1, 1, r2
11     branz r2, L_END
12     ; on teste la parité r2 = r1 & 0x0001
13     and r1, 1, r2
14     branz r2, L_IMPAIR
15
16     ; si r1 est pair r1 /= 2
17  L_PAIR: div r1, 2, r1
18         jmp L_LOOP, r0
19
20     ; sinon r1 = r1*3 + 1
21  L_IMPAIR: mult r1, 3, r1
```

```
22         add r1, 1, r1
23         jmp L_LOOP, r0
24 L_END:
25         stop
```

Qui donne après assemblage par notre programme :



```
~/Doc/Repos/SIMJI main 11 ?2 ./simji assemble testdata/syracuse.asm

===== SIMJI : Simulateur de Jeu d'Instructions =====
-- Designed and Developed by Alexandre FROELICH
-- For the U.V. 4.5-Architectures numériques class
-- Contact : nightlyside@gmail.com
-- Website : https://nightlyside.github.io
=====

6:47PM INF No output file specified. Printing binary to console.
0x00000000 0x90000000
0x00000001 0x08c00000
0x00000002 0x90000001
0x00000003 0x58600022
0x00000004 0x8880000c
0x00000005 0x28600022
0x00000006 0x88800009
0x00000007 0x20600041
0x00000008 0x7c000040
0x00000009 0x18600061
0x0000000a 0x08600021
0x0000000b 0x7c000040
0x0000000c 0x00000000
```

**FIG. 7 :** Assemblage du programme syracuse en utilisant la commande “assemble” de SIMJI

## 4 La machine virtuelle

Le deuxième programme que comporte le projet SIMJI est une machine virtuelle permettant d’exécuter les instructions machines que l’assembleur nous a sorti.

Une machine virtuelle est simplement un logiciel permettant d’exécuter un autre programme dans une machine existante, d’où le nom de virtuelle.

La machine virtuelle effectue les actions suivante pour exécuter un programme :

1. Création de la machine et de sa mémoire
2. Chargement du fichier du programme dans la machine virtuelle
3. Execution ligne par ligne du programme
  1. on récupère la ligne à exécuter avec un compteur : PC

2. on exécute son contenu en modifiant les registres et PC
4. On atteint la fin du programme sur une instruction `stop`

## 4.1 Interprétation des instructions

Pour que la machine puisse comprendre les instructions on doit suivre un ensemble de règles, les mêmes que pour l'assemblage.

On va d'abord chercher à comprendre l'instruction. Pour cela on peut écrire une fonction de décodage qui utilise encore une fois le principe du décalage binaire avec les doubles chevrons `>>` :

```
1 // Decode retourne le decodage d'une instruction machine
2 func Decode(instruction int) (int, int, int, int, int, int, int, int,
   int) {
3     instrNum    := (instruction & 0xF8000000) >> 27
4     imm1        := (instruction & 0x04000000) >> 26
5     o1          := (instruction & 0x03FFFFE0) >> 5
6     r1          := (instruction & 0x07C00000) >> 22
7     imm2        := (instruction & 0x00200000) >> 21
8     o2          := (instruction & 0x001FFFFE0) >> 5
9     r2          := instruction & 0x00000001F
10    a            := instruction & 0x003FFFFFF
11    n            := instruction & 0x07FFFFFF
12
13    o1 = ReverseBinaryComplement(o1, 21)
14    o2 = ReverseBinaryComplement(o2, 16)
15
16    return instrNum, imm1, o1, r1, imm2, o2, r2, a, n
17 }
```

Sans oublier la fonction permettant de passer d'un nombre en écriture en complément à 2, en un nombre entier signé :

```
1 // ReverseBinaryComplement permet de retourner un entier signé
2 // a partir d'un nombre binaire en complément à 2
3 func ReverseBinaryComplement(number int, size int) int {
4     bit := number >> (size - 1)
5     if bit == 1 {
6         return -1 * (number - 1<<(size-1))
7     }
8     return number
9 }
```

L'interprétation du code s'effectue alors dans une grande fonction `eval()` qui comporte un switch sur le numéro d'instruction `instrNum`. Je me permet de ne pas gâcher de place en la réécrivant. Elle est disponible dans son propre fichier dans les sources du projet.

Voici un extrait de cette fonction `eval()` j'ai choisi en particulier une fonction simple, celle d'addition distinguant les cas où l'on utilise une valeur immédiate ou bien un registre :

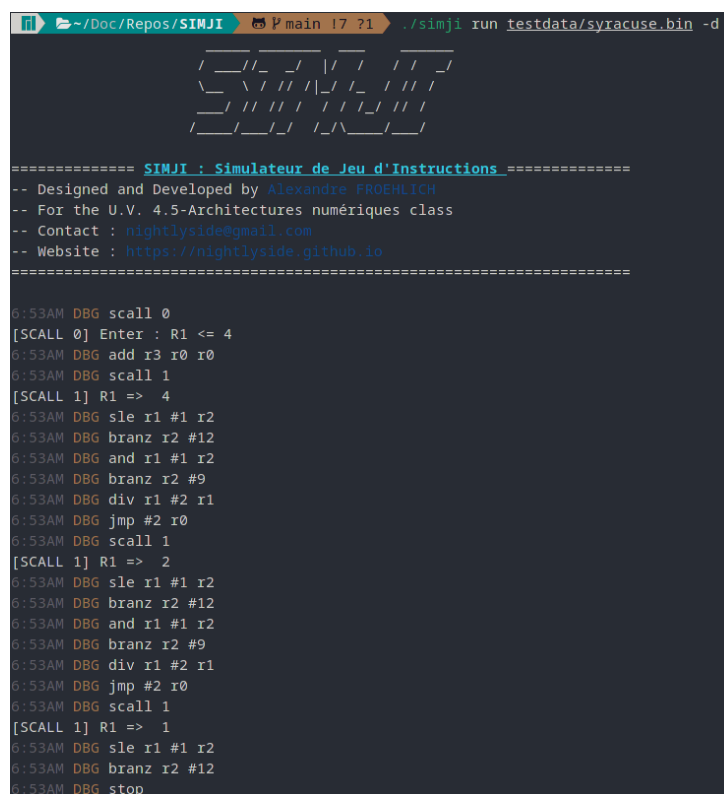
```

1  case 1:
2      // registre sinon immediate
3      if imm2 == 0 {
4          log.Debug().Msgf("add r%d r%d r%d", r1, o2, r2)
5          vm.regs[r2] = vm.regs[r1] + vm.regs[o2]
6      } else {
7          log.Debug().Msgf("add r%d #d r%d", r1, o2, r2)
8          vm.regs[r2] = vm.regs[r1] + o2
9      }
10     break

```

Il y a une exception à faire, il s'agit des appels systèmes `scall` qui sont implémentés de façon arbitraire. Ainsi un 0 veut dire “afficher le contenu de `r1` et un 1 veut dire”demander à l'utilisateur de rentrer une valeur pour `r1`”. J'utilise pour cela une fonction à part qui s'occupe de la prise en charge de ces événements.

Par exemple le programme `syracuse.asm` donne ceci comme sortie :



```

~/Doc/Repos/SIMJI main 17 ?1 ./simji run testdata/syracuse.bin -d

===== SIMJI : Simulateur de Jeu d'Instructions =====
-- Designed and Developed by Alexandre FROELICH
-- For the U.V. 4.5-Architectures numériques class
-- Contact : nightlyside@gmail.com
-- Website : https://nightlyside.github.io
=====

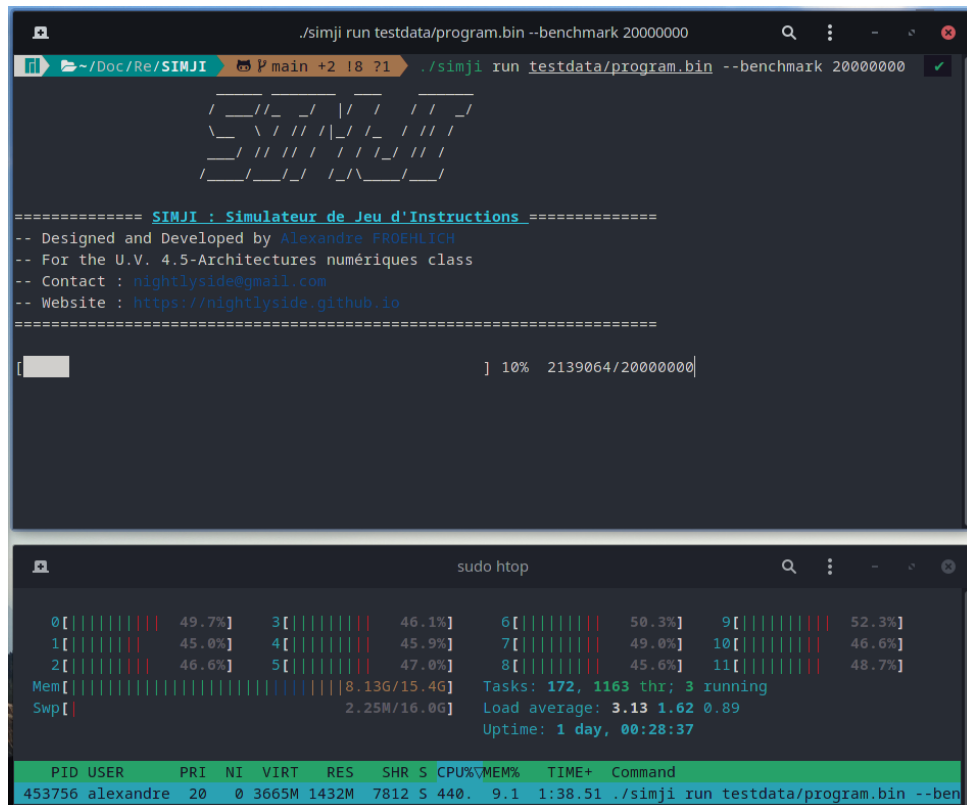
6:53AM DBG scall 0
[SCALL 0] Enter : R1 <= 4
6:53AM DBG add r3 r0 r0
6:53AM DBG scall 1
[SCALL 1] R1 => 4
6:53AM DBG sle r1 #1 r2
6:53AM DBG branz r2 #12
6:53AM DBG and r1 #1 r2
6:53AM DBG branz r2 #9
6:53AM DBG div r1 #2 r1
6:53AM DBG jmp #2 r0
6:53AM DBG scall 1
[SCALL 1] R1 => 2
6:53AM DBG sle r1 #1 r2
6:53AM DBG branz r2 #12
6:53AM DBG and r1 #1 r2
6:53AM DBG branz r2 #9
6:53AM DBG div r1 #2 r1
6:53AM DBG jmp #2 r0
6:53AM DBG scall 1
[SCALL 1] R1 => 1
6:53AM DBG sle r1 #1 r2
6:53AM DBG branz r2 #12
6:53AM DBG stop

```

**FIG. 8 :** Lancement du programme qui calcule la suite de syracuse en mode “debug”







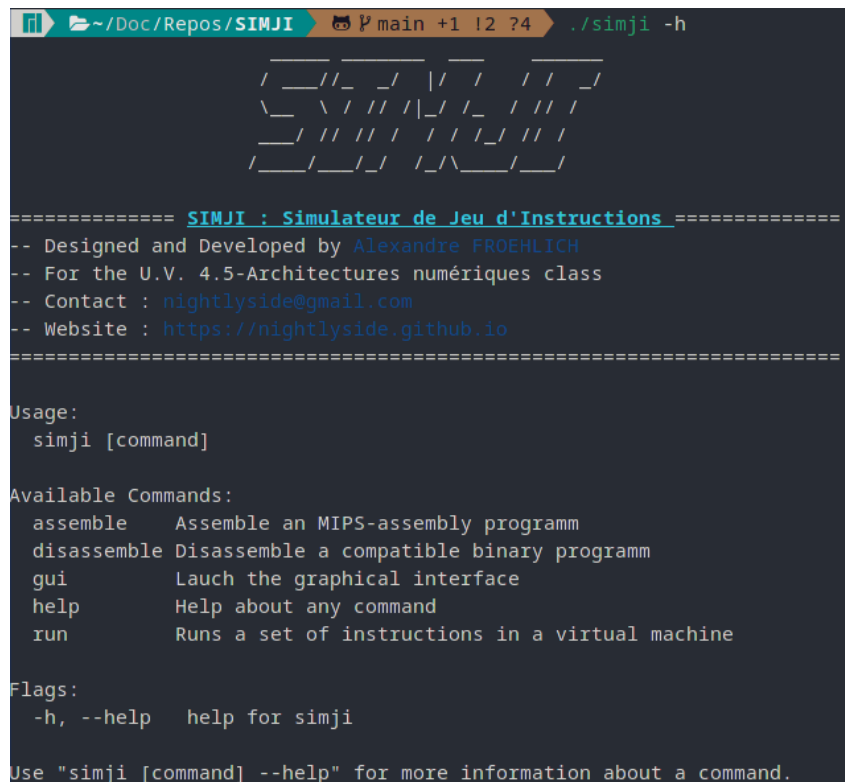
**Fig. 10 :** Benchmark d'un produit matriciel sur une matrice 3x3 avec une vue des ressources utilisées avec `htop`

On peut voir que SIMJI profite des 12 coeurs logiques de mon ordinateur pour executer le programme de test. Les résultats que j'obtiens à la fin n'indique pas d'anomalie (écart type). Il faut donc aller plus profondément dans le code et pour ça utiliser le profiling.

Pour cela golang nous fournit l'outil *pprof* qui gère toute cette partie il suffit de l'appeler depuis le code puis d'analyser les résultats avec l'outil en ligne de commande qui peut sortir un graphique similaire au suivant :



[illegible]



**Fig. 13 :** SIMJI utilise la bibliothèque Cobra pour standardiser la CLI

SIMJI fournit ainsi les commandes suivantes en CLI :

- **assemble** : permet d'assembler un programme en langage assembleur/mini-MIPS
- **disassemble** : permet de désassembler sommairement un programme assemblé par SIMJI
- **run** : permet d'exécuter un programme assemblé par SIMJI
- **gui** : permet de lancer l'interface graphique de SIMJI
- **help** : permet d'afficher l'aide du programme ou d'une commande

La plupart des commandes demandent à l'utilisateur de donner un fichier en entrée. Lorsque celui-ci est omis, un message semble au suivant est affiché dans la console :

```

~/Doc/Repos/SIMJI main +1 !3 ?4 ./simji assemble

  ____  ____  ____  ____
 /  __//_  _/  |//  /  //  /
 \__ \  //  //  |//  /  //  /
  __//  //  //  |//  /  //  /
 /___//_//  /  /_\___//  /

===== SIMJI : Simulateur de Jeu d'Instructions =====
-- Designed and Developed by Alexandre FROELICH
-- For the U.V. 4.5-Architectures numériques class
-- Contact : nightlside@gmail.com
-- Website : https://nightlside.github.io
=====

Error: accepts 1 arg(s), received 0
Usage:
  simji assemble <filename.asm> [flags]

Flags:
  -p, --cpuprofile string  Exports profiling data into the specified file
  -d, --debug              Print debugging logs of the assembly process
  -h, --help              help for assemble
  -o, --output string      Exports hex instructions to binary file

accepts 1 arg(s), received 0

```

**FIG. 14 :** SIMJI est capable d’indiquer ce qui se passe à l’utilisateur, notamment lorsqu’il oublie de préciser un fichier source

SIMJI est de plus capable d’afficher les erreurs d’exécution joliment dans la console comme par exemple une erreur d’assemblage en indiquant la ligne du programme à laquelle l’erreur s’est produite.

Enfin certaines commandes acceptent en plus des arguments de base, des “flags” qui sont des éléments supplémentaires permettant de déclencher quelques comportements différents de la commande. En voici quelques uns :

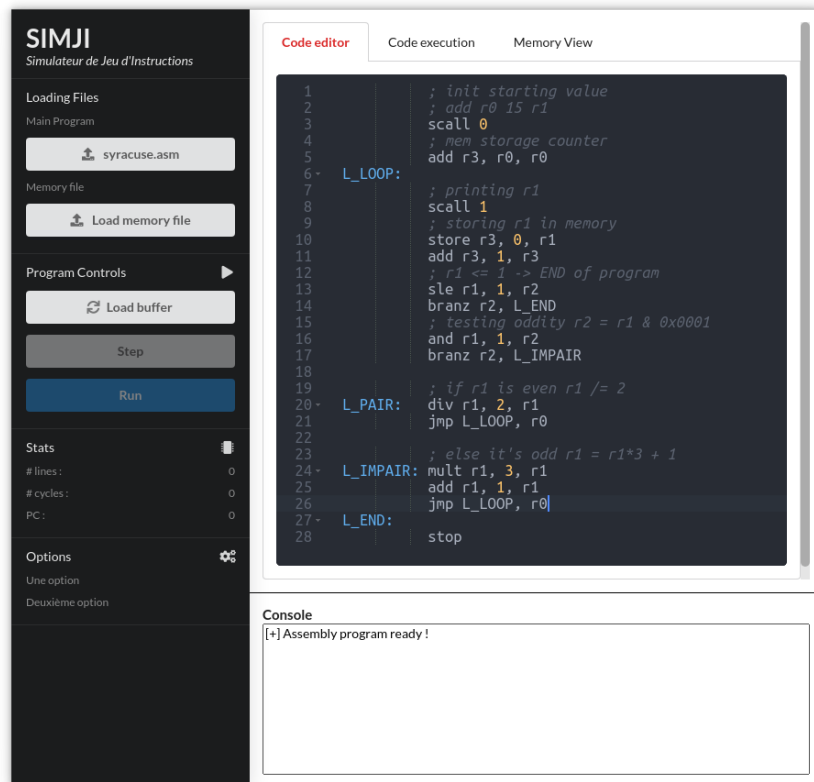
- `--debug` (en court `-d`) : permet d’afficher les informations de débogage du programme
- `--output` (en court `-o`) : permet d’enregistrer les informations de sortie dans un fichier au lieu de les afficher dans le terminal
- `--benchmark` (en court `-b`) : permet d’évaluer les performance du simulateur en exécutant un programme un certain nombre de fois
- `--cpuprofiling` (en court `-p`) : permet de sortir un fichier de profiling CPU pour chercher à optimiser l’exécution du programme

## 5.2 L’interface utilisateur (GUI)

En utilisant le flag `--gui` (ou `-g`) SIMJI affiche une interface graphique permettant d’écrire du code en assembleur, de l’exécuter et de voir l’état des registres ainsi que des cases mémoires de l’applica-

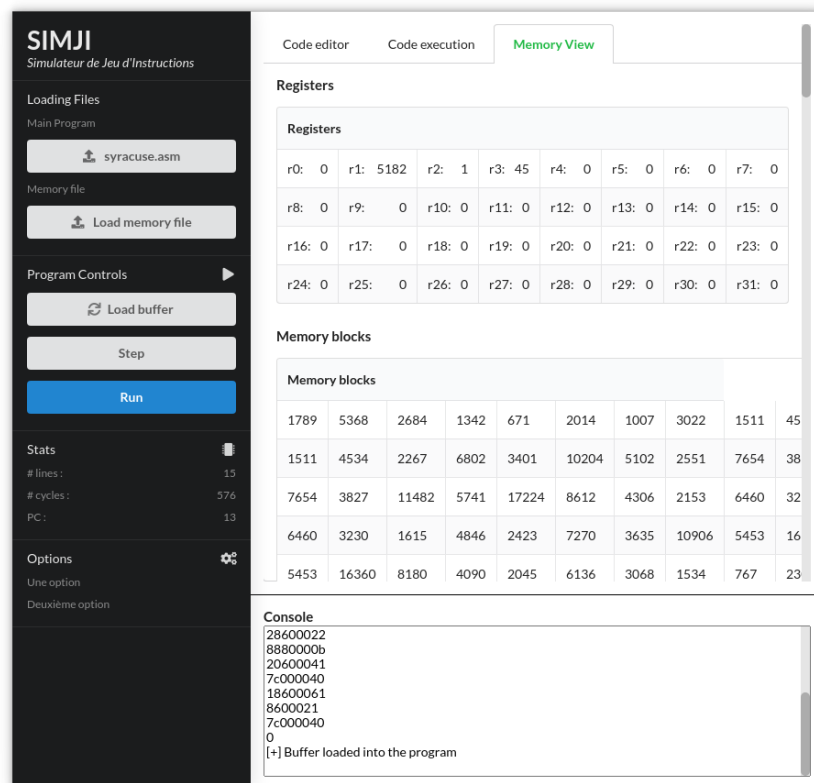
tion.

Attention l'interface graphique est l'élément de ce projet que j'ai le moins travaillé. Il est donc normal de voir des bugs, des fonctionnalités qui ne marchent pas voir même des dragons ([https://fr.wikipedia.org/wiki/Hic\\_sunt\\_dracones](https://fr.wikipedia.org/wiki/Hic_sunt_dracones))



**FIG. 15 :** En utilisant le flag `--gui` SIMJI ouvre une interface graphique permettant d'interagir avec le code

Les autres onglets affichent des informations sur le code ou encore les registres et la mémoire de la machine virtuelle. La colonne sur la gauche donne des informations sur le nombre de cycles effectués ou encore le PC.



**FIG. 16 :** L'onglet mémoire permet de voir l'état des registres et des cellules mémoires de la machine virtuelle

## 6 Conclusion

En dehors des objectifs principaux de ce cours, i.e. la découverte des architectures des processeurs et de compilation du langage, ce projet m'a permis d'apprendre un nouveau langage de programmation, la couverture par les tests unitaires (et le pourcentage de code couvert avec codecov) mais surtout comment profiler un programme afin d'en optimiser le fonctionnement.

Ces nouvelles compétences me seront bien utiles dans le futur pour écrire rigoureusement un code cohérent et juste tout en étant bien testé et possédant une bonne documentation.

Le code du projet est open source sous licence GPL et est disponible sur mon repository github ici : <https://github.com/NightlySide/SIMJI>

La documentation officielle est disponible ici : <https://pkg.go.dev/github.com/Nightlyside/simji> (pensez bien à dérouler le dossier "pkg" pour voir la documentation de chacun des packages)