Compilation

Éléments d'assembleur

Jules Chouquet

SOM2IF15 – 2024

Point d'étape

Nous avons terminé la vérification de types de notre programme. Si elle est positive, l'étape suivante est de le traduire en exécutable.

Nous utiliserons un langage intermédiaire (prochain cours), pour transformer notre AST en code assembleur en plusieurs étapes. Pour comprendre ce à quoi devra ressembler notre représentation intermédiaire, regardons le langage cible de notre compilateur.

Question existentielle n° 1 : Qu'est-ce que c'est ?

L'assembleur est à la fois :

- Le nom d'un langage de programmation [1]
- Le nom du programme qui traite ce langage [2]

Le langage est dit de bas niveau (il est « proche »de la machine).
Les instructions disponibles indiquent au processeur quelles données charger et lire dans les blocs de la mémoire physique.

Conséquence immédiate

Chaque assembleur (langage et programme) est indissociable du type de processeur pour lequel il est écrit. Contrairement à un langage de haut niveau qui *masque* ces aspects et ne varie pas d'une architecture à l'autre.

1. Dans ce contexte, remplacez « un »par « une famille de »(et accordez !)
2. D'après vous, est-ce un compilateur ou un interpréteur ?

Remarque

Les instructions de l'assembleur correspondent à des instructions du langage machine pour le processeur, mais ce n'est *pas* du langage machine.

C'est en revanche le langage le plus facile à traduire en langage machine.

Remarque

Les instructions de l'assembleur correspondent à des instructions du langage machine pour le processeur, mais ce n'est *pas* du langage machine.

C'est en revanche le langage le plus facile à traduire en langage machine.

Autre remarque

C'est un langage *Turing-Complet* : n'importe quel algorithme peut être écrit en assembleur [a]

a. Par contre ce n'est pas toujours une bonne idée...

Dans le cours et dans les travaux pratiques, nous utiliserons l'assembleur MIPS32. Au moins quatre raisons motivent ce choix :

- Il est de la famille RISC (*Reduced Instruction Set*), qui est de plus « bas niveau »que d'autres (notamment que les CISC : *Complex Instruction Set*). Il demande plus de travail au compilateur.
- Il est représentatif des processeurs modernes ; en général présente une meilleure efficacité (on utilise plus les registres que la mémoire, les opérations élémentaires sont plus rapides,...)
- On peut l'exécuter sur des simulateurs[3] : il devient plus simple de travailler à plusieurs sur des machines différentes avec le même code assembleur.
- C'est le seul que j'ai appris.

3. Téléchargeables ou en ligne. Liens sur Celene.

À quoi ça ressemble ?

[demo : add.s]

Question existentielle n° 2 : Pourquoi ?

- L'assembleur nous permet de « parler »avec la machine, tout en restant dans le logiciel [4].

4. Pour lui parler de plus près, on arrive vite à des considérations matérielles.

L'assembleur dans la vie

- Meilleure efficacité et prédictibilité du code : parfois plus fiable.
- Toutes les machines disposent en général d'un assembleur. Ce n'est pas toujours le cas pour le compilateur/interpréteur d'un langage de haut niveau.
- Parler l'assembleur peut permettre de faire des optimisations à la main (remplacer une boucle par une instruction unique est parfois possible, sans que le compilateur le détecte.)

L'assembleur dans la vie

- Meilleure efficacité et prédictibilité du code : parfois plus fiable.
- Toutes les machines disposent en général d'un assembleur. Ce n'est pas toujours le cas pour le compilateur/interpréteur d'un langage de haut niveau.
- Parler l'assembleur peut permettre de faire des optimisations à la main (remplacer une boucle par une instruction unique est parfois possible, sans que le compilateur le détecte.)

Difficultés posées par l'assembleur

Relativement évidentes, ou vont le devenir rapidement.

L'assembleur dans un cours de compilation

On considèrera que l'on a écrit un compilateur complet quand on aura une traduction du programme source vers l'assembleur.

L'assembleur dans un cours de compilation

On considèrera que l'on a écrit un compilateur complet quand on aura une traduction du programme source vers l'assembleur.

Pourquoi on s'arrête avant le langage machine ?

En gros : la dernière étape vers l'assembleur n'est pas la plus compliquée à reproduire et adapter. (Mais avec le langage machine, ce serait infernal).

Question existentielle n° 3 : Comment ?

Avant d'écrire un premier programme, il nous faut :

- La syntaxe globale d'un fichier d'assembleur (.asm ou .s)
- Les instructions disponibles
- Des informations sur l'utilisation des registres
- La connaissance des identifiants de quelques appels système.

Pour écrire des programme moins élémentaire il nous faudra :

- Maîtriser la représentation de la mémoire physique gérée par l'assembleur.
- Plus d'informations sur l'utilisation des registres.

Question existentielle n° 3 : Comment ?

Avant d'écrire un premier programme, il nous faut :

- La syntaxe globale d'un fichier d'assembleur (.asm ou .s)
- Les instructions disponibles
- Des informations sur l'utilisation des registres
- La connaissance des identifiants de quelques appels système.

Pour écrire des programme moins élémentaire il nous faudra :

- Maîtriser la représentation de la mémoire physique gérée par l'assembleur.
- Plus d'informations sur l'utilisation des registres.

Remarque importante

Ce cours ne présente pas l'intégralité des commandes MIPS. Il est indispensable de consulter la documentation et d'explorer le reste du langage (liens sur Celene).

Retour sur l'addition

[add1.s]

Instruction	**Effet**	**Sens** [5]
`li reg, n`	charge l'entier *n* dans `reg`	*load immediate (=integer)*
`syscall`	exécute l'appel système dont l'identifiant est dans `$v0`	*system call*

syscall	**effet**
1	imprime l'entier chargé en `$a0`
5	charge en `$v0` l'entier de l'entrée standard
10	arrêt

5. On parle aussi de *mnémonique* pour les commandes, dont le nom est supposé être simple à se rappeler.

Un mot Bonjour en MIPS

[hw.s]

Instruction	**Effet**	**Sens**
`la reg, a`	charge l'adresse *a* dans `reg`	*load adress*

syscall	**effet**
4	imprime la chaîne stockée à l'adresse chargée en $a0

Et pour programmer vraiment ?

Il faut avoir en tête à quoi ressemble la mémoire, pour stocker des données ailleurs que dans les registres (et les retrouver).
Chaque instruction est représentée en mémoire sur 4 octets (car on est sur une architecture 32 bits).

Exemple

```
000100 01000 00111 0000000000000011
addi    $8    $7         3
```

(car dans ce système, les constantes directes sont stockées sur 16 bits)

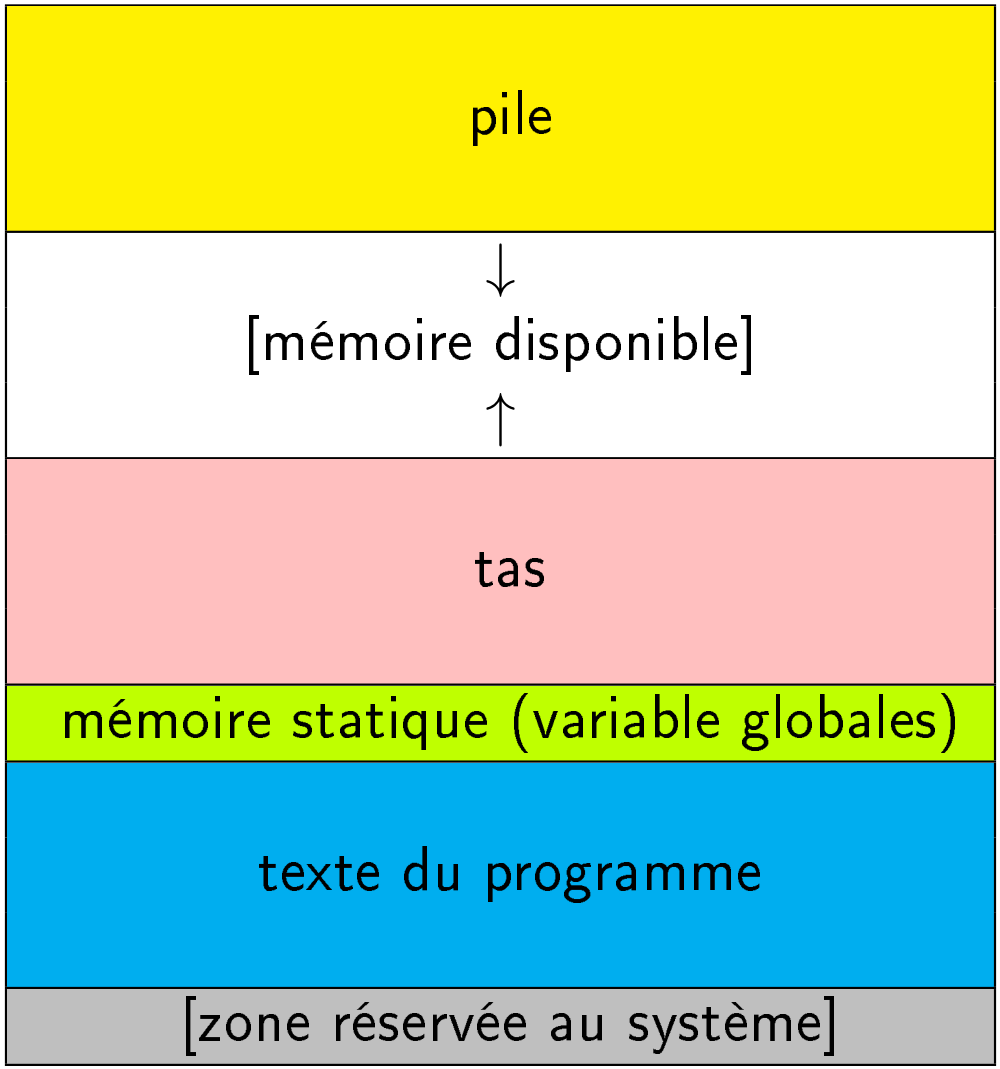
Utilisation des registres

$zero	valeur 0
$v0,$v1	Résultat des fonctions, et appels système
$a0...$a3	Arguments des fonctions
$t0...$t9	Registres temporaires
$s0...$s7	Registres temporaires, mais préservés après les appels de fonctions.
$sp	Pointe sur le sommet de la pile.
$gp	Pointe sur la zone de mémoire globale
$fp	Pointe sur la trame (*frame pointer*)
$ra	Adresse de retour (peut être sauvegardée avant un saut de procédure)

Remarque

Ces utilisations sont des conventions mais pas seulement : les registres sont matériellement optimisés pour des données persistantes ou non, selon l'usage prévu.

Structure


pile
[mémoire disponible]
tas
mémoire statique (variable globales)
texte du programme
[zone réservée au système]

Calcul de Fibonacci

(Récursion terminale avec mémoisation : on s'en sort avec seulement quelques registres pour mémoire)

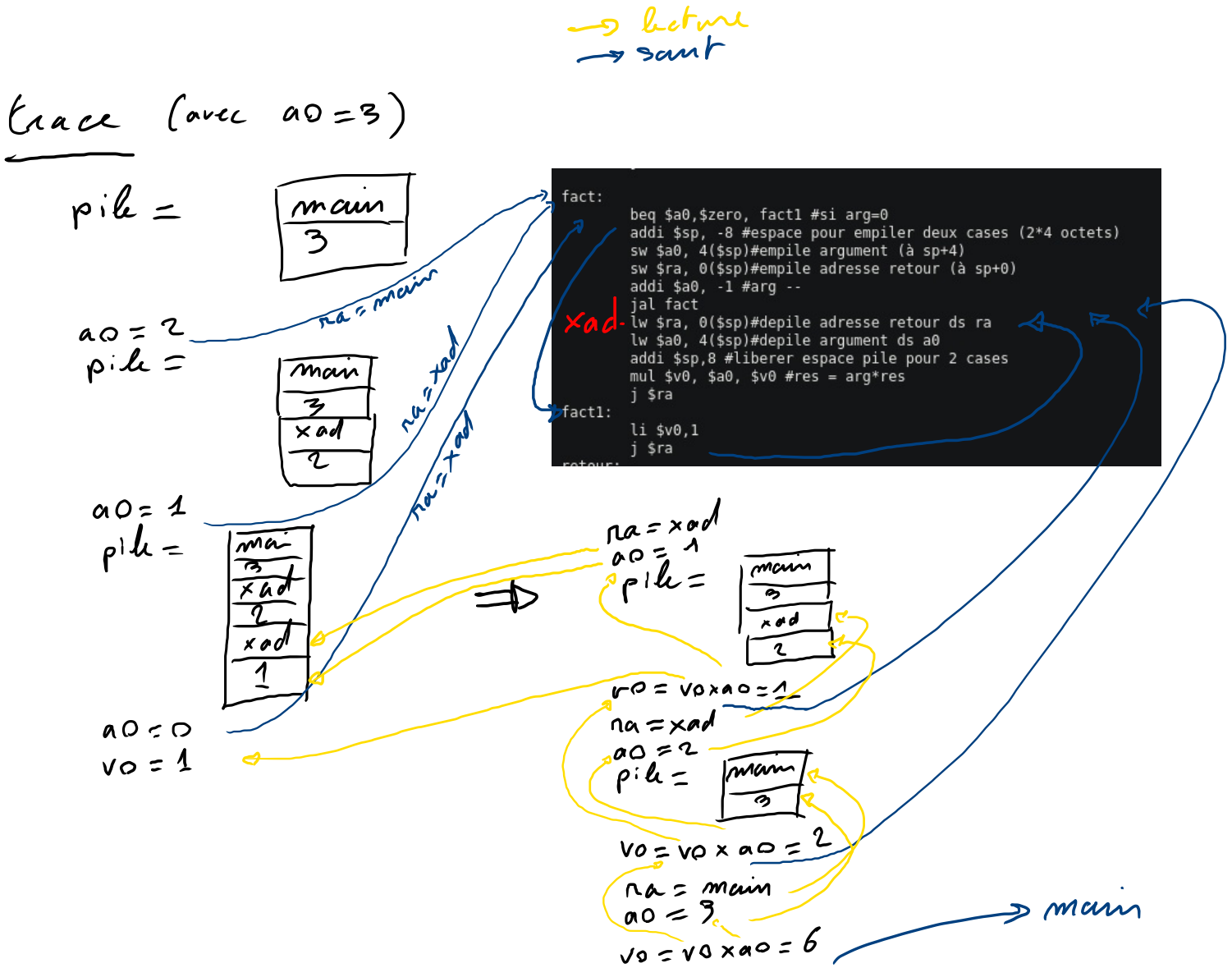
[demo (fibo.s)]

Et un programme récursif non terminal ?

La factorielle $\rightarrow$ Il faut utiliser la pile.[demo (fact.s)]

Et un programme récursif non terminal ?

La factorielle → Il faut utiliser la pile.[demo (fact.s)]



Instructions, expressions,. . .

À quel **langage** a-t-on affaire ?

6. Machine à pile -> Prochain TP.

Instructions, expressions,. . .

À quel **langage** a-t-on affaire ?

On n'a pas d'expressions composées : une expression est une constante ou une adresse.

Les instructions ne sont pas composées non plus. À quoi ressemblerait l'AST d'un programme en MIPS32 ?

Comment traduire l'évaluation d'expressions complexes dans ce contexte ? [6]

6. Machine à pile -> Prochain TP.

Conclusions sur l'assembleur

- C'est un langage de programmation pénible mais intéressant : on donne des instructions à la machine de façon plus directe que d'habitude.
- Pour le reste du cours : il est important de comprendre ce que l'on peut faire directement ou non en assembleur.

Devoirs

- Lisez les documentations pour maîtriser le reste du langage et les conventions.
- Traduisez quelques programmes simples en assembleur.
- Générez du code assembleur à partir de votre langage préféré, en traduisant quelques instructions basiques.