Code Machine
Code Assembleur
Processeurs RISC
(Byte code)

Didier Rémy
Octobre 2000

http://cristal.inria.fr/~remy/poly/compil/1/
http://w3.edu.polytechnique.fr/profs/informatique//Didier.Remy/compil/1/

Informations utiles

- La pico-machine décrite dans le livre *Le langage Caml* de Pierre Weis et Xavier Leroy.
- Le simulateur SPIM du processeur MIPS R2000 est présenté succinctement ci-dessous. Voir aussi son manuel de référence en ligne et en Postscript.

Les processeurs

Les processeurs se ressemblent tous du point de vue de l'utilisateur (*i.e.* de la partie visible de leur architecture). Ils comportent principalement une mémoire, un ensemble de registres, et un jeu d'instructions.

Les différences proviennent surtout du jeu d'instructions :

- Les (vieux) processeurs CISC (*Complex Instruction Set*)
  - Leurs instructions sont de taille variable et beaucoup réalisent des transferts avec la mémoire ; ils possèdent en général peu de registres (et pas uniformes)
  - Conçus avant 1985. Typiquement : Intel 8086 et Motorola 68000.
- Les (nouveaux) processeurs RISC (*Reduced Instruction Set*)
  - Instructions de taille fixe, régulières (*trois adresses*) dont peu font des transfert avec la mémoire. Ils possèdent en général beaucoup de registres (uniformes).
  - Conçus après 1985. Typiquement : Alpha, Sparc et Mips. G4.

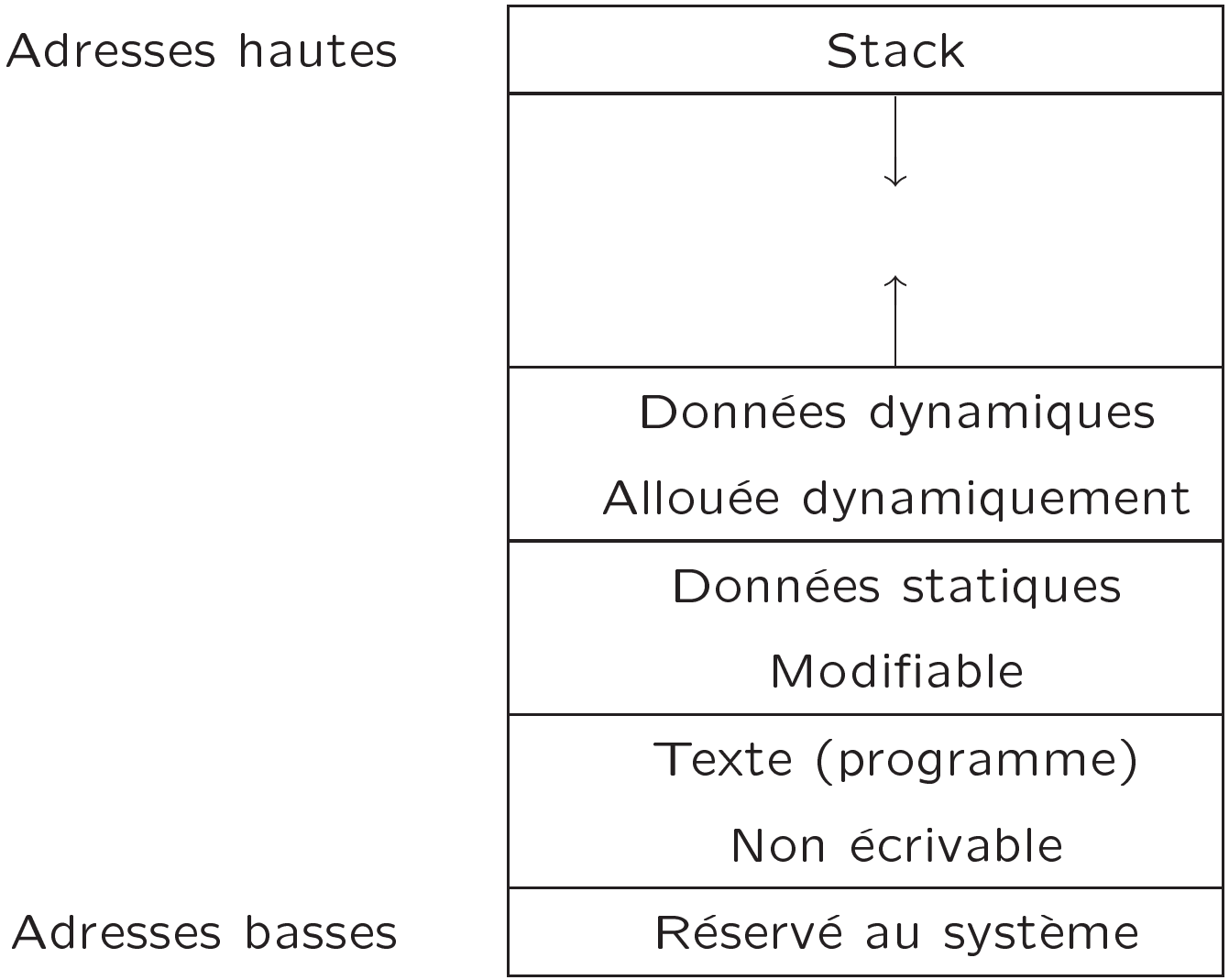
La mémoire

Tous les processeurs modernes comportent une unité mémoire (MMU) qui permet de manipuler des adresses virtuelles, *i.e.* de faire un renommage, transparent pour l'utilisateur, entre les adresses virtuelles du programme et les adresses réelles en mémoire.

Cela permet à chaque programme de choisir ses adresses indépendemment des autres programmes (qui peuvent être exécutés en même temps sur la même machine avec les mêmes adresses virtuelles mais des adresses réelles différentes).

Du point de vue de l'utilisateur, la mémoire est un (grand) tableau dont les indices sont les adresses.

Les zones mémoire



Les registres du MIPS

Le MIPS comporte 32 registres généraux interchangeables, sauf

- le registre `zero` qui vaut toujours `0`, même après une écriture.
- Le registre `ra`, utilisé implicitement par certaines instructions pour sauver l'adresse de retour avant un saut.

Les autres registres ont des utilisations préférentielles, mais cela n'est strict que pour communiquer avec d'autres programmes (par exemple fournir ou utiliser des programmes en librairies) :

- passage (`a0`, ... `a3`) et retour (`v0`, `v1`) des arguments ;
- registres temporaires sauvegardés (`s0`, .. `s7`) ou non (`t0`, ... `t9`) par les appels de fonction.
- pointeurs de pile `sp`, `fp` ou de donnée `gp` ;
- réservés par l'assembleur `at` et le système `k0`, `k1`.

Le jeu d'instructions

n	une constante entière
ℓ	une étiquette (adresse)

r	nom de registre
a	absolu (n ou ℓ)
o	opérande (r ou a)

La plupart des instructions suivent le modèle

– **add** r_1, r_2, o qui place dans r_1 la valeur $r_2 + o$.

Les instructions qui interagissent avec la mémoire sont uniquement les instructions `load` et `store`.

– **lw** r_1, $n(r_2)$ place dans r_1 le mot contenu à l'adresse $r_2 + n$.

– **sw** r_1, $n(r_2)$ place r_1 dans le mot contenu à l'adresse $r_2 + n$.

Les instructions de contrôle conditionnel ou inconditionnel :

– **bne** r, a, ℓ saute à l'adresse ℓ si r et a sont différents,

– **jal** o qui sauve $pc + 1$ dans `ra` et saute à l'étiquette o.

Syntaxe	Effet
move r_1, r_2	$r_1 \leftarrow r_2$
add r_1, r_2, o	$r_1 \leftarrow o + r_2$
sub r_1, r_2, o	$r_1 \leftarrow r_2 - o$
mul r_1, r_2, o	$r_1 \leftarrow r_2 \times o$
div r_1, r_2, o	$r_1 \leftarrow r_2 \div o$
and r_1, r_2, o	$r_1 \leftarrow r_2$ land o
or r_1, r_2, o	$r_1 \leftarrow r_2$ lor o
xor r_1, r_2, o	$r_1 \leftarrow r_2$ lxor o
sll r_1, r_2, o	$r_1 \leftarrow r_2$ lsl o
srl r_1, r_2, o	$r_1 \leftarrow r_2$ lsr o
li r_1, n	$r_1 \leftarrow n$
la r_1, a	$r_1 \leftarrow a$

Syntaxe	Effet
lw $r_1, o(r_2)$	$r_1 \leftarrow tas.(r_2 + o)$
sw $r_1, o(r_2)$	$r_1 \rightarrow tas.(r_2 + o)$
slt r_1, r_2, o	$r_1 \leftarrow r_2 < o$
sle r_1, r_2, o	$r_1 \leftarrow r_2 \leq o$
seq r_1, r_2, o	$r_1 \leftarrow r_2 = o$
sne r_1, r_2, o	$r_1 \leftarrow r_2 \neq o$
j o	$pc \leftarrow o$
jal o	$ra \leftarrow pc + 1 \wedge pc \leftarrow o$
beq r, o, a	$pc \leftarrow a$ si $r = o$
bne r, o, a	$pc \leftarrow a$ si $r \neq o$
syscall	appel système
nop	ne fait rien

Les appels systèmes

Ils permettent l'interaction avec le système d'exploitation, et en dépendent. Le numéro de l'appel système est lu dans `v0` (attention, ce n'est pas la convention standard). Selon l'appel, un argument supplémentaire peut être passé dans `a0`.

Le simulateur SPIM implémente les appels suivants :

Nom	N^o	Effet
`print_int`	1	imprime l'entier contenu dans `a0`
`print_string`	4	imprime la chaîne en `a0` jusqu'à '\000'
`read_int`	5	lit un entier et le place dans `v0`
sbrk	9	alloue `a0` bytes dans le tas, retourne l'adresse du début dans `v0`.
exit	10	arrêt du programme en cours d'exécution

Le jeu d'appel système dépend du système d'exploitation.

Langage d'assembleur et langage machine

Le langage d'assembleur est un langage symbolique qui donne des noms aux instructions (plus lisibles que des suites de bits). Il permet aussi l'utilisation d'étiquettes symboliques et de pseudo-instructions et de modes d'adressage surchargés.

Le langage machine est une suite d'instructions codées sur des mots (de 32 bits pour le MIPS). Les instructions de l'assembleur sont expansées en instructions de la machine à l'édition de lien. Les étiquettes sont donc résolues et les pseudo-instructions remplacées par une ou plusieurs instructions machine.

L'assemblage est la traduction du langage d'assembleur en langage machine. Le résultat est un fichier object qui contient, en plus du code, des informations de relocation qui permettent de lier *(linker)* le code de plusieurs programmes ensemble.

Pseudo-instructions

La décompilation du langage machine en langage assembleur est facile. Elle permet de présenter les instructions machine (mots de 32 bits) sous une forme plus lisible.

Exemple d'expansion (présentée sous une forme décompilée).

Assembleur	Langage machine	Commentaire
blt r, o, a	**slt** \$1, r, o **bne** \$1, \$0, a	Justifie le registre `at` (\$1) réservé par l'assembleur.
li \$t0, 400020	**lui** \$1, 6 **ori** \$8, \$1, 6804	charge les 16 bits de poids fort puis les 16 bits de poids faible
add \$t0, \$t1, 1	**addi** \$8, \$9, 1	addition avec une constante
move \$t0, \$t1	**addu** \$8, \$0, \$9	addition "unsigned" avec zéro

La pico-machine

C'est une machine inventée (Pierre Weis et Xavier Leroy) pour des raisons pédagogiques.

C'est un modèle simplifié de machine RISC, très proche du MIPS.

Son jeu d'instructions est (pas tout à fait) un sous-ensemble du MIPS.

Une grosse différence de syntaxe... Dans le code 3 adresses, source et destination sont inversés :

Code MIPS	Pico code	signification
`nom` r_1, o, r_2	`nom` r_2, o, r_1	$r_1 \leftarrow o$ `nom` r_2

Exemple de programme assembleur

hello.spi

```
        .data
hello : .asciiz  "hello\n"    # hello pointe vers "hello\n\0"
        .text
        .globl  __start
__start :
        li      $v0, 4        # la primitive  print_string
        la      $a0, hello    # a0 l'adresse de hello
        syscall
```

Le programme est assemblé, chargé puis exécuté par :

```
spim -notrap -file hello.spi
```

Par convention le programme commence à l'étiquette __start.

Si on retire l'option -notrap, le chargeur ajoute un prélude qui se branche à l'étiquette main (remplacer alors __start par main).

if-then-else

On utilise des sauts conditionnels et inconditionnels :

Pascal la fonction minimun

```
if t1 < t2 then t3 := t1 else t3 := t2
```

Assembleur Mips

```
        blt   $t1, $t2, Then    # si t1 >= t2 saut à Then
        move  $t3, $t2          # t3 := t1
        j     End               # saut à Fi
Then:   move  $t3, $t1          # t3 := t2
End:                            # suite du programme
```

Boucles

Pascal : calcule dans t2 = 0 la somme des entiers de 1 à t1

```
while t1 > 0 do begin t2 := t2 + t1; t1 := t1 -1 end
```

Programme équivalent

```
While:
    if  t1 <= 0 then goto End
    else
      begin
        t2 := t2 + t1;
        t1 := t1 - 1;
        goto While
      end;
End:
```

Code Mips

```
While:
    ble   $t1, $0, End


    add   $t2, $t2, $t1
    sub   $t1, $t1, 1
    j     While

End:
```

Appel de procédure

Pour appeler une procédure, il faut sauver l'adresse de retour à l'appel, en général dans $ra, pour revenir après l'appel.

À la fin d'une procédure, on retourne à l'appelant en sautant à l'adresse contenu dans $ra.

Par exemple on définit une procédure writeln qui imprime un entier puis un retour à la ligne.

```
writeln :                   # l'argument est dans a0
        li    $v0, 1        # le numéro de print_int
        syscall             # appel système
        li    $v0, 4        # la primitive  print_string
        la    $a0, nl       # la chaîne "\n"
        syscall
        j     $ra           # saut à l'adresse ra
```

Le programme complet

```
        .data                   # le tas
nl:     .asciiz   "\n"          # la chaîne "\n"
        .text                   # la zône code
        .globl  __start
__start :
        li      $a0, 1          # a0 <- 1
        jal     writeln         # ra <- pc+1; saut à writeln
        li      $a0, 2          # on recommence avec 2
        jal     writeln
        j       Exit            # saut à la fin du programme
writeln :
        ...
Exit:                           # fin du programme
```

Noter la différence entre les instructions `j` et **jal**.

Procédures récursives

Lorsqu'une procédure est récursive plusieurs appels imbriqués peuvent être actifs simultanément :

$$
f_n \begin{cases} a_0 \leftarrow v_0 - 1 \\ f_{n-1} \begin{cases} a_0 \leftarrow v_0 - 1 \\ \ldots \\ v_0 \leftarrow v_0 \times a_0 \end{cases} \\ v_0 \leftarrow v_0 \times a_0 \end{cases}
$$

C'est bien sûr le même code qui est utilisé pour f_n et f_{n-1}. Il faut donc sauver avant l'appel récursif la valeur des registres qui seront (ou risque d'être) modifiés par l'appel récursif et dont les valeurs seront encore nécessaires (lues) au retour de l'appel.

On utilise la pile.

La pile

Par convention, la pile grossit vers les adresses décroissantes. Le registre sp pointe vers le dernier mot utilisé.

Pour sauver un registre r sur la pile

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

Pour restaurer un mot de la pile dans un registre r

```
lw   r, 0($sp)       # lit le sommet de la pile dans r
add  $sp, $sp, 4     # désalloue un mot sur la pile
```

En général, on alloue et désalloue l'espace en pile par blocs pour plusieurs temporaires à la fois.

On peut aussi utiliser la pile pour allouer des tableaux si leur durée de vie le permet.

Conventions d'appel

En général :

- les arguments sont passés dans les registres `a0` à `a3` puis dans la pile.
- la ou les valeurs de retour dans `v0` et `v1`.

De même :

- les registres t_i sont sauvés par l'appelant (l'appelé peut les écraser)
- les registres s_i sont sauvés par l'appelé (qui doit les remettre dans l'état où il les a pris)

Mais ce n'est qu'une convention ! (celle proposé par le fabriquant)

La respecter permet de comminiquer avec d'autres programmes (qui la respectent également).

Choisir une autre convention est possible tant que l'on n'intéragit pas avec le monde extérieur.

Exemple : calcul de la factorielle

L'argument est dans `a0`, le résultat dans `v0`.

```
fact:    blez $a0, fact_0       # si a0 <= 0 saut à fact_0
         sub  $sp, $sp, 8       # réserve deux mots en pile
         sw   $ra, 0( $sp)      # sauve l'adresse de retour
         sw   $a0, 4($sp)       # et la valeur de a0
         sub  $a0, $a0, 1       # décrément a0
         jal  fact              # v0 <- appel récursif (a0-1)
         lw   $a0, 4($sp)       # récupére a0
         mul  $v0, $v0, $a0     # v0 <- a0 * v0
         lw   $ra, 0( $sp)      # récupére l'adresse de retour
         add  $sp, $sp, 8       # libère la pile
         j    $ra               # retour à l'appelant

fact_0:  li   $v0, 1            # v0 <- 1
         j    $ra               # retour à l'appelant
```

Allocation dans le tas

Allocation statique Un espace est réservé au chargement.

```
Tableau:                    # adresse symbolique sur le début
        .align  2           # aligner sur un mot (2^2 bytes)
        .space 4000         # taille en bytes
```

Allocation dynamique En cours de calcul on peut demander au système d'agrandir le tas.

En SPIM, l'appel système numéro 9 prend la taille dans `v0` et retourne le pointeur vers le début de bloc dans `a0`.

```
malloc:                     # procédure d'allocation dynamique
        li   $v0, 9         # appel système n. 9
        syscall             # alloue une taille a0 et
        j    $ra            # retourne le pointeur dans v0
```

Gestion de l'allocation

En pratique, on alloue un gros tableau `Tas` (statiquement ou dynamiquement) dans lequel on s'alloue des petits blocs.

Par exemple on réserve un registre pour pointer sur le premier emplacement libre du tas.

```
__start :
        la   $t8, Tas        # on réserve le registre t8
        ...
array :
        sw   $a0, ($t8)      # écrit la taille dans l'entête
        add  $v0, $t8, 4     # v0 <- adresse de la case 0
        add  $t8, $v0, $a0   # augmente t8 de la taille +1
        j    $ra
```

(On ignore ici les problèmes de débordement —qu'il conviendrait de traiter.)

Le byte-code

C'est un jeu d'intructions inventé pour une machine virtuelle (ou abstraite) qui est alors interprété par un programme, lui même compilé dans l'assembleur de la machine réelle.

Avantages : le byte-code est indépendant de l'architecture, il est portable (fonctionne sur différentes machines).

Le jeu d'instructions peut être mieux adpapté au langage compilé.

Inconvénient : l'interprétation du byte-code ajoute au moins un facteur 5 en temps d'exécution (pour de très bons byte-codes).

Exemples : Java, Ocaml.

Micro-code : Dans une machine réelle, les instructions sont en fait interprétées par du micro-code (qui lui est vraiment câblé).

Exercices

Utilisation de SPIM en td Spim est installé sur les machines de la salle 33. Dans les autres salles, il faut éventuellement se loger à distance sur des machines de cette salle linux. (Voir les détails dans le version HTML.)

La fonction de Fibonnaci

Écrire la fonction fib en assembleur MIPS. On pourra commencer par une version simple, inefficace.

Écrire également une version optimisée qui mémorise les deux appels précédents dans des registres. On pourra lire l'entier au clavier et l'imprimer à l'écran.

Compilation des expressions arithmétiques

On se propose de compiler les expressions arithmétiques en langage MIPS. On se limitera à des expressions avec les 4 opérations de bases, les constantes entières et une seule variable (qui sera lue au début du programme dans le flux d'entrée).

Pour simplifier, une expression arithmétique sera donnée par sa syntaxe abstraite, et on prendra la définition de type suivante (**imposée**) :

expression.mli

```
type expression =
  | Binexp of binop * expression * expression
  | Int of int
  | X (* variable *)
and binop = Plus | Minus | Times | Div;;
```

Pour éviter les problèmes d'entrée sortie, on écrira le programme à compiler sous la forme d'un fichier source. Par exemple, pour

compiler l'expression `((x * x) + (1 - x * 2))`, on écrira le source dans un fichier

arith.ml

```
open Expression;;
open Arithmetique;;
let  e =
   Binexp (Plus,
           Binexp (Times, X, X),
           Binexp (Minus,
                   Int  1,
                   Binexp (Times, X, Int 2)))
in  compile e;;
```

On fabriquera un exécutable `arith.out` par la commande :

```
ocamlc -o arith.out expression.mli  arithmetique.ml arith.ml
```

puis le fichier de code Spim compilé en lançant cette commande

```
./arith.out > arith.spi
```

Le programme compilé `arith.spi` devra être exécutable sur la machine SPIM par la commande :

```
spim -notrap -file arith.spi
```

standard, évalue l'expression, et ainsi de suite tant que l'entier lu est non nul.

L'exercice consiste donc a écire un fichier source `arithmetique.ml` qui implémente la fonction `compile : epxressions -> unit` qui imprime dans la sortie standard le code spim correspondant à l'évaluattion de l'expression reçue en argument.

Dans un premier temps, on ne recherchera pas l'efficacité, mais la simplicité. Par exemple :

- on effectue toutes les opérations dans les registres a_1 et a_2.
- on utilise le registre a_3 pour la valeur de la variable, et
- on sauve les arguments dans la pile lors d'un calcul auxiliaire.
- Commencer par une version de la fonction compile qui ne traire que les cas simples, *i.e.* où l'expression arithmétique et une

constante, une variable, ou ne contient qu'une seule opération.

Extensions

- Cummuler les résultats auxilliaires, et afficher la somme à la fin.
- utiliser les registres au mieux et la pile seulement si nécessaire.