

Université Paris-Sud

# Compilation et langages

Thibaut Balabonski  
d'après Jean-Christophe Filliâtre @ ENS

Cours 1 / 16 septembre 2016

```
Closure 33, 0. Push. Closurerec 12, 0.  
Const [0: 2 [0: 3 [0 : 4 0a]]]. Push. Const 1.  
    Push. Acc 3.  
    Push. Acc 3.  
    Appterm 3, 6. Restart.  
    Grab 2. Acc 2. Branchifnot 30. Acc 2.  
    Getfield 1. Push. Acc 3.  
    Getfield 0. Push. Acc 3.  
    Push. Acc 3.  
    Apply 2. Push. Acc 2. Push. Offsetclosure 0.  
    Appterm 3, 6. Acc 1. Return 3. Restart.  
    Grab 1. Acc 1. Push. Acc 1.  
    Mulint. Return 2.
```

```
List.fold_left (fun acc n -> acc * n) 1 [2; 3; 4]
```

- Mieux connaître les **langages de programmation**, en comprenant la manière dont ils fonctionnent.
- Connaître les mécanismes de base de la **compilation**, c'est-à-dire de la traduction d'un langage dans un autre.
- **Construire** un compilateur.

## Les techniques vues dans ce cours sont aussi...

... utiles pour concevoir des outils qui manipulent des programmes de manière symbolique, comme les outils de :

- *preuve de programmes* (VCC, Dafny, Spec#, Frama-C, Spark, GNATProve, Why3, Boogie, etc.),
- *vérification par model checking* (Slam, Spin, CBMC, Murphi, Cubicle, Blast, Uppaal, Java Pathfinder, etc.),
- *analyse par interprétation abstraite* (Astrée, Polyspace, etc.),
- *démonstration automatique* (Z3, CVC4, Alt-Ergo, etc.),
- *test formel* (Pex, PathCrawler, etc.)
- ...

Toutes ces thématiques seront abordées dans le **M2 FIIL** (Fondements de l'Informatique et Ingénierie du Logiciel).

Ici on programme :

- en cours
- en TD/TP
- chez soi
- à l'examen

Et tout ça, en **Objective Caml**.

Point.

- **Cours**

- Vendredi, 8h30–10h30, petit amphi (PUIO).
- Pas de photocopié, mais transparents disponibles.

- **TD/TP**

- Vendredi 10h45–12h45.
- Surtout TP.
- Préparation du TP sur la fin de l'amphi.
- Contrôle continu.

- **Défis de programmation**

- Chaque semaine, pour découvrir la problématique du cours suivant.

- **MCC**

- 1ère session : 0.5 CC1 + 0.5 EX1
- 2ème session : 0.5 CC1 + 0.5 EX2
- Pas de partiel.

Le site web du cours :

<http://www.lri.fr/~blsk/Compilation/>

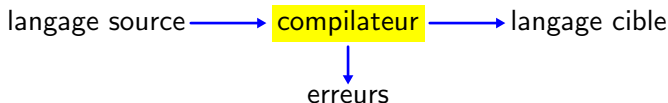
Ces supports de cours sont construits principalement à partir de ceux de **Jean-Christophe Filliâtre**.

Certains éléments des cours et TP viennent également de **Sylvain Conchon**, **François Pottier**, **Yann Régis-Gianas** et **Kim Nguyen**.



# Compilation : le Grand Plan

Schématiquement, un compilateur est un programme qui traduit un « programme » d'un langage **source** vers un langage **cible**, en signalant d'éventuelles erreurs



# Compilation vers le langage machine

Quand on parle de compilation, on pense typiquement à la traduction d'un langage de haut niveau (C, Java, OCaml, ...) vers le langage machine d'un processeur (Intel Pentium, PowerPC, ...)

```
% gcc -o sum sum.c
```

source `sum.c` → **compilateur C (gcc)** → exécutable `sum`

```
int main(int argc, char **argv) {  
    int i, s = 0;  
    for (i = 0; i <= 100; i++) s += i*i;  
    printf("0*0+...+100*100 = %d\n", s);  
}
```

```
00100111101111011111111111111100000  
1010111110111111100000000000010100  
101011111010010000000000000100000  
101011111010010100000000000100100  
101011111010000000000000000011000  
101011111010000000000000000011100  
100011111010111000000000000011100  
...
```

Dans ce cours, nous allons effectivement nous intéresser à la compilation vers de **l'assembleur**, mais ce n'est qu'un aspect de la compilation.

Un certain nombre de techniques mises en œuvre dans la compilation ne sont pas liées à la production de code assembleur.

Certains langages sont d'ailleurs :

- interprétés (Basic, COBOL, Ruby, Python, etc.)
- compilés dans un langage intermédiaire qui est ensuite interprété (Java, OCaml, etc.)
- compilés vers un autre langage de haut niveau
- compilés à la volée

# Différence entre compilateur et interprète

Un **compilateur** traduit un programme  $P$  en un programme  $Q$  tel que pour toute entrée  $x$ , la sortie de  $Q(x)$  soit la même que celle de  $P(x)$

$$\forall P \exists Q \forall x \dots$$

Un **interprète** est un programme qui, étant donné un programme  $P$  et une entrée  $x$ , calcule la sortie  $s$  de  $P(x)$

$$\forall P \forall x \exists s \dots$$

# Différence entre compilateur et interprète

Dit autrement,

Le compilateur fait un travail complexe **une seule fois**, pour produire un code fonctionnant pour n'importe quelle entrée

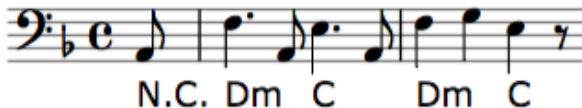
L'interprète effectue un travail plus simple, mais le refait sur chaque entrée

Autre différence : le code compilé est généralement bien plus efficace que le code interprété

# Exemple de compilation et d'interprétation

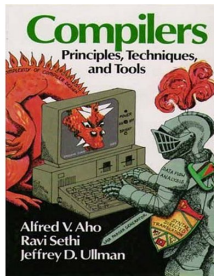
source → lilypond → fichier PostScript → gs → image

```
<<  
  \new Staff { \clef bass \key d \minor \time 4/4  
               \partial 8 a,8 f4. a,8 e4. a,8 f4 g e4 r8 }  
  \chords{ r8 d2:m c d:m c }  
>>
```



À quoi juge-t-on la qualité d'un compilateur ?

- à sa correction
- à l'efficacité du code qu'il produit
- à sa propre efficacité

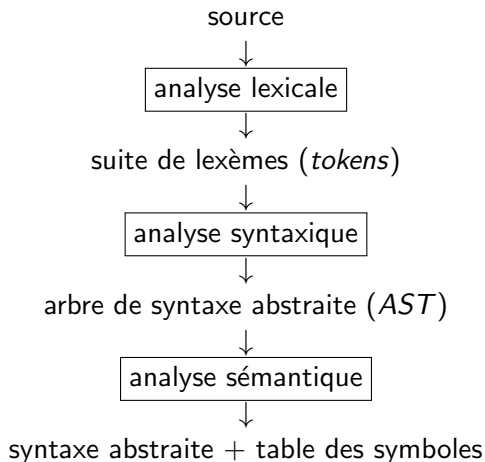


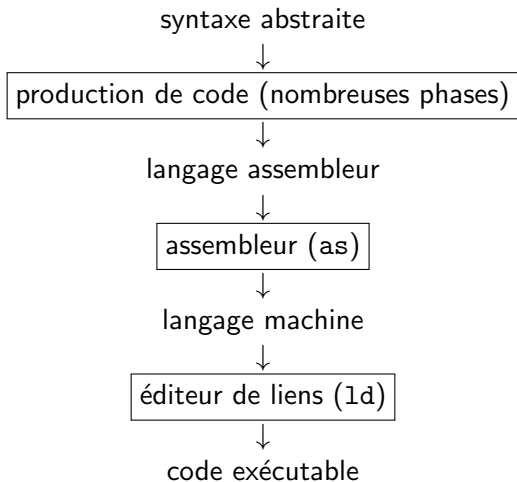
*"Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free! Thus, the most important objective in writing a compiler is that it is correct."*  
(Dragon Book, 2006)



Typiquement, le travail d'un compilateur se compose

- d'une phase d'**analyse**
  - reconnaît le programme à traduire et sa signification
  - signale les erreurs et peut donc échouer (erreurs de syntaxe, de portée, de typage, etc.)
- puis d'une phase de **synthèse**
  - production du langage cible
  - utilise de nombreux langages intermédiaires
  - n'échoue pas

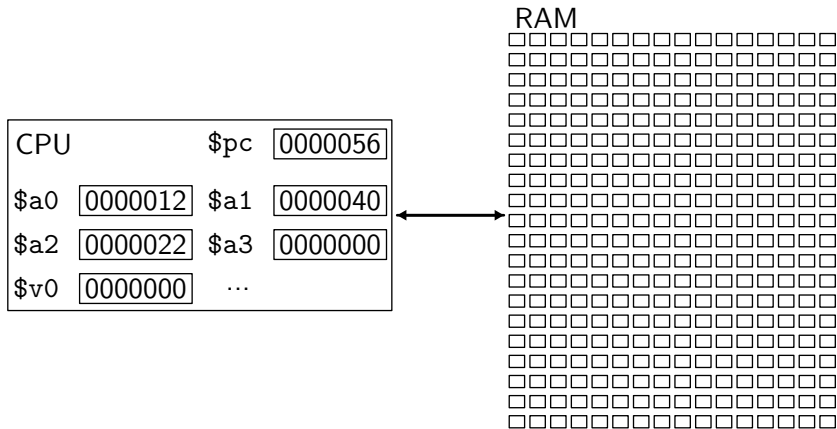




# L'architecture cible

Très schématiquement, un ordinateur est composé :

- d'une unité de calcul (CPU), contenant
  - un petit nombre de registres entiers ou flottants
  - des capacités de calcul
- d'une mémoire vive (RAM)
  - composée d'un très grand nombre d'octets (8 bits)  
par exemple, 1 Go =  $2^{30}$  octets =  $2^{33}$  bits, soit  $2^{2^{33}}$  états possibles
  - qui contiennent des données et des instructions



L'accès à la mémoire coûte cher (à un milliard d'instructions par seconde, la lumière ne parcourt que 30 centimètres entre 2 instructions !)

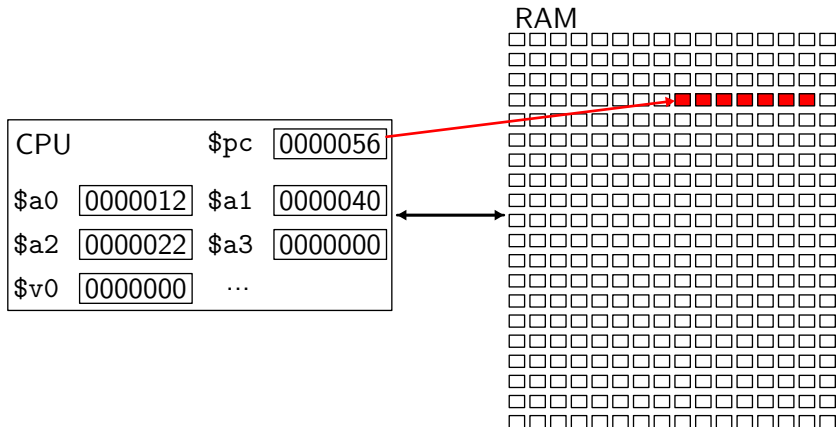
La réalité est bien plus complexe :

- plusieurs (co)processeurs, dont certains dédiés aux flottants
- un ou plusieurs caches
- une virtualisation de la mémoire (MMU)
- etc.

Schématiquement, l'exécution d'un programme se déroule ainsi :

- un registre (\$pc) contient l'adresse de l'instruction à exécuter
- on lit les 4 (ou 8) octets à cette adresse (*fetch*)
- on interprète ces bits comme une instruction (*decode*)
- on exécute l'instruction (*execute*)
- on modifie le registre \$pc pour passer à l'instruction suivante (typiquement celle se trouvant juste après, sauf en cas de saut)





instruction : 

001000	00110	00101	00000000000001010
--------	-------	-------	-------------------

décodage :     addi        \$a2        \$a1                    10

i.e. ajouter 10 au registre \$a2 et stocker le résultat dans le registre \$a1

Là encore la réalité est bien plus complexe

- pipelines
  - plusieurs instructions sont exécutées en parallèle
- prédiction de branchement
  - pour optimiser le pipeline, on tente de prédire les sauts conditionnels

Deux grandes familles de microprocesseurs

- CISC (*Complex Instruction Set*)
  - beaucoup d'instructions
  - beaucoup de modes d'adressage
  - beaucoup d'instructions lisent / écrivent en mémoire
  - peu de registres
  - exemples : VAX, PDP-11, Motorola 68xxx, Intel x86
- RISC (*Reduced Instruction Set*)
  - peu d'instructions, régulières
  - très peu d'instructions lisent / écrivent en mémoire
  - beaucoup de registres, uniformes
  - exemples : Alpha, Sparc, MIPS, ARM

On choisit **MIPS** pour ce cours.

# L'assembleur MIPS

(voir la documentation sur la page du cours)

- 32 registres, \$0 à \$31
  - \$0 contient toujours 0
  - utilisables sous d'autres noms, correspondant à des conventions (zero, at, v0–v1, a0–a3, t0–t9, s0–s7, k0–k1, gp, sp, fp, ra)
- trois types d'instructions
  - instructions de transfert, entre registres et mémoire
  - instructions de calcul
  - instructions de saut

Documentation : sur le site du cours

Registres **réservés** à l'assembleur et à l'OS (ne pas toucher) :

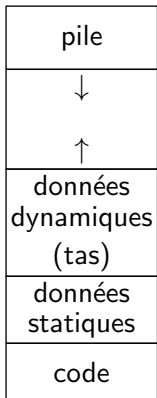
- \$at, \$k0 et \$k1

Registres contenant des **pointeurs** :

- \$sp et \$fp vers la **mémoire dynamique**
- \$gp vers la zone des **données statiques**
- \$ra vers la zone de **code** (adresse de retour)

Registres utilisables pour les **valeurs** intermédiaires des calculs :

- Le reste : \$a0-\$a3, \$v0-\$v1, \$t0-\$t9 et \$s0-\$s7
- Nous verrons les conventions plus tard



La **pile** est stockée tout en haut, et croît dans le sens des adresses décroissantes ; `$sp` pointe sur le sommet de la pile

Les données dynamiques (survivant aux appels de fonctions) sont allouées sur le **tas** (éventuellement par un GC), en bas de la zone de données, juste au dessus des données statiques

Ainsi, on ne se marche pas sur les pieds

En pratique, on utilisera un simulateur MIPS : **MARS** (ou **SPIM**).

En ligne de commande :

- `java -jar Mars_4_2.jar file.asm`

En mode graphique et interactif :

- `java -jar Mars_4_2.jar`
- charger le fichier et l'assembler
- mode pas à pas, avec visualisation des registres et de la mémoire

Documentation : sur le site du cours



On ne programme pas en langage machine mais en assembleur

L'assembleur fournit un certain nombre de facilités :

- étiquettes symboliques
- allocation de données globales
- pseudo-instructions

Le langage assembleur est transformé en langage machine par un programme appelé également **assembleur** (c'est un compilateur)

## La directive

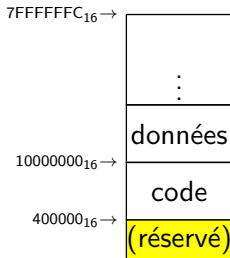
```
.text
```

indique que des instructions suivent, et la directive

```
.data
```

indique que des données suivent

Le code sera chargé à partir de l'adresse  $0x400000$   
et les données à partir de l'adresse  $0x10000000$



Une étiquette symbolique est introduite par

```
label :
```

```
        .text
main:   li      $v0, 4      # code de print_string
        la      $a0, hw    # adresse de la chaîne
        syscall          # appel système
        li      $v0, 10    # exit
        syscall
        .data
hw:     .asciiz "hello world\n"
```

(.asciiz est une facilité pour .byte 104, 101, ... 0)

# Jeu d'instructions : constantes, adresses et copies

- chargement d'une constante dans un registre

```
li    $a0, 42      # a0 <- 42
li    $a0, -65536   # a0 <- -65536
```

- chargement de l'adresse représentée par une étiquette

```
la    $a0, label
```

- copie d'un registre dans un autre

```
move  $a0, $a1     # copie a1 dans a0 !
```

- addition de deux registres

```
add  $a0, $a1, $a2  # a0 <- a1 + a2
add  $a2, $a2, $t5   # a2 <- a2 + t5
```

de même, sub, mul, div

- addition d'un registre et d'une constante

```
addi $a0, $a1, 42   # a0 <- a1 + 42
```

(mais pas subi, muli ou divi !)

- négation

```
neg  $a0, $a1       # a0 <- -a1
```

- valeur absolue

```
abs  $a0, $a1       # a0 <- |a1|
```

# Jeu d'instructions : opérations sur les bits

- NON logique ( $\text{not}(100111_2) = 011000_2$ )

```
not  $a0, $a1      # a0 <- not(a1)
```

- ET logique ( $\text{and}(100111_2, 101001_2) = 100001_2$ )

```
and  $a0, $a1, $a2  # a0 <- and(a1, a2)
andi $a0, $a1, 0x3f # a0 <- and(a1, 0...0111111)
```

- OU logique ( $\text{or}(100111_2, 101001_2) = 101111_2$ )

```
or   $a0, $a1, $a2  # a0 <- or(a1, a2)
ori  $a0, $a1, 42    # a0 <- or(a1, 0...0101010)
```

- décalage à gauche (insertion de zéros)

```
sll  $a0, $a1, 2    # a0 <- a1 * 4  
sllv $a1, $a2, $a3  # a1 <- a2 * 2^a3
```

- décalage à droite arithmétique (copie du bit de signe)

```
sra  $a0, $a1, 2    # a0 <- a1 / 4
```

- décalage à droite logique (insertion de zéros)

```
srl  $a0, $a1, 2
```

- rotation

```
rol  $a0, $a1, 2  
ror  $a0, $a1, 3
```

- comparaison de deux registres

```
slt    $a0, $a1, $a2    # a0 <- 1 si a1 < a2  
                        #      0 sinon
```

ou d'un registre et d'une constante

```
slti   $a0, $a1, 42
```

- variantes : sltu (comparaison non signée), sltiu
- de même : sle, sleu / sgt, sgtu / sge, sgeu
- égalité : seq, sne



- lire un mot (32 bits) en mémoire

```
lw    $a0, 42($a1)    # a0 <- mem[a1 + 42]
```

l'adresse est donnée par un registre et un décalage sur 16 bits signés

- variantes pour lire 8 ou 16 bits, signés ou non (lb, lh, lbu, lhu)

# Jeu d'instructions : transfert (écriture)

- écrire un mot (32 bits) en mémoire

```
sw    $a0, 42($a1)    # mem[a1 + 42] <- a0  
                        # attention au sens !
```

l'adresse est donnée par un registre et un décalage sur 16 bits signés

- variantes pour écrire 8 ou 16 bits (sb, sh)

on distingue

- **branchement** : typiquement un saut conditionnel, dont le déplacement est stocké sur 16 bits signés (-32768 à 32767 instructions)
- **saut** : saut inconditionnel, dont l'adresse de destination est stockée sur 26 bits

- branchement conditionnel

```
beq  $a0, $a1, label # si a0 = a1 saute à label  
                        # ne fait rien sinon
```

- variantes : bne, blt, ble, bgt, bge (et comparaisons non signées)
- variantes : beqz, bnez, bgez, bgtz, bltz, blez

## saut inconditionnel

- à une adresse (*jump*)

```
j    label
```

- avec sauvegarde de l'adresse de l'instruction suivante dans \$ra

```
jal  label    # jump and link
```

- à une adresse contenue dans un registre

```
jr   $a0
```

- avec l'adresse contenue dans \$a0 et sauvegarde dans \$a1

```
jalr $a0, $a1
```

Quelques appels système fournis par une instruction spéciale

```
syscall
```

Le code de l'instruction doit être dans \$v0, les arguments dans \$a0-\$a3 ;  
le résultat éventuel sera placé dans \$v0

exemple : appel système `print_int` pour afficher un entier

```
li      $v0, 1      # code de print_int
li      $a0, 42     # valeur à afficher
syscall
```

de même `read_int`, `print_string`, etc. (voir la documentation)

Beaucoup de ces instructions sont en fait des **pseudo-instructions** : elles sont traduites par l'assembleur (le programme) en une ou plusieurs instructions de la machine

Exemple : quand on écrit

```
li    $a0, 42
```

l'assembleur le traduit en

```
addiu $a0, $zero, 42
```

Autre exemple :

si l'étiquette `hw` correspond à l'adresse `0x10010020`, l'instruction

```
la    $a0, hw
```

est traduite en

```
lui    $at, 0x1001    # load upper immediate
ori    $a0, $at, 0x0020
```

`$at` est le registre réservé à l'assembleur



C'est de traduire un programme d'un langage de haut niveau vers ce jeu d'instructions

En particulier, il faut

- traduire les structures de contrôle (tests, boucles, exceptions, etc.)
- traduire les appels de fonctions
- traduire les structures de données complexes (tableaux, enregistrements, objets, clôtures, etc.)
- allouer de la mémoire dynamiquement

Une machine fournit

- un jeu limité d'instructions, très primitives
- des registres efficaces, un accès coûteux à la mémoire

La mémoire est découpée en

- code / données statiques / tas (données dynamiques) / pile