

Compilation

Introduction & architecture MIPS

Sandrine Blazy
(d'après le cours de François Pottier)

ensiiē - 2^e année

20 octobre 2008

**école nationale supérieure d'informatique
pour l'industrie et l'entreprise**

ensiiē

1 Introduction

Le processeur MIPS

Programmation du MIPS

Qu'est-ce qu'un compilateur ?

Un compilateur *traduit efficacement* un langage de haut niveau (adapté à l'esprit humain) vers un langage de bas niveau (conçu pour être exécuté efficacement par une machine).

Un exemple de compilation optimisante

```
double dotproduct(int n, double a[], double b[])
{
    double dp = 0.0;
    int i;
    for (i = 0; i < n; i++) dp += a[i] * b[i];
    return dp;
}
```

Compilé pour Alpha et retranscrit à peu près en C ...

```

double dotproduct(int n, double a[], double b[])
{ dp = 0.0;
  if (n <= 0) goto L5;
  r2 = n - 3; f1 = 0.0; r1 = 0; f10 = 0.0; f11 = 0.0;
  if (r2 > n || r2 <= 0) goto L19;
  prefetch(a[16]); prefetch(b[16]);
  if (4 >= r2) goto L14;
  prefetch(a[20]); prefetch(b[20]);
  f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
  r1 = 8; if (8 >= r2) goto L16;
L17:
  f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
  f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
  f12 = a[4]; f16 = f18 * f16;
  f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
  f11 += f17; r1 += 4; f10 += f15;
  f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
  f1 += f16; dp += f19; b += 4;
  if (r1 < r2) goto L17;
L16:
  f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
  f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
  f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
  a += 4; b += 4; f14 = a[8]; f15 = b[8];
  f11 += f22; f1 += f21; dp += f24;
L18:
  f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
  f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
  a += 4; f28 = f29 * f28; b += 4;
  f10 += f14; f11 += f12; f1 += f26;
  dp += f28; dp += f1; dp += f10; dp += f11;
  if (r1 >= n) goto L5;
L19:
  f30 = a[0]; f18 = b[0]; r1 += 1; a += 8; f18 = f30 * f18; b += 8;
  dp += f18;
  if (r1 < n) goto L19;
L5: return dp;
}

```

- C
- Java → JVM
- langages de description de textes (Latex → Postscript)
- langages spécialisés (SQL, langages réactifs, langages de macros)
- processeurs spécialisés (téléphones mobiles, PDA, téléviseurs, robots, ...)

- choix de structures de données (*arbres de syntaxe abstraite*)
- décomposition en passes de compilation (*langages intermédiaires*)
- gestion des erreurs
 - détecter erreurs (statiques vs. dynamiques) et avertissements
 - diagnostic à fournir à l'utilisateur

- aide à l'utilisateur
- efficacité
- correction

- *Conception d'un compilateur* d'un langage source modeste (sous-ensemble de Pascal et C) vers un langage assembleur (SPIM).
- *Chaque cours expose une des phases* du compilateur.

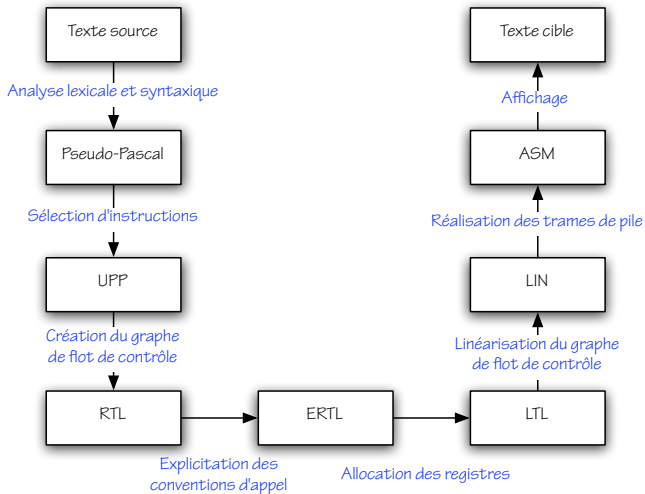
Pourquoi suivre ce cours ?

- pour écrire un programme *complexe* et *élégant* dans un langage de très haut niveau ;
- pour comprendre le *fossé* entre l'intention humaine et le langage de bas niveau exécuté par le microprocesseur ;
- pour découvrir des *techniques* et *algorithmes* d'usage général : transformation d'arbres de syntaxe abstraite, analyse de flot de données, allocation de registres par coloriage de graphe, etc.

Un domaine mature :

- 50 ans ... (Fortran I : 1957)
- Énorme corpus d'algorithmes (optimisations).
- Nombreux compilateurs qui font des choses très rusées.

Un compilateur de Pseudo-Pascal vers MIPS



- 1 Analyse lexicale et *syntaxique* (Lex & Yacc).
- 2 Architecture *MIPS*.
- 3 Syntaxe, sémantique et interprétation de *Pseudo-Pascal*.
- 4 Typage. *Sélection d'instructions* (de PP vers UPP).
- 5 Création du *graphe de flot de contrôle* (de UPP vers RTL).
- 6 Explicitation de la *convention d'appel* (de RTL vers ERTL).
- 7 Analyse de *durée de vie* (sur ERTL).
- 8 *Coloriage de graphe* et allocation des registres (de ERTL vers LTL).
- 9 *Linéarisation* du code (de LTL vers LIN) puis *réalisation des trames* de pile (de LIN vers ASM).

Introduction

2 Le processeur MIPS

Programmation du MIPS

Du point de vue du programmeur ou du compilateur, un ordinateur est constitué principalement d'un *processeur* et d'une *mémoire*.

Le processeur est capable de lire et d'écrire des informations en mémoire. Il dispose également d'un petit nombre d'emplacements mémoire plus rapides, appelés *registres*.

Les *instructions* que doit exécuter le processeur sont elles-mêmes stockées en mémoire. Un registre spécial nommé *pc* contient *l'adresse* de la prochaine instruction à exécuter.

De façon répétée, le processeur lit l'instruction stockée à l'adresse *pc* (*fetch*), l'analyse (*decode*), puis l'interprète (*execute*), ce qui peut avoir pour effet de modifier certains registres (dont *pc*) et/ou la mémoire.

Les principales différences entre processeurs concernent leur jeu d'instructions.

- Les processeurs CISC (*Complex Instruction Set*)
 - Leurs instructions, de taille variable, sont variées et réalisent souvent des transferts avec la mémoire. Ces processeurs possèdent en général peu de registres, dont certains sont réservés pour des usages spécifiques.
 - Exemples : Intel 8086 et Motorola 68000 (pré-1985).
- Les processeurs RISC (*Reduced Instruction Set*)
 - Leurs instructions, de taille fixe, sont régulières et peu d'entre elles lisent ou écrivent en mémoire. Ces processeurs possèdent en général de nombreux registres, lesquels sont uniformes.
 - Exemples : Alpha, Sparc, Mips, PowerPC (post-1985).
- Le Pentium d'Intel mélange les deux designs...

La mémoire est un grand tableau dont les indices sont les adresses.

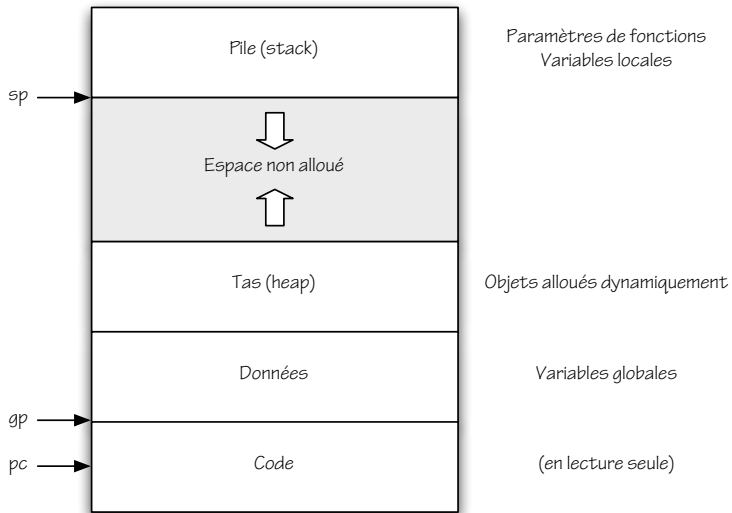
Pour des raisons historiques, les adresses sont mesurées en *octets* (8 bits).

Cependant, les lectures et écritures en mémoire se font sur des quantités de mémoire plus importantes que l'octet. Le *mot* est la taille des données que l'on peut transférer en une instruction : typiquement 32 ou 64 bits, selon les processeurs. Le mot est également la taille des adresses et des registres.

Tous les processeurs modernes contiennent une unité de gestion mémoire (MMU) qui distingue adresses *physiques* et adresses *virtuelles*.

La MMU est utilisée par le système d'exploitation pour donner à chaque processus l'illusion qu'il dispose de toute la mémoire.

La mémoire vue depuis un processus



Les registres du MIPS

Le MIPS comporte 32 registres généraux, nommés $r0$ à $r31$. Le registre $r0$, appelé *zero*, contient toujours la valeur 0, même après une écriture. Les 31 autres registres sont *interchangeables*.

Néanmoins, ces 31 registres peuvent être, par *convention* :

- réservés pour le passage d'arguments ($a0-a3$, ra) ou le renvoi de résultats ($v0-v1$) ;
- considérés comme sauvegardés par l'appelé ($s0-s7$) ou non ($t0-t9$) lors des appels de fonctions ;
- réservés pour contenir des pointeurs vers la pile (sp , fp) ou vers les données (gp) ;
- réservés par le noyau ($k0-k1$) ;
- réservés par l'assembleur (at).

Le MIPS propose trois types principaux d'instructions :

- les instructions de *transfert* entre registres et mémoire ;
- les instructions de *calcul* ;
- les instructions de *saut*.

Seules les premières permettent d'accéder à la mémoire ; les autres opèrent uniquement sur les registres.

- *Lecture* (load word) :

```
lw  dest, offset(base)
```

On ajoute la constante (de 16 bits) *offset* à l'adresse contenue dans le registre *base* pour obtenir une nouvelle adresse ; le mot stocké à cette adresse est alors transféré vers le registre *dest*.

- *Écriture* (store word) :

```
sw source, offset(base)
```

On ajoute la constante (de 16 bits) *offset* à l'adresse contenue dans le registre *base* pour obtenir une nouvelle adresse ; le mot stocké dans le registre *source* est alors transféré vers cette adresse.

Ces instructions lisent la valeur de 0, 1 ou 2 registres dits *arguments*, effectuent un calcul, puis écrivent le résultat dans un registre dit *destination*.

Un même registre peut figurer plusieurs fois parmi les arguments et destination.

- *Écriture d'une constante* (Load Immediate) :

```
li    dest, constant
```

Produit la constante *constant*.

- *Addition d'une constante* (Add Immediate) :

```
addi  dest, source, constant
```

Produit la somme de la constante (de 16 bits) *constant* et du contenu du registre *source*.

- *Déplacement* (Move) :

```
move  dest, source
```

Produit le contenu du registre *source*. Cas particulier de **addi** !

- *Négation* (Negate) :

```
neg   dest, source
```

Produit l'opposé du contenu du registre *source*. Cas particulier de **sub** !

- *Addition* (Add) :

```
add    dest, source1, source2
```

Produit des contenus des registres *source1* et *source2*.

- On a également **sub**, **mul**, **div**.
- *Comparison* (Set On Less Than) :

```
slt    dest, source1, source2
```

Produit 1 si le contenu du registre *source1* est inférieur à celui du registre *source2*; produit 0 sinon.

- On a également **sle**, **sgt**, **sge**, **seq**, **sne**.

On distingue les instructions de saut selon que :

- leurs destinations possibles sont au nombre de 1 (saut *inconditionnel*) ou bien 2 (saut *conditionnel*) ;
- leur adresse de destination est *constante* ou bien *lue* dans un registre ;
- une *adresse de retour* est sauvegardée ou non.

- *Saut* (Jump) :

```
j      address
```

Saute à l'adresse constante *address*. Celle-ci est en général donnée sous forme symbolique par une *étiquette* que l'assembleur traduira en une constante numérique.

- *Saut conditionnel unaire* (Branch on Greater Than Zero) :

```
bgtz source, address
```

Si le contenu du registre *source* est supérieur à zéro, saute à l'adresse constante *address*.

- On a également **bgez**, **blez**, **bltz**.
- *Saut conditionnel binaire* (Branch On Equal) :

```
beq source1, source2, address
```

Si les contenus des registres *source1* et *source2* sont égaux, saute à l'adresse constante *address*.

- On a également **blt** et **bne**.

- *Saut avec retour* (Jump And Link) :

```
jal    address
```

Sauvegarde l'adresse de l'instruction suivante dans le registre *ra*, puis saute à l'adresse constante *address*.

- *Saut vers adresse variable* (Jump Register) :

```
jr    target
```

Saute à l'adresse contenue dans le registre *target*.

L'instruction **jr** *\$ra* est typiquement employée pour rendre la main à l'appelant à la fin d'une fonction ou procédure.

- *Appel système* (System Call) :

syscall

Provoque un appel au noyau. Par convention, la nature du service souhaité est spécifiée par un code entier stocké dans le registre *v0*. Des arguments supplémentaires peuvent être passés dans les registres *a0-a3*. Un résultat peut être renvoyé dans le registre *v0*.

SPIM propose un très petit nombre d'appels système : voir la page A-44 de sa [documentation](#).

Le *langage assembleur* attribue des noms symboliques aux instructions, aux registres, et aux adresses constantes.

Le *langage machine* est une suite de mots, qui peuvent représenter des données ou des instructions.

L'assembleur (pour nous, SPIM) est un programme qui traduit le langage assembleur en langage machine.

Instructions et pseudo-instructions

Certaines des instructions précédentes ne sont en fait pas implantées par le processeur, mais *traduites* par l'assembleur en séquences d'instructions plus simples.

Par exemple,

```
blt $t0, $t1, address
```

est expansée en

```
slt $at, $t0, $t1  
bne $at, $zero, address
```

Ou encore,

```
li $t0, 400020
```

est expansée en

```
lui $at, 6  
ori $t0, $at, 6804
```

Pour nous, la distinction entre instructions et pseudo-instructions n'aura pas d'importance.

Exemple de programme assembleur

```
.data           # Des données suivent.
hello:          # L'adresse de la donnée qui suit.
.ascii "hello_world\n" # Chaîne terminée par un zéro.
.text         # Des instructions suivent.
main:          # Adresse de l'instruction qui suit.
li $v0, 4     # Code de l'appel print_string.
la $a0, hello # Adresse de la chaîne.
syscall       # Appel au noyau.
jr $ra       # Fin de la procédure main.
```

Introduction

Le processeur MIPS

3 Programmation du MIPS

Des constructions qui peuvent sembler élémentaires dans un langage tel que Pascal (*tests, boucles, fonctions*) seront traduites par de nombreuses instructions assembleur.

Par ailleurs, des notions d'apparence pourtant simple n'ont pas d'équivalent. Par exemple, les *variables* de Pascal seront traduites en termes d'emplacements mémoire et de registres – une tâche non triviale.

Le fragment de C suivant :

```
if (t1 < t2) t3 = t1; else t3 = t2;
```

peut être traduit en assembleur MIPS comme ceci :

```
blt $t1, $t2, then      # si t1 < t2 saut vers l'étiquette then  
move $t3, $t2          # t3 := t2  
j    done              # saut vers l'étiquette done  
then:  
move $t3, $t1          # t3 := t1  
done:                   # suite du programme
```


Le fragment de C suivant :

```
t2 = 0;
while (t1 > 0) { t2 = t2 + t1; t1 = t1 - 1; }
```

peut être traduit en assembleur MIPS comme ceci :

```
li    $t2, 0           # t2 := 0
while:           # début de la boucle
blez $t1, finboucle  # si t1 <= 0 saut vers l'étiquette done
add  $t2, $t2, $t1    # t2 := t2 + t1
sub  $t1, $t1, 1      # t1 := t1 - 1
j    while           # retour vers l'étiquette while
done:           # suite du programme
```

La fonction C suivante :

```
int succ (int x)
{
    return x + 1;
}
```

peut être traduite en assembleur MIPS comme ceci :

```
succ:                # $a0 contient l'argument x
addi $v0, $a0, 1    # $v0 contient le résultat
jr   $ra           # retour à l'appelant
```

Le choix des registres est imposé par la *convention d'appel*.

La convention d'appel régit la *communication* entre appelant et appelé lors de l'appel de procédure ou de fonction. Pour le MIPS, la convention *proposée* par le fabricant est :

- les arguments sont passés dans *a0-a3* puis (s'il y en a plus de quatre) sur la pile ;
- l'adresse de retour est passée dans *ra* ;
- la valeur de retour est renvoyée dans *v0* ;
- les registres mentionnés ci-dessus, ainsi que *t0-t9*, *peuvent être modifiés* par l'appelé ; l'appelant doit donc les sauvegarder si nécessaire ; on les appelle *caller-save* ;
- les registres *s0-s7* *doivent être préservés* par l'appelé ; on les appelle *callee-save*.

Plus difficile : voici une fonction C récursive.

```
int f (int n)
{
    if (n <= 0)
        return 1;
    else
        return n * f (n - 1);
}
```

Voici une traduction naïve et *incorrecte* en assembleur MIPS :

```
fact:
blez $a0, fact0      # si a0 ≤ 0 saut vers l'étiquette fact0
sub  $a0, $a0, 1     # a0 := a0 - 1
jal  fact            # appel récursif; lit a0 et écrit dans v0
mul  $v0, $a0, $v0   # v0 := a0 * v0
jr   $ra             # retour à l'appelant
fact0:
li   $v0, 1          # v0 := 1
jr   $ra             # retour à l'appelant
```

Quelle est l'erreur ?

Parce que *fact* est récursive, plusieurs appels imbriqués peuvent être actifs simultanément :

$$\text{fact} \left\{ \begin{array}{l} a_0 \leftarrow a_0 - 1 \\ \text{fact} \left\{ \begin{array}{l} a_0 \leftarrow a_0 - 1 \\ \dots \\ v_0 \leftarrow a_0 \times v_0 \end{array} \right. \\ v_0 \leftarrow a_0 \times v_0 \end{array} \right.$$

L'appel récursif *modifie* a_0 – en fait, il en détruit le contenu. Or, celui-ci est utilisé *après* l'appel.

Il faut donc *sauvegarder* le contenu de a_0 avant l'appel, puis le *restaurer* après l'appel. (Et de même pour *ra*.)

Par convention, la pile grandit dans le sens des adresses décroissantes. Le registre *sp* pointe vers le *sommet* de la pile, c'est-à-dire vers le dernier mot alloué.

On *sauvegarde* le contenu de *a0* sur la pile comme ceci :

```
sub $sp, $sp, 4      # alloue un mot sur la pile  
sw  $a0, 0($sp)    # copie $a0 vers le sommet de la pile
```

On *restaure* le contenu de *a0* depuis la pile comme cela :

```
lw  $a0, 0($sp)    # copie le sommet de la pile vers $a0  
add $sp, $sp, 4     # désalloue un mot sur la pile
```

Voici le code produit par une version de notre compilateur :

```
f11:                                f16:
addiu $sp, $sp, -8                lw    $ra, 4($sp)
sw    $ra, 4($sp)                 lw    $s0, 0($sp)
sw    $s0, 0($sp)                 addiu $sp, $sp, 8
move  $s0, $a0                    jr    $ra
blez  $s0, f4                     f4:
addiu $a0, $s0, -1                li    $v0, 1
jal   f11                          j     f16
mul   $v0, $s0, $v0
```

Exercice recommandé : expliquez chaque instruction.