Le processeur MIPS 2000 et SPIM

Il y a plusieurs microprocesseurs, classés en deux grandes classes

CISC : Intel

RISC : MIPS, SPARC, etc.

Pour ce cours, on utilisera comme machine cible le processeur MIPS 2000, un RISC pour lequel on dispose d'un excellent émulateur, SPIM.

Architecture du processeur MIPS 2000

L'architecture des processeurs MIPS est simple et regulière, ce qui en facilite la comprehension et l'apprentissage. On peut resumer en quelques points les choix architecturaux essentiels:

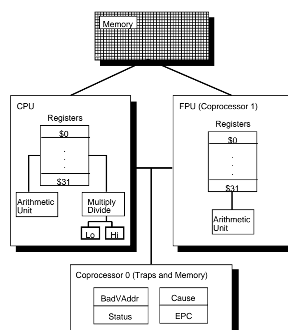
registres le MIPS compte 32 registres génériques[1] de 32 bits

taille du mot une `word` fait 4 octets, i.e. 32 bits (la taille d'un registre); l'accès à la memoire est en général `aligné` à la word (donc, les adresses sont des multiples de 4[2])

load/store l'accès à la memoire se fait *seulement* par des instructions explicites de chargement et mémorisation à partir de registres

arithmétique les opérations arithmétiques utilisent deux registres en entrée et un registre en sortie

format les instructions tiennent toutes sur 32 bits (une word)



Exercice: comparez avec ce que vous savez de la famille x86.

Registres, et conventions

Les 32 registres sont tous équivalents, *mais* pour garantir l'intéropérabilité entre programmes assembleurs produits par compilateurs différents, on a fixé des *conventions d'usage* qui sont détaillées comme suit.

[1] Toutes les opérationssont disponiblessur tous les registres!

[2] comment sontles 2 derniers bitsd'une adresse alignée?

Registre	Numéro	Usage
zero	0	Constante 0
at	1	Reservé pour l'assembleur
v0	2	Evaluation d'expressions et
v1	3	resultats d'une fonction
a0	4	Argument 1
a1	5	Argument 2
a2	6	Argument 3
a3	7	Argument 4
t0	8	Temporary (not preserved across call)
t1	9	Temporaire (non preservé lors d'un appel)
t2	10	Temporaire (non preservé lors d'un appel)
t3	11	Temporaire (non preservé lors d'un appel)
t4	12	Temporaire (non preservé lors d'un appel)
t5	13	Temporaire (non preservé lors d'un appel)
t6	14	Temporaire (non preservé lors d'un appel)
t7	15	Temporaire (non preservé lors d'un appel)
s0	16	Temporaire sauvegardé (preservé lors d'un appel)
s1	17	Temporaire sauvegardé (preservé lors d'un appel)
s2	18	Temporaire sauvegardé (preservé lors d'un appel)
s3	19	Temporaire sauvegardé (preservé lors d'un appel)
s4	20	Temporaire sauvegardé (preservé lors d'un appel)
s5	21	Temporaire sauvegardé (preservé lors d'un appel)
s6	22	Temporaire sauvegardé (preservé lors d'un appel)
s7	23	Temporaire sauvegardé (preservé lors d'un appel)
t8	24	Temporaire (non preservé lors d'un appel)
t9	25	Temporaire (non preservé lors d'un appel)
k0	26	Reservé pour OS kernel
k1	27	Reservé pour OS kernel
gp	28	Pointeur à la zone globale
sp	29	Pointeur de pile
fp	30	Pointeur de bloc
ra	31	Adresse de retour (pour les fonctions)

Table 1: Registres MIPS et conventions d'usage.

Les registres `$a0...$a3` sont utilisés pour passer les premiers 4 paramètres d'une fonction lors d'un appel.
Les temporaires "sauvegardés" doivent être préservés par la fonction appellante, les temporaires "non sauvegardés" doivent être préservés par la fonction appelée.
Les registres \$sp, \$fp, \$gp sont utilisés pour compiler des langages comme celui du projet, alors que \$ra contient l'adresse de retour après l'appel à une fonction.
Le registre `zero` contient la constante 0.

Machine virtuelle assembleur

Pour des soucis d'efficacité, la machine MIPS réelle dispose d'instructions *retardées*, permettant d'accélerer l'éxécution des sauts non locaux, et impose des restrictions sur les modes d'adressage, qui compliquent la compréhension du système.

L'assembleur MIPS offre à l'utilisateur une interface qui cache la complexité de l'architecture matérielle réélle.

Dans ces notes, nous nous intéressons exclusivement à la *machine virtuelle* assembleur.

Machine virtuelle assembleur: syntaxe

Commentaires tout ce qui suite # est ignoré

```
# ceci est un commentaire
```

Identificateurs séquence alphanumérique ne commençant pas par un entier; les symboles _ et `.` sont admis; les noms des instructions assembleur sont résérvés

```
j end_loop.2
```

Etiquettes identificateurs qui se trouvent au debut d'une ligne et sont suivis de `:`

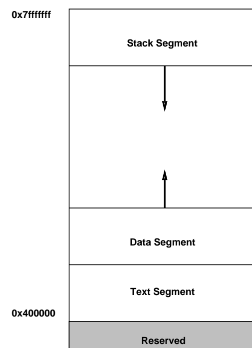
```
v:            .word 33
end_loop.2:   lw $a0, v
```

Chaînes de caractères sont délimitées par `"`. Les caractères spéciaux suivent les conventions habituelles:

```
fin de ligne        \n
tabulation          \t
guillemets          \"
```

Machine virtuelle assembleur: mémoire et directives

La mémoire sur l'émulateur SPIM est divisée en zones, comme suit:



Pour le projet, il nous suffit de connaître les directives suivantes (pour les autres, voir le document de référence sur la page du projet):

`.ascii str`
Met la chaîne en mémoire, sans fin de ligne

`.asciiz str`
Met la chaîne en mémoire, avec fin de ligne

`.data <addr>` Ce qui suit doit aller dans le segment DATA (eventuellement à partir de l'adresse `addr`)

`.text <addr>` Ce qui suit doit aller dans le segment TEXT (eventuellement à partir de l'adresse `addr`)

`.word w1, ..., wn`
Met les n valeurs sur 32-bit dans des mots successifs

Machine virtuelle assembleur: Byte Order

Une `word` occupe plusieurs octets consecutifs:
si on a le mot 00000000000000000000010000000001 (l'entier 1025), et on dispose pour le mémoriser des 4 octets consécutifs aux adresses 10,11,12, et 13, on peut le mémoriser comme ça

Big endian[3] (SPARC,...)

addr	contenu
10	00000001
11	00000100
12	00000000
13	00000000

ou comme

Little endian (PC,...)

addr	contenu
10	00000000
11	00000000
12	00000100
13	00000001

SPIM suit la convention de la machine hôte!

Modes d'adressage

La machine virtuelle assembleur fournit les modes d'adressage suivants pour acceder à la memoire:

Format	**Adresse**
(register)	contenu du registre
imm	l'adresse est imm
imm (register)	imm + contenu du registre
symbol	adresse du symbole
symbol ± imm	adresse du symbole ± immediate
symbol ± imm (register)	adresse du symbole ± (imm + contenu du registre)

N.B.: la plupart des accès mémoire sont *alignés*

Instructions

- Accès à la mémoire
- Arithmétique
- Logique
- Comparaison
- Contrôle

Instruction d'accès à la mémoire (load/store)

`la Rdest, adresse` *Load Address* †

Charge l'*adresse*, (et non le contenu de la case mémoire correspondante), dans le registre `Rdest`.

```
.data
start:  .word 0,0,0,0,0,0,0,0,0,0,1
.text
        la $t0, start+28
```

`lw Rdest, adresse` *Load Word*

Charge l'entier 32-bit (word) qui se trouve à l'*adresse* dans le registre `Rdest`.

```
lw $t0, 32($sp)
lw $t1, start+2($a0)
```

`li Rdest, imm` *Load Immediate* †

Charge la valeur sur 16 bits `imm` dans le registre `Rdest`.

```
li $t0, 32
```

`sw Rsrc, adresse` *Store Word*

Mémorise la word contenue dans le registre `Rsrc` à l'*adresse*.

```
sw $t0, 32($sp)
```

Copie de registres

`move Rdest, Rsrc` *Move* †

Copie le contenu de `Rsrc` dans `Rdest`.

```
move $t0, $a1
```

Opérations arithmétiques

Dans ce qui suit, `Src2` peut-être un registre ou une valeur immédiate sur 16 bits.

`add Rdest, Rsrc1, Src2` *Addition (with overflow)*
`addi Rdest, Rsrc1, Imm` *Addition Immediate (with overflow)*
`addu Rdest, Rsrc1, Src2` *Addition (without overflow)*
`addiu Rdest, Rsrc1, Imm` *Addition Immediate (without overflow)*

Somme registre `Rsrc1` et `Src2` (ou `Imm`) dans le registre `Rdest`.

`div Rdest, Rsrc1, Src2` *Divide (signed, with overflow)* †

Met dans le registre `Rdest` le quotient de la division de `Rsrc1` par `Src2` dans `Rdest`.

`mul Rdest, Rsrc1, Src2` *Multiply (without overflow)* †
`mulo Rdest, Rsrc1, Src2` *Multiply (with overflow)* †

`mulou Rdest, Rsrc1, Src2` *Unsigned Multiply (with overflow)* †

Met le produit de `Rsrc1` et `Src2` dans le registre `Rdest`.

`rem Rdest, Rsrc1, Src2` *Remainder* †
`remu Rdest, Rsrc1, Src2` *Unsigned Remainder* †

Met le reste de la division de `Rsrc1` par `Src2` dans `Rdest`.

`sub Rdest, Rsrc1, Src2` *Subtract (with overflow)*
`subu Rdest, Rsrc1, Src2` *Subtract (without overflow)*

Met la différence entre `Rsrc1` et `Src2` dans `Rdest`.

Opérations logiques

and

and Rdest, Rsrc1, Src2 — *AND*
andi Rdest, Rsrc1, Imm — *AND Immediate*
AND logique de `Rsrc1` et `Src2` (ou `Imm`) dans le registre `Rdest`.
`not Rdest, Rsrc` — *NOT* †
Met la negation logique de `Rsrc` dans `Rdest`.
`or Rdest, Rsrc1, Src2` — *OR*
`ori Rdest, Rsrc1, Imm` — *OR Immediate*
Met le OU logique de `Rsrc1` et `Src2` (ou `Imm`) dans `Rdest`.
`xor Rdest, Rsrc1, Src2` — *XOR*
`xori Rdest, Rsrc1, Imm` — *XOR Immediate*
Met le XOR de `Rsrc1` et `Src2` (ou `Imm`) dans `Rdest`.

`rol Rdest, Rsrc1, Src2` — *Rotate Left* †
`ror Rdest, Rsrc1, Src2` — *Rotate Right* †
Rotation du contenu de `Rsrc1` à gauche (droite) du nombre de places indiqué par `Src2`; le résultat va dans le registre `Rdest`.

`sll Rdest, Rsrc1, Src2` — *Shift Left Logical*
`sllv Rdest, Rsrc1, Rsrc2` — *Shift Left Logical Variable*
`sra Rdest, Rsrc1, Src2` — *Shift Right Arithmetic*[4]
`srav Rdest, Rsrc1, Rsrc2` — *Shift Right Arithmetic Variable*
`srl Rdest, Rsrc1, Src2` — *Shift Right Logical*
`srlv Rdest, Rsrc1, Rsrc2` — *Shift Right Logical Variable*
Décale `Rsrc1` à gauche (droite) du nombre de places indiqué par `Src2` (`Rsrc2`) et met le résultat dans `Rdest`.

Instructions de comparaison

(`Src2` est un registre, ou une valeur 32 bits)

`seq Rdest, Rsrc1, Src2` — *Set Equal* †
Met le registre `Rdest` à 1 si le registre `Rsrc1` est égal à `Src2`, et à 0 sinon.

`sge Rdest, Rsrc1, Src2` — *Set Greater Than Equal* †
`sgeu Rdest, Rsrc1, Src2` — *Set Greater Than Equal Unsigned* †
Met le registre `Rdest` à 1 si le registre `Rsrc1` est plus grand que, ou égal à `Src2`, et à 0 sinon.

`sgt Rdest, Rsrc1, Src2` — *Set Greater Than* †
`sgtu Rdest, Rsrc1, Src2` — *Set Greater Than Unsigned* †
Met registre `Rdest` à 1 si le registre `Rsrc1` est plus grand que `Src2`, et à 0 sinon.

`sle Rdest, Rsrc1, Src2` — *Set Less Than Equal* †
`sleu Rdest, Rsrc1, Src2` — *Set Less Than Equal Unsigned* †

[4] sra 100100 2 = 111001

Met le registre `Rdest` à 1 si le registre `Rsrc1` est plus petit ou égal à `Src2`, et à 0 sinon.

`slt Rdest, Rsrc1, Src2` *Set Less Than*
`slti Rdest, Rsrc1, Imm` *Set Less Than Immediate*
`sltu Rdest, Rsrc1, Src2` *Set Less Than Unsigned*
`sltiu Rdest, Rsrc1, Imm` *Set Less Than Unsigned Immediate*
Met le registre `Rdest` à 1 si le registre `Rsrc1` est plus petit que `Src2` (or `Imm`), et à 0 sinon.

`sne Rdest, Rsrc1, Src2` *Set Not Equal* †
Met le registre `Rdest` à 1 si le registre `Rsrc1` is not equal to `Src2`, et à 0 sinon.

Instructions de contrôle

b

`label` *Branch instruction* †
Saut (court) à `label`.

`beq Rsrc1, Src2, label` *Branch on Equal*
Saut (court) conditionnel à `label` si le contenu de `Rsrc1` est égale à `Src2`.

`bgt Rsrc1, Src2, label` *Branch on Greater Than* †

`blt Rsrc1, Src2, label` *Branch on Less Than* †

`bne Rsrc1, Src2, label` *Branch on Not Equal*

`j label` *Jump*
Saut long à `label`.

`jal label` *Jump and Link*
Saut long à `label`. Sauve l'adresse de la prochaine instruction dans le registre 31.

`Src2` est un registre ou une constante. Les sauts de type *branch* utilisent un champ de décalage à 16 bits, donc on peut sauter $2^{15} - 1$ *instructions* en avant ou 2^{15} instructions en arrière. Les sauts de type *jump* utilisent un décalage sur 26 bits.

Appels système

SPIM fournit quelques appels système minimalistes: le programme charge dans `$v0` le code de l'appel et les arguments dans les registres `$a0`...`$a3` (ou `$f12` les valeurs en virgule flottante). Le resultat se trouve dans `$v0` (ou `$f0`).

Appels système

Par exemple, pour imprimer "`deux plus trois = 5`", on peut écrire:

Service	System Call Code	Arguments	Result
print_int	1	$a0 = integer	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk[5]	9	$a0 = amount	adresse (in $v0)
exit	10		

```
        .data
str:    .asciiz "deux plus trois = "
        .text
        li $v0, 4           # system call code for print_str
        la $a0, str         # address of string to print
        syscall             # print the string

        li $v0, 1           # system call code for print_int
        li $a0, 5           # integer to print
        syscall             # print it
```