Assembleur

Rappels d'architecture

Un ordinateur se compose principalement

- d'un processeur,
- de mémoire.

On y attache ensuite des périphériques, mais ils sont optionnels.

données : disque dur, etc

entrée utilisateur : clavier, souris

sortie utilisateur : écran, imprimante

processeur supplémentaire : GPU

Le processeur

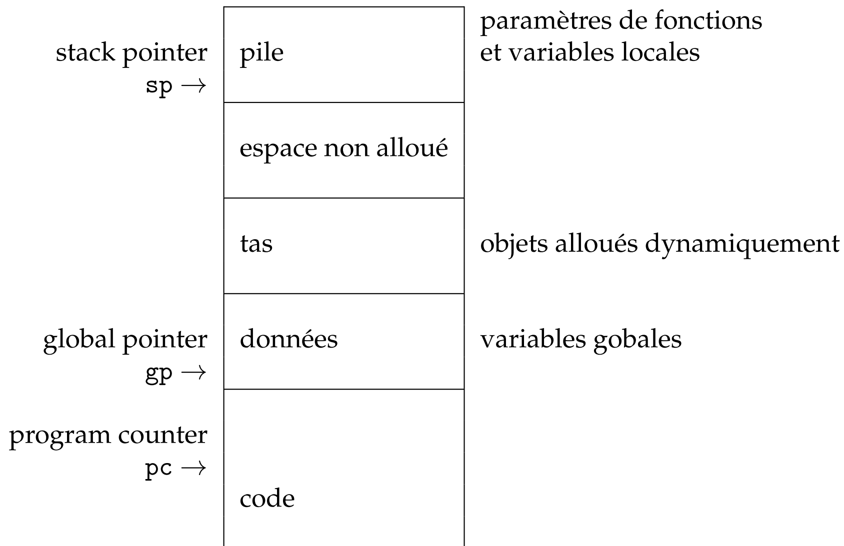
- Le processeur lit et écrit des informations en mémoire
- Il peut de plus effectuer des **opérations** arithmétiques et logiques
- Chaque action qu'il peut effectuer est appelée **instruction**
- Les instructions effectuées par le processeur sont stockées dans la mémoire.
- Il dispose d'un petit nombre d'emplacements mémoire d'accès plus rapide, les **registres**.
- Un registre spécial nommé pc (program counter) contient l'adresse de la prochaine instruction à exécuter
- De façon répétée le processeur :
  1. lit l'instruction stockée à l'adresse contenue dans pc
  2. l'interprète ce qui peut modifier certains registres (dont pc) et la mémoire

CISC / RISC

C'est principalement le jeu d'instruction qui distingue les processeurs

- Les processeurs CISC (Complex Instruction Set Computer)
  - Nombre d'instruction élevé
  - Les instructions réalisent souvent les transferts vers et depuis la mémoire
  - peu de registres
  - Exemples : Intel 8068, Motorola 68000
- Les processeurs RISC (Reduced Instruction Set Computer)
  - Peu d'instructions
  - Les instructions opèrent sur des registres
  - Registres nombreux
  - Exemples :Alpha, Sparc, MIPS, PowerPC

Organisation de la mémoire



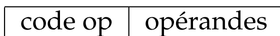
Organisation de la mémoire

- L'organisation de la mémoire est conventionnelle
- En théorie toutes les zones sont accessibles de la même manière

Langage machine

Une instruction de langage machine correspond à une instruction possible du processeur.
Elle contient :

- un code correspondant à opération à réaliser,
- les arguments de l'opération : valeurs directes, numéros de registres, adresses mémoire.



Langage machine

Si on ouvre un fichier exécutable avec un éditeur (hexadécimal), on obtient

```
...
01ebe814063727473747566662e6305f5f43544f525f4c
5f05f5f44544f525f4c4953545f5f05f5f4a43525f4c49
53545f5f05f5f646f5f676c6f62616c5f64746f72735f6
75780636f6d706c657465642e36353331064746f725f69
...
```

Langage ~~machine~~ lisible

Si on ouvre un fichier exécutable avec un éditeur (hexadécimal), on obtient

```
...
01ebe8140637274737475666 62e6305f5f43544f525f4c
5f05f5f44544f525f4c4953545f5f05f5f4a43525f4c49
53545f5f05f5f646f5f676c6f62616c5f64746f72735f6
75780636f6d706c657465642e36353331064746f725f69
...
```

C'est une suite d'instructions comme `01ebe814`, que l'on peut traduire directement de façon plus lisible :

```
add    $t7, $t3 , $sp
```

C'est ce qu'on appelle l'*assembleur*.

- L'assembleur est donc une *représentation* du langage machine.
- Il y a autant d'assembleurs que de type de processeurs différents.

MIPS

- processeur de type RISC
- sorti dans les années 1980, utilisé jusqu'au début 2000 (PlayStation 2)
- utilisé en informatique embarquée
- émulateurs :

spim	en ligne de commande,
qtspim	en version graphique (préférable).
mars	implémentation en JAVA.

Exemple MIPS

```
        .data
vars:   .word 5
        .word 10
        .text
__start:la    $t0, vars
        lw    $t1, 0($t0)
        lw    $t2, 4($t0)
saut:   bge   $t1, $t2, exit
        move  $a0, $t1
        li    $v0, 1
        syscall
        addi  $t1, $t1, 1
        j     saut
exit:   li    $v0, 10
        syscall
```

On y trouve :

- des mots clefs : `.data, .word, .text`
- des instructions : `lw, la, add, addi, bge ...`
- des registres : `$t0, $t1, $t2`
- des étiquettes qui correspondent à des adresses : `vars, saut,`

Espace mémoire

La zone de mémoire de données se déclare avec le mot-clef `.data`. C'est là que l'on stocke les "variables".

- Le "type" des variables renseigne uniquement sur la place en mémoire de chaque variable : ce ne sont en réalité que des adresses.
- Le "type" `.space` indique simplement que l'on réserve le nombre d'octet indiqués (sans les mettre à zéro).

```
     .data
c    .byte      'a'     ; octet
n1   .halfword   26     ; 2 octets
n2   .word       353    ; 4 octets
tab  .space      40
```

Programme

La zone mémoire du programme est signalée par le mot-clef `.text`.

```
       .text
__start:la    $t0, vars
        lw    $t1, 0($t0)
        lw    $t2, 4($t0)
saut:   bge   $t1, $t2, exit
        move  $a0, $t1
        li    $v0, 1
        syscall
        addi  $t1, $t1, 1
        j     saut
exit:   li    $v0, 10
        syscall
```

On y lit :

- des instructions,
- des *étiquettes* (`saut`), c'est-à-dire des adresses dans le code.

Registres

Un processeur MIPS dispose de 32 registres de 4 octets chacun.
On les distingue des étiquettes par le signe '`$`'.
On travaillera principalement avec les suivants :

numéro	nom	utilité
8-15, 24, 25	`$t0-$t9`	registres courants
31	`$ra`	adresse de retour (après un saut)
29	`$sp`	pointeur de haut de pile
0	`$0`	la valeur zéro
2, 3	`$v0,$v1`	appel et résultat de routines
4-7	`$a0-$a3`	arguments des routines

Instructions

Trois types d'instructions

- instructions de transfert entre registres et mémoire
  - chargement
  - sauvegarde
- instructions de calcul
  - additions
  - multiplications
  - opérations logiques
  - comparaisons
  - sauvegarde
- instructions de saut
  - sauts inconditionnels
  - sauts conditionnels
  - sauvegarde
- appels système

Chargement

load word	lw
load immediate	li
load address	la
load byte	lb
load byte unsigned	lbu
load halfword	lh
load halfword unsigned	lhu

Ces instructions permettent de changer le contenu des registres

Chargement

- `lw dest, adr`
  - charge dans le registre `dest` le contenu de l'adresse `adr`
- `lw dest,offset(base)`
  - ajoute `offset` octets à l'adresse contenue dans le registre `base` pour obtenir une nouvelle adresse
  - `offset` doit être une constante
  - le mot stocké à cette adresse est chargé dans le registre `dest`
- `li dest,const`
  - charge dans le registre `dest` la constante `const`
- `la dest, adr`
  - charge dans le registre `dest` l'adresse `adr`

Sauvegarde

`store word`	`sw`
`store halfword`	`sh`
`store byte`	`b`
`move`	`move`

- `sw source, adr`
  - sauvegarde le registre `source` à l'adresse `adr`
- `sw source, offset(base)`
  - ajoute la constante `offset` à l'adresse contenue dans le registre `base` pour obtenir une nouvelle adresse
  - sauvegarde le registre `source` à cette adresse
- `move dest, source`
  - sauvegarde le contenu du registre `source` dans le registre `dest`

Additions

`add`	`add`
`add immediate`	`addi`
`add immediate unsigned`	`addiu`
`add unsigned`	`addu`
`substract`	`sub`
`substract unsigned`	`subu`

- `add r0, r1, r2`
  additionne le contenu des registres `r1` et `r2` et sauvegarde le résultat dans le registre `r0`
- `addi r0, r1, c`
  additionne le contenu du registre `r1` avec la constante `c` et sauvegarde le résultat dans le registre `r0`

Additions

- Les instructions `add`, `addi` et `sub` provoquent une exception du processeur en cas de dépassement de capacité .
- Une exception provoque un appel de procédure à un gestionnaire
- Les variantes non-signées `addu`, `addiu` et `subu` ne provoquent pas d'exceptions.

Multliplications

Multiply	mult
Multiply unsigned	multu
Divide	div
Divide unsigned	divu

- `mult r0, r1`
  multiplie le contenu des registres `r0` et `r1`
- Une multiplication de deux entiers à 32 bits peut prendre 64 bits. L'opération `mult` sépare le résultat dans deux registres cachés `Lo` et `Hi`. On récupère la valeur avec les opérations `mflo, mfhi` (*move from lo, hi*).
- L'opération de division utilise ces mêmes registres pour enregistrer le quotient (dans `Lo`) et le reste (dans `Hi`).

```
div   $t4, $t5
mflo  $t2          ; t2 = t4 / t5
mfhi  $t3          ; t3 = t4 % t5

mult  $t4, $t5
mflo  $t0          ; t0 = t4 * t5 si ce n'est pas trop grand
```

Opérations logiques

`and`	`and`
`and immediate`	`andi`
`nor`	`nor`
`or`	`or`
`or immediate`	`ori`
`exclusive or`	`xor`
`exclusive or immediate`	`xori`

- `and r0, r1, r2`
  effectue un et logique bit à bit du contenu des registres `r1` et `r2` et sauvegarde le résultat dans le registre `r0`
- `andi r0, r1, c`
  effectue un et logique bit à bit du contenu du registre r1 et de la constante c et sauvegarde le résultat dans le registre r0

Comparaisons

`set less than`	`slt`
`set less than immediate`	`slti`
`set less than immediate unsigned`	`sltiu`
`set less than unsigned`	`sltu`

- `slt r0, r1, r2`
  charge `1` dans le registre `r0` si le contenu de `r1` est inférieur à celui de `r2`, charge `0` sinon
- `slti r0, r1, c`
  charge `1` dans le registre `r0` si le contenu de `r1` est inférieur à la constante `c`, charge `0` sinon

Sauts inconditionnels

Jump	`j`
Jump and link	`jal`
Jump register	`jr`
Jump and link register	`jalr`

- `j adr`
  va à l'adresse `adr`
- `jal adr`
  sauvegarde en plus `$pc` dans le registre `$ra`
- `jr $t0`
  va à l'adresse contenue dans le registre `$t0`
- `jalr $t0`
  sauvegarde en plus `$pc` dans le registre `$ra`

Sauts conditionnels

`Branch on equal`	`beq`
`Branch on not equal`	`bne`
`Branch on less than`	`blt`
`Branch on greater than`	`bgt`
`Branch on less or equal than`	`ble`
`Branch on greater or equal than`	`bge`

- `beq r1, r2, adr`
  si les contenus des registres `r1` et `r2` sont égaux, saute à l'adresse `adr`

Appels système

MIPS permet de communiquer avec le système de façon simple par la commande `syscall`.
La fonction utilisée est déterminée selon la valeur de `$v0`.

`$v0`	commande	argument	résultat
1	`print_int`	`$a0` = entier à lire	
4	`print_string`	`$a0` = adresse de chaîne	
5	`read_int`		`$v0` = entier lu
8	`read_string`	`$a0` = adresse de chaîne, `$a1` = longueur max	
10	`exit`		

Et plus encore ...

Il reste des zones de MIPS que l'on n'utilisera pas :

- coprocesseur pour flottants
- exceptions (retenue de l'addition, etc ...)

Voir la table de référence sur la page web du cours.
Par exemple, on pourra utiliser l'opération `sll` (*shift left logical*) pour multiplier les indices de tableau par 4 :

```
add  $t1, $t1, $t1
add  $t1, $t1, $t1

sll  $t1, $t1, 2
```

Compilation directe

L'assembleur est une version "lisible" du langage machine, mais on pourrait en écrire directement.

```
add    $t7, $t3, $sp
```

- Le registre `$t0` est en fait le numéro 8, et `$sp` le numéro 29.
- Le code de l'opération est ici coupé en deux : le premier (0) indique c'est une opération arithmétique, le deuxième de quelle opération arithmétique précisément.

op1	$t7	$t3	$sp	(inutilisé)	op2
000000	01111	01011	11101	00000	010100

Compilation directe

L'assembleur est une version “lisible” du langage machine, mais on pourrait en écrire directement.

```
add    $t7, $t3, $sp
```

- Le registre `$t0` est en fait le numéro 8, et `$sp` le numéro 29.
- Le code de l'opération est ici coupé en deux : le premier (0) indique c'est une opération arithmétique, le deuxième de quelle opération arithmétique précisément.

op1	$t7	$t3	$sp	(inutilisé)	op2
000000	01111	01011	11101	00000	010100

Soit, en hexadécimal, `01ebe814`.
On pourrait donc compiler directement un exécutable MIPS.

Pseudo-instructions

Certaines opérations ne sont pas des opérations réelles, et ne peuvent pas être traduites directement. Par exemple, l'instruction `li` (chargement d'une valeur directe) doit être encodée sous une autre forme.

```
li      $t0, 12
```

```
ori     $t0, $0, 12
```

C'est toutefois une traduction mineure, que l'on ferait si on produisait du code exécutable, mais qui n'apporte pas grand-chose.

Références

- Cours d'architecture de Peter Niebert :
  `http://www.cmi.univ-mrs.fr/~niebert/archi2012.php`
- Introduction au MIPS :
  `http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm`
- Table de référence du MIPS :
  `http://pageperso.lif.univ-mrs.fr/~alexis.nasr/Ens/Compilation/mipsref.pdf`
- Cours de compilation de François Pottier :
  `http://www.enseignement.polytechnique.fr/informatique/INF564/`