SIMJI - Simulateur de Jeu d'Instructions

Un projet en Golang pour l'ENSTA Bretagne

Alexandre Froehlich

2021-03-02

Table des matières

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.

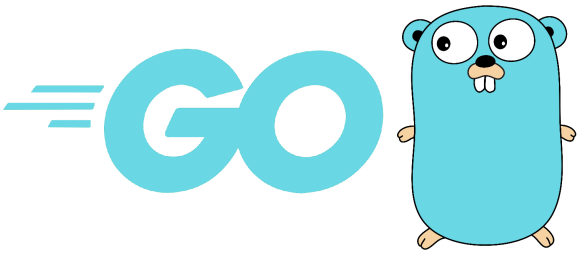

FIG. 1 : Le logo de Golang (prononcé "Go") est une marmotte bleue

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`, ..., `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

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 donné :

- 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

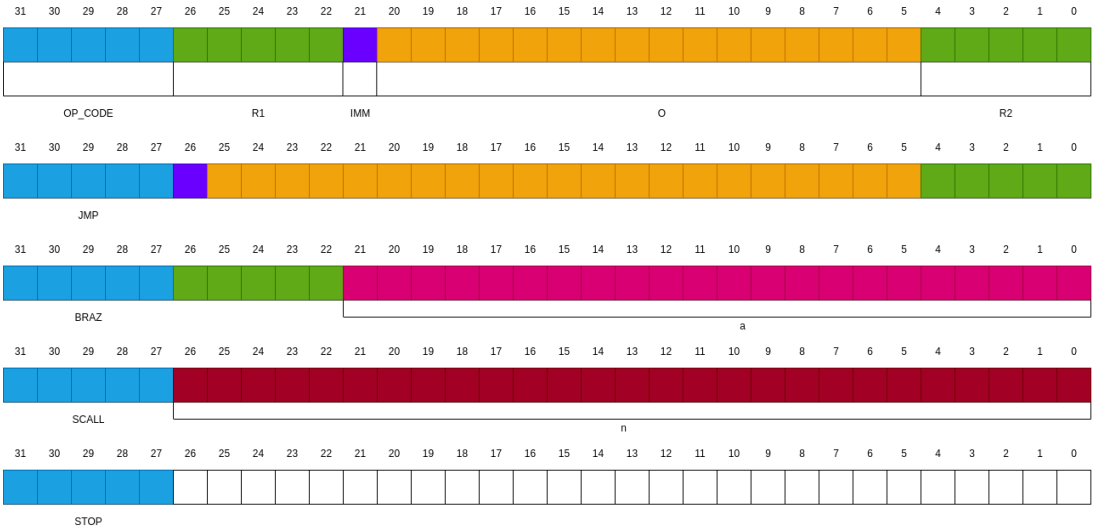

FIG. 3 : Structure des instructions du jeu mini-MIPS sur 32 bits

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 :

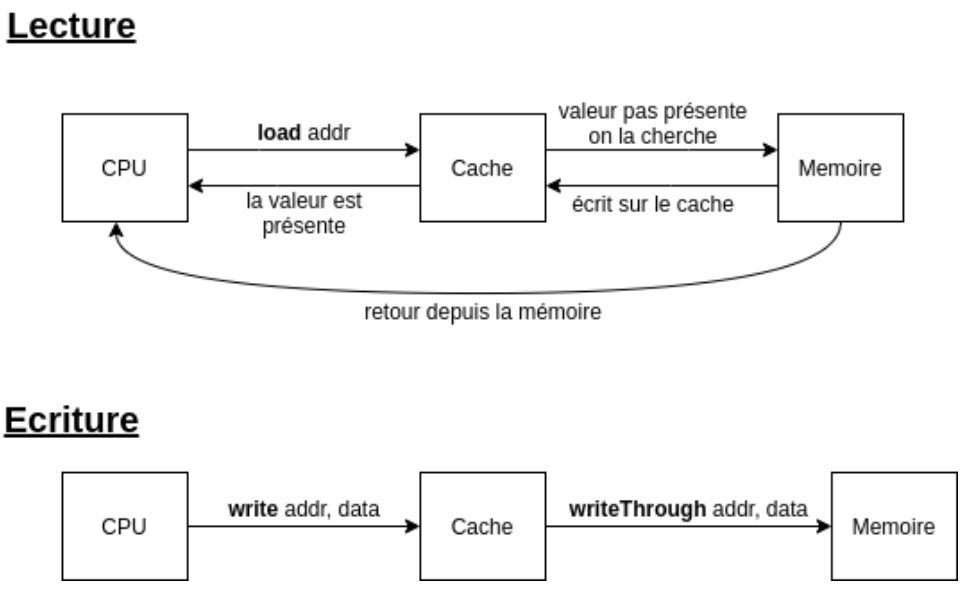

Fig. 4 : Cycle de vie du cache avec des "hits" et des "misses". Ce cache fonctionne en mode "write through" c'est a dire en écrivant à la fois dans le cache et dans la mémoire.

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 diffcultés encontrées.

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 prochaines 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 définit 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 entiére 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 :

```go
// AdressesLabels construit un dictionnaire contenant
// les adresses des labels
func AdressesLabels(lignes []string) map[string]int {
    var adresses map[string]int
    var pc int

    for _, ligne := range lignes {
        // si la ligne n'est pas viable
        if !EstVideOuCommentaire(ligne) { continue }

        label := ExtraitLabel(ligne)
        // si il y a un label on enregistre son adresse
        if label != "" { adresses[label] = pc }
        pc++
    }

    return adresses
}
```

```
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 :

```go
// EstUnRegistre permet de parser un argument et de dire si c'est un
   registre
func EstUnRegistre(argument string, labels map[string]int) (int, bool)
   {
    // on retire le "r" du registre si il est présent
    if argument[0] == 'r' {
        // on essaie de parser l'argument
        value, err := strconv.Atoi(argument[1:])
        // 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 golang 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 :

```go
func TraductionHexa(inst []int) int {
    decInstr := instr[0] << 27

    // on fait un switch sur le nombre d'arguments
    switch len(instr) {
    case 1:
        break
    case 2:
        // scall
        decInstr += instr[1] // num
        break
    case 3:
        // braz
        decInstr += instr[1] << 22 // reg
        decInstr += instr[2]       // address
    case 4:
        // jmp
        decInstr += instr[1] << 26                          // imm
        decInstr += BinaryComplement(instr[2], 21) << 5 // o
        decInstr += instr[3]                                // r
    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 :

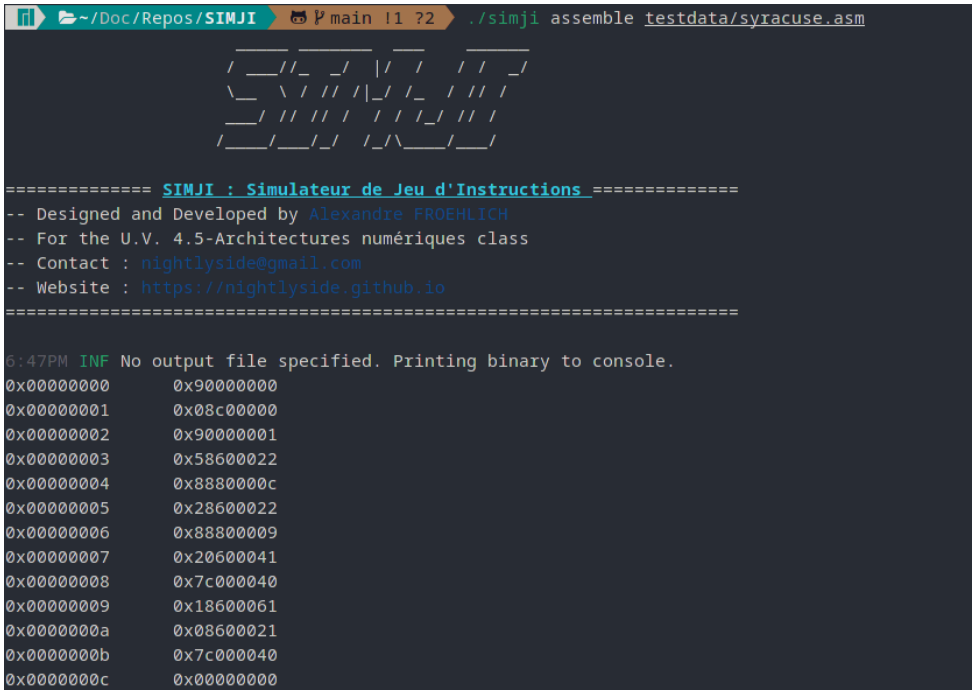

FIG. 7 : Assemblage du programme syracuse en utilisant la commande "assemble" de SIMJI

4 La machine virtuelle

4.1 Interprétation des instructions

4.2 Désassemblage

5 Le logiciel

5.1 La ligne de commande (CLI)

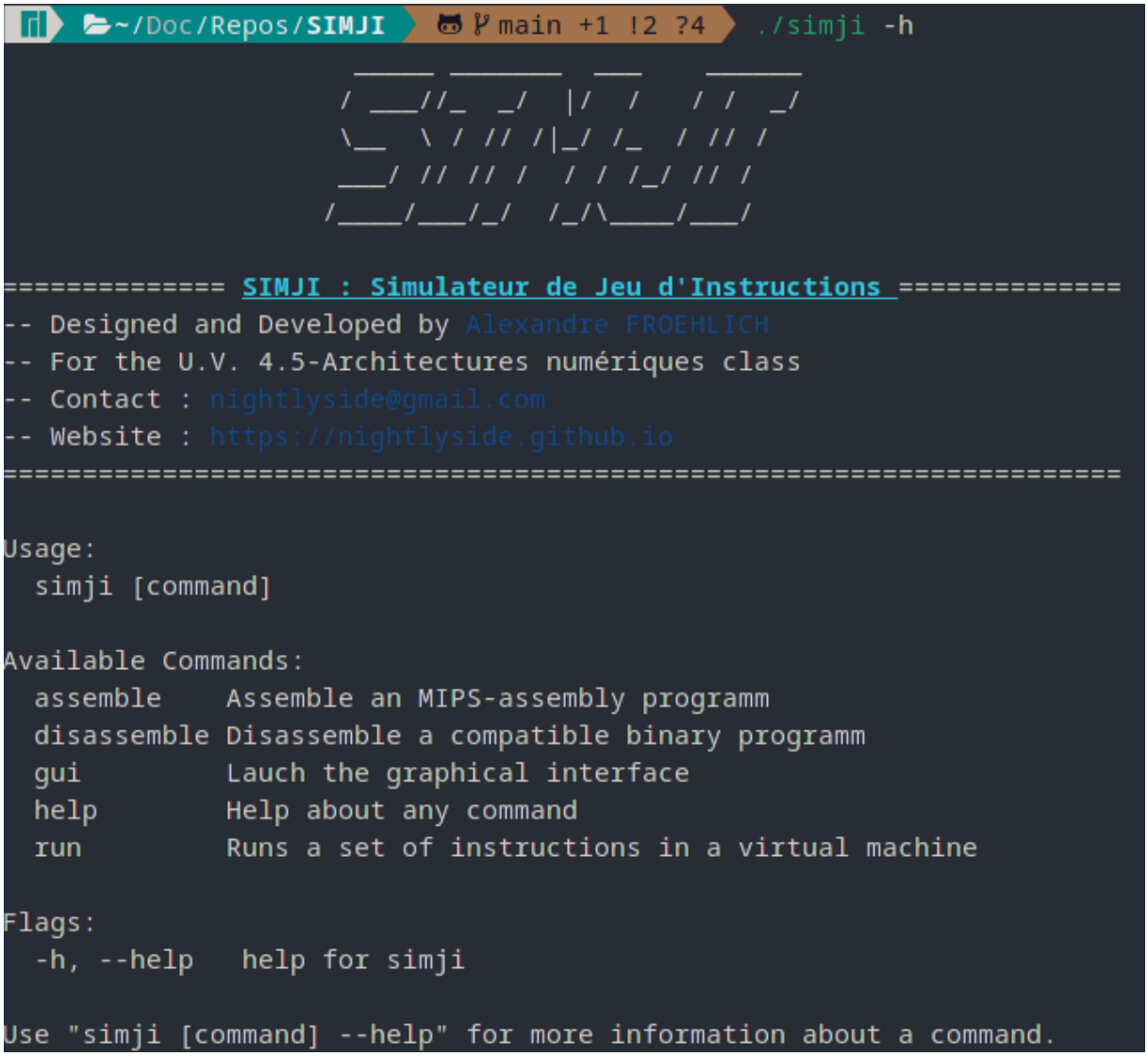

FIG. 8 : SIMJI utilise la bibliothèque Cobra pour standardiser la CLI

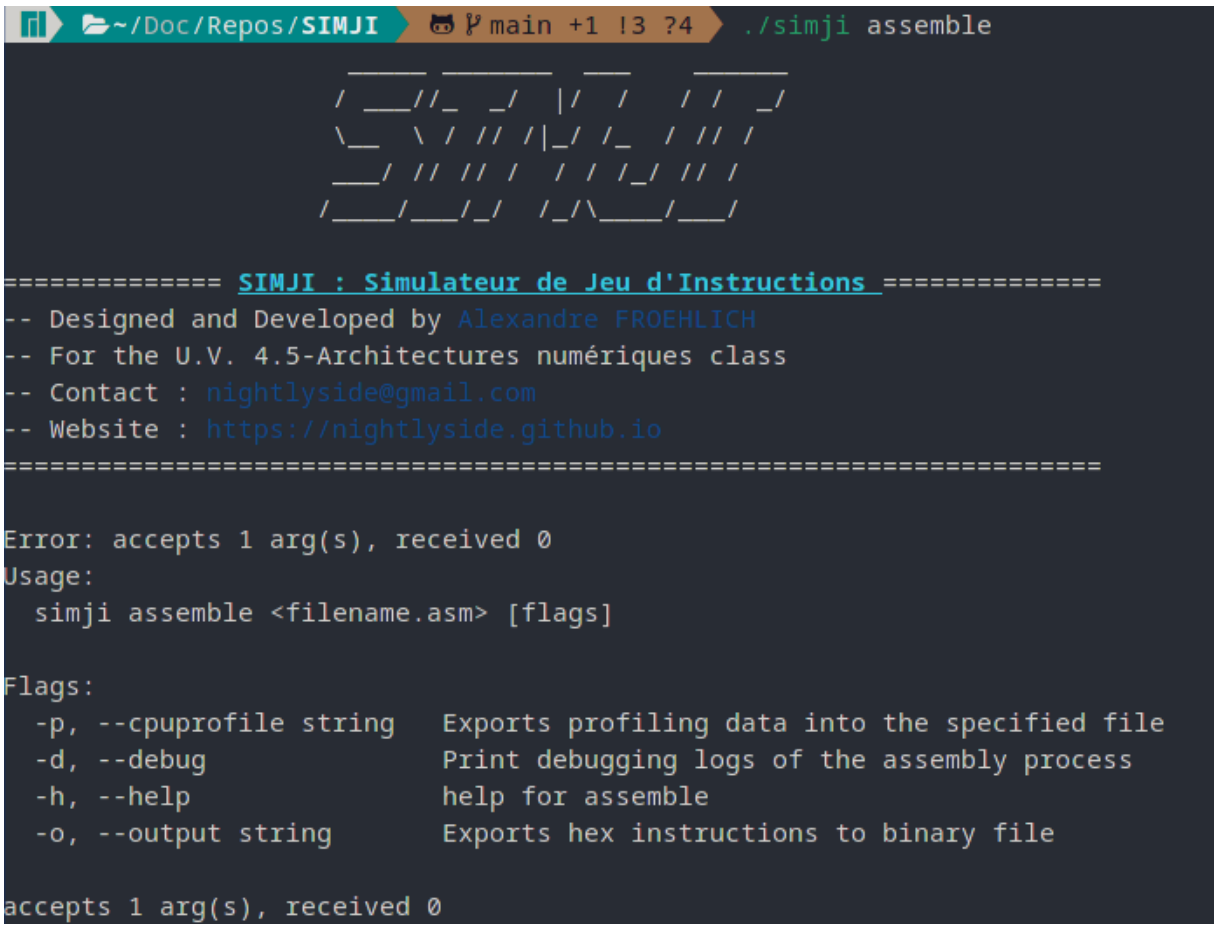

FIG. 9 : SIMJI est capable d'indiquer ce qui se passe à l'utilisateur, notamment lorsqu'il oublie de préciser un fichier source

5.2 L'interface utilisateur (GUI)

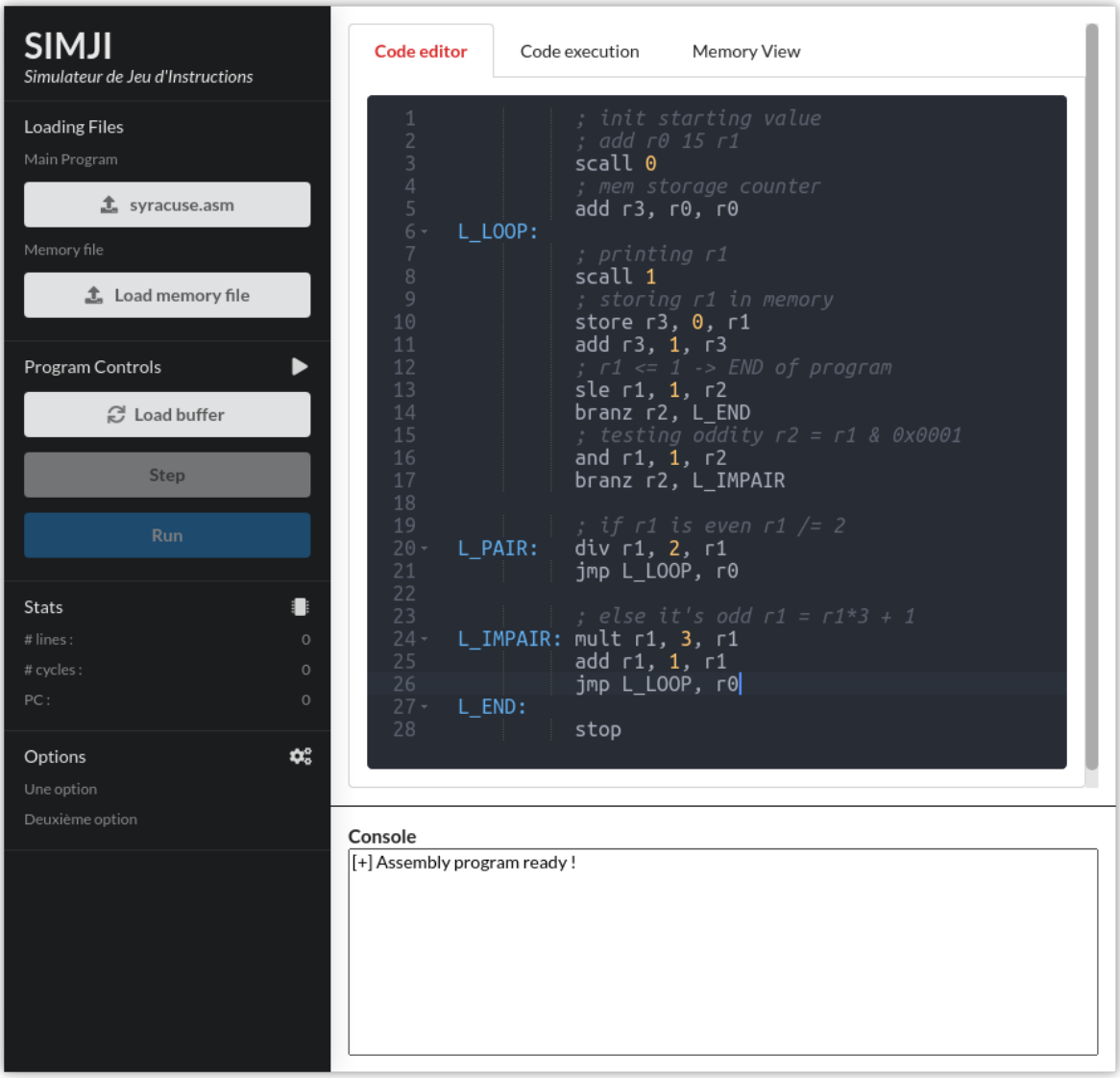

FIG. 10 : En utilisant le flag `--gui` SIMJI ouvre une interface graphique permettant d'interagir avec le code

6 Conclusion