

Architecture des ordinateurs

Chapitre 3 : Architecture matérielle - Processeur et jeu d'instructions

* *

Clément MOREAU, Olivier PLOTON

{clement.moreau, olivier.ploton}@univ-tours.fr

Université de Tours ~ Département informatique de Blois

Licence 2 - Informatique

Sommaire

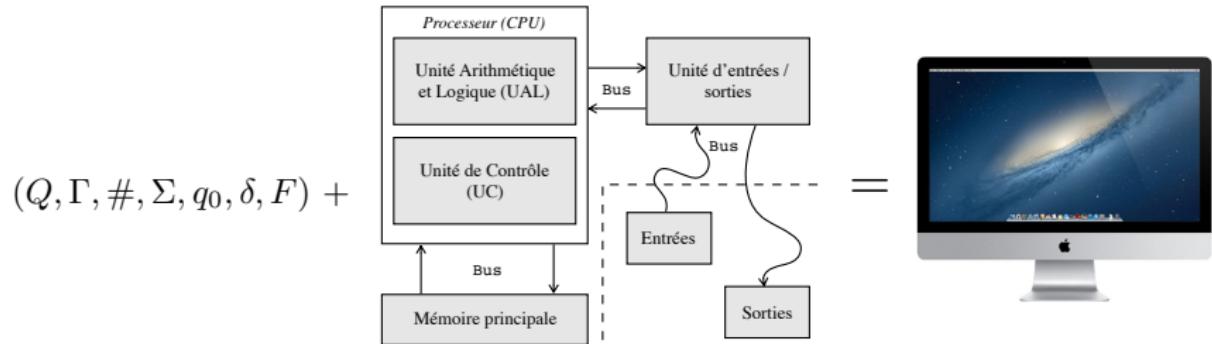
- 1 Généralités
- 2 Processeur (CPU)
- 3 Jeu d'instructions
- 4 Assembleur avancé – Récursion et Pile
- 5 Optimisations CPU et Pipelines

Généralités

Mais qu'est-ce qu'un ordinateur ?

Un *ordinateur* est une machine disposant d'interfaces de communication (Entrées et Sorties) et permettant le traitement de l'information via la notion de *programme*.

Un ordinateur peut être synthétisé par l'union d'un modèle abstrait (Machine de Turing) et d'un modèle physique (Architecture de Von Neumann).



$$(Q, \Gamma, \#, \Sigma, q_0, \delta, F) +$$

Généralités

Et, qu'est-ce qu'un programme ?

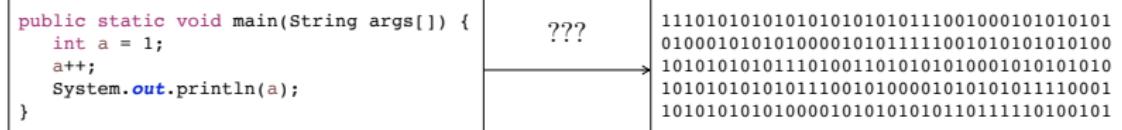
Un *programme* est une suite d'instructions élémentaires pouvant être exécutées par un ordinateur.

On appelle *processus* un programme en cours d'exécution.

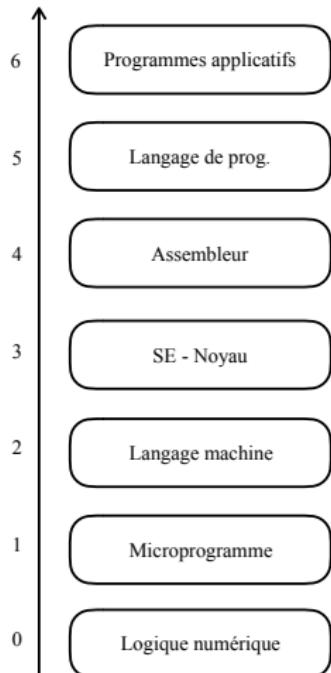
Pour éviter d'avoir à rentrer les codes binaires correspondants directement aux instructions en mémoire, on utilise un langage de haut niveau (C, C++, Java, etc) ou mnémonique (Assembleur).

Cependant, on a vu précédemment que les ordinateurs ne comprennent que le langage binaire. Comment se faire comprendre ?

⇒ Cela dépend du langage, mais c'est en partie le rôle de la *compilation*.



Généralités

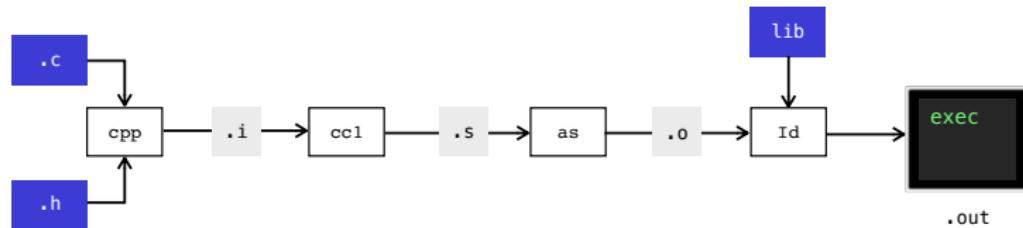


On distingue plusieurs niveaux d'abstraction au niveau de la machine :

- *Logique numérique* : Abstraction des circuits électroniques par portes logiques
- *Microprogramme* : Séquence d'étapes pour réaliser une instruction machine.
- *Langage machine* : Jeu d'instructions propre à l'architecture.
- *SE - Noyau* : Ensemble des fonctionnalités dédiées à la gestion et l'allocation des ressources matérielles et logiciels de l'ordinateur.

Généralités

La vie d'un programme en C :



- ➊ *Préprocesseur* : traitement sur le texte du fichier source. – `gcc -E file.c -o file.i`
- ➋ *Compilation* : transformation du source en langage assembleur et optimisations. – `gcc -S file.i`
- ➌ *Assemblage* : transformation du code assembleur en code machine à l'aide d'une table de symboles. – `gcc -c file.s`
- ➍ *Édition de liens* : liaison entre différents codes machines pour la création du programme cible. – `gcc -o exec file1.o file2.o ...`

Généralités

```
1 #include <stdio.h>
2 #define MOT "World"
3 #define MAX(A,B) (A) > (B) ? A : B
4
5 int main(void) {
6     int i = 2;
7     int j = 3;
8     int max; // Ceci est un commentaire
9
10    max = MAX(i ,j );
11    printf("Hello" MOT);
12    return 0;
13 }
```



```
1 ...
2 extern int __vsprintf_chk (char * restrict, size_t, int, size_t,
3     const char * restrict, va_list);
4 # 412 "/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.
5     platform/Developer/SDKs/MacOSX10.14.sdk/usr/include/stdio.h" 2
6     3 4
7 # 2 "hello_world.c" 2
```



```
6
7 int main(int argc, char * argv[])
8     int i = 2;
9     int j = 3;
10    int max;
11
12    max = (i) > (j) ? i : j;
13    printf("Hello " "World");
14    return 0;
15 }
```

```
.section      __TEXT,__text,regular,pure_instructions
.build_version    macos, 10, 14      sdk,version 10, 14
.globl _main          ## — Begin function main
.p2align   4, 0x90
in:           ## @main
```



```

## %%bb.0:
    pushq   %rbp
    .cfi_offset %rbp, -16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_offset %rbp, 16
    .cfi_register %rbp
    subq    $48, %rsp
    movl    $0, -4(%rbp)
    movl    %edi, -8(%rbp)
    movq    %rsi, -16(%rbp)
    movl    $2, -20(%rbp)
    movl    $3, -24(%rbp)
    movl    -20(%rbp), %edi
    cmpb    -24(%rbp), %edi
    jle     LBB0_2

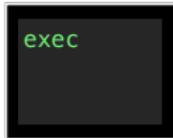
```

```

## %%b.1:
    movl    -20(%rbp), %eax
    movl    %eax, -32(%rbp)      ## 4-byte Spill
    jmp     LBB0_3

LBB0_2:
    movl    -24(%rbp), %eax
    movl    %eax, -32(%rbp)      ## 4-byte Spill

```



**Édition des
liens /
Exécution**

Généralités

On s'intéresse ici à la mécanique qui permet d'exécuter les programmes, c'est-à-dire au modèle proposé à l'initial par Von Neumann (1945) (et qui a peu évolué depuis).

On note trois composants principaux :

- Le *processeur* qui traite (UAL) et commande (UC) les instructions.
- La *mémoire principale* où sont stockées les données et instructions avec lesquelles le processeur travaille.
- Les *bus de données* qui assurent la communication interne entre les entités.
- Les périphériques d'entrées/sorties de l'information.

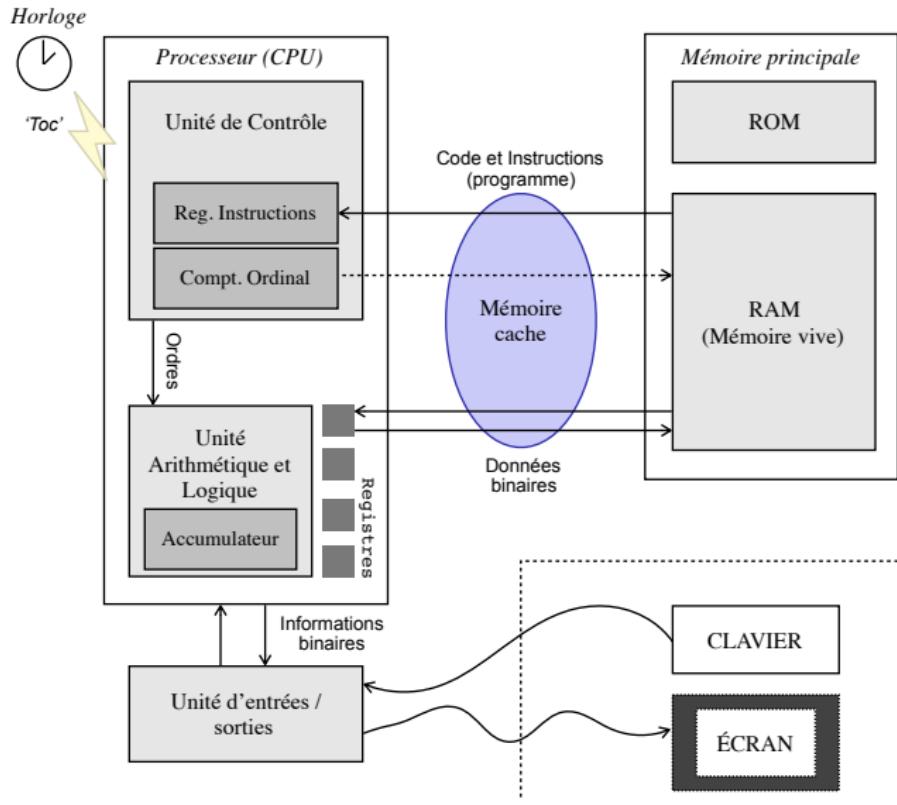
En fonctionnement synchrone :

- Une *horloge* cadence le travail des composants.
- Le processeur réalise une instruction élémentaire par cycle ('toc') horloge.

Généralités

FIGURE -

Schéma synthétique du fonctionnement d'un ordinateur



Processeur (CPU)

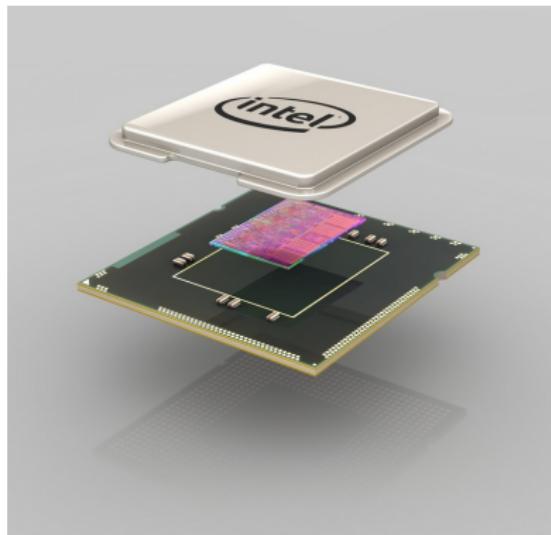


FIGURE - Assemblage d'un processeur
Intel®

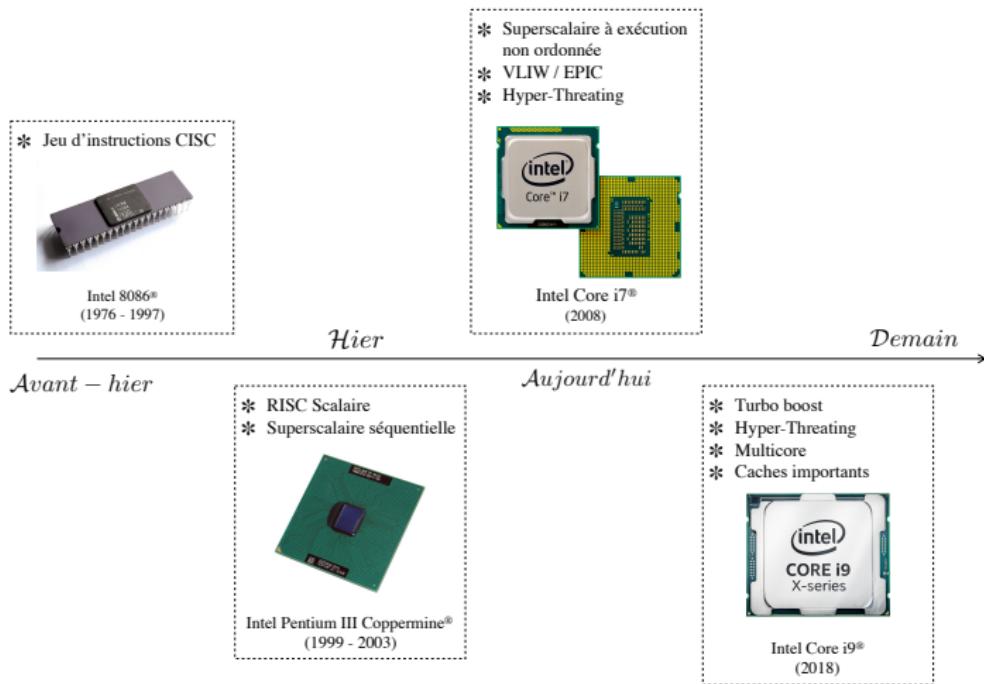
Le *processeur* (ou CPU) est considéré comme le cœur de l'ordinateur.

Sur la **FIGURE 3**, de bas en haut :

- Le *substrat*, relié au socket de la carte mère qui joue un rôle d'interface électrique et mécanique entre le processeur et le reste des composants,
- Le *circuit intégré* (ou die) issu des galettes de silicium (ou wafer) et où sont inscrits les circuits logiques,
- La *capsule* (ou heatspreader) est quant à elle l'interface thermique sur laquelle la solution de refroidissement est appliquée.

Optimisations CPU

Aujourd'hui, on en est où ?



Processeur (CPU)

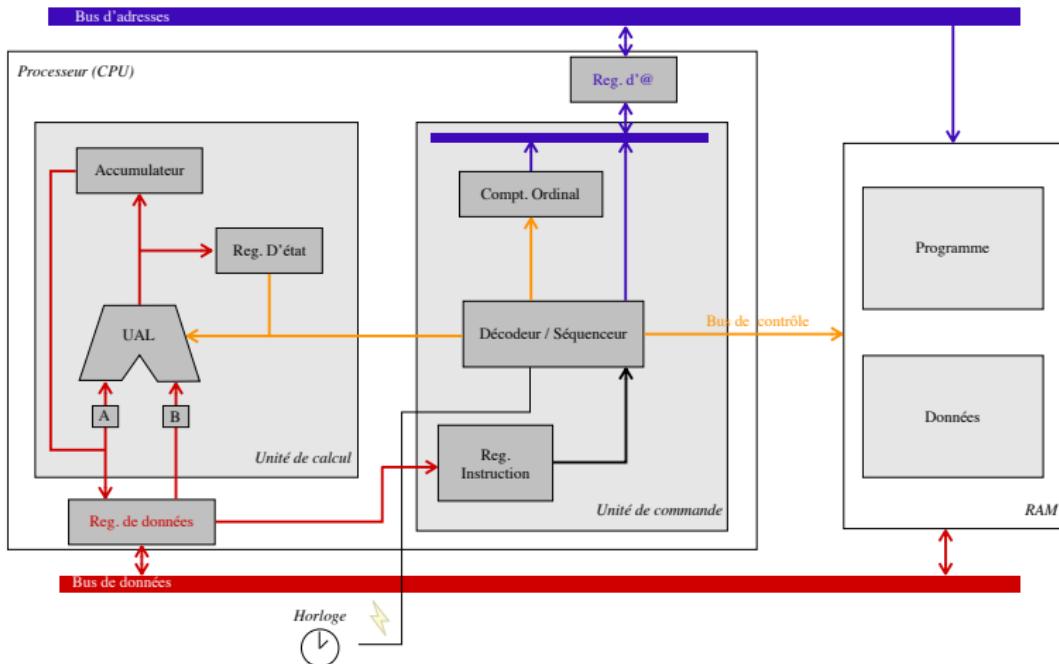


FIGURE - Schéma du fonctionnement d'un processeur

Processeur (CPU)

Registres

Les registres sont des petites mémoires internes, très véloques, utilisées pour stocker temporairement une données, instruction ou adresse.
Un registre stocke 8, 16, 32 ou 64 bits selon les modèles d'ordinateurs.

L'Unité de Contrôle

Circuits coordonnant l'activité de l'UAL, des registres et des interactions avec la mémoire.

L'Unité Arithmétique et Logique

Circuits réalisant des opérations de calcul de l'ordinateur :

- Arithmétique (Entiers, flottants).
- Logiques (AND, OR, XOR, Décalage).
- Comparaison ($=, \geq, \leq$, etc.)

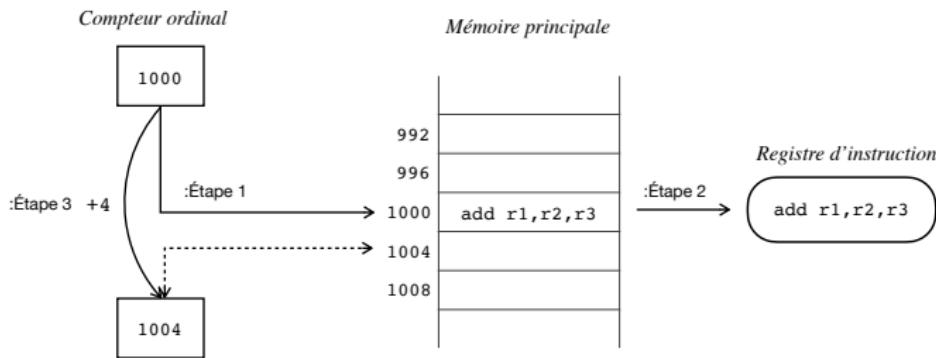
Processeur (CPU)

Le processeur est le responsable de l'exécution de toute instruction réalisée par l'ordinateur. On considère le cycle d'exécution des instructions suivant :

- ① *Lecture de l'instruction* - Lecture de l'adresse de l'instruction et mise à jour du compteur ordinal
- ② *Chargement de l'instruction* - Récupération de l'instruction depuis la RAM vers le registre d'instruction.
- ③ *Décodage de l'instruction* - Analyse du code contenu dans le registre d'instruction et lecture des opérandes.
- ④ *Chargement registres* - Recherche et rangement des opérandes vers les registres.
- ⑤ *Exécution de l'instruction* - Calcul UAL, modification du compteur ordinal si instruction de branchement, etc.
- ⑥ *Accès mémoire* - Accès mémoire en lecture ou écriture pour les instructions de rangement/chargement.
- ⑦ *Écriture mémoire* - Écriture des résultats en RAM/registres et modification du registre d'état.

Processeur (CPU)

- Le *compteur ordinal* est un registre qui contient l'adresse de la prochaine instruction à exécuter. Le programmeur n'a pas directement accès à ce registre mais peut le modifier via les instructions de branchement.



Chaque adresse mémoire correspond à 1 octet et une instruction est longue de plusieurs octets ; le compteur doit donc être augmenté du nombre d'octets de l'instruction, et non de 1.

Processeur (CPU)

Pour une poignet de registres supplémentaires...

- L'*accumulateur* est un registre spécial utilisé pour stocker les résultats intermédiaires de l'UAL des calculs longs afin d'éviter de les verser puis les recharger depuis la RAM.
- Lorsqu'une instruction est récupérée en mémoire pour être exécutée, elle est mémorisée dans le *registre d'instruction*. Ce registre est entièrement géré par le décodeur, le programmeur n'y a pas accès.
- Registre d'état de la *pile* du programme : En général 2.

Autres éléments du CPU : les caches

- Capacité de stockage de quelques Mo.
- Vitesse : Registres > Caches > mem. principale
- Occupe près de la moitié de la surface de la puce
⇒ Coûte très cher !!

Jeu d'instructions

Jeu d'instructions

Un jeu d'instructions est le vocabulaire utilisé par la machine au niveau architecture. Il contient en général des instructions pour :

- Le *transfert* pour déplacer les données entre la mémoire et les registres.
- *Arithmétiques, logiques et de décalages pour les calculs de l'UAL.*
- Instructions de *branchement* pour réaliser des sauts en mémoire.

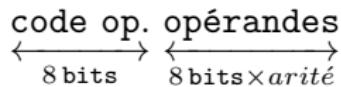
On utilise une notation symbolique pour coder en assembleur :

- Les codes opérations sont désignés par des *mnémoniques* (**add**, **lw**, ...)
- Les adresses mémoires sont nommées par des registres ou étiquettes programmes établis par le programmeur.

- **add \$r1, \$r1, \$r2**
- **lw \$r1, \$r2**

Jeu d'instructions

Une instruction peut être décomposée en deux parties : l'opération elle-même et ses opérandes. On appelle la première le *code opération*, le reste est utilisé pour identifier les *opérandes* selon un *mode d'adressage* donné.



La taille des **codes op.** et des opérandes peut varier en fonction des processeurs. La taille standard étant de 8 bits. On peut déterminer alors $2^8 = 256$ instructions différentes.

Jeu d'instructions

Il existe deux architectures majeures au sein des processeurs qui se distinguent par l'ensemble d'instructions élémentaires :

- *CISC* (Complex Instruction Set Computer)
 - **Constat** : La mémoire est très lente par rapport au processeur et 1 instruction = 1 accès mémoire.
 - **Solution** : On va chercher à minimiser le nombre d'instructions.
⇒ Instructions plus complexes et plus complètes.
- *RISC* (Reduced Instruction Set Computer)
 - **Constat** : 80% du traitement d'un programme de haut niveau correspond à 20% de l'ensemble des instructions disponibles.
 - **Solution** : On réduit le jeu d'instructions au 20% plus courantes et les optimiser en améliorant leur vitesse de traitement.

Jeu d'instructions

Avantages

- RISC
 - Petit jeu d'instructions standards,
 - Traitement efficace,
 - Possibilité de pipeline.
- CISC
 - Programmation de plus haut niveau,
 - Compilateur simple,
 - Toute instruction peut accéder en mémoire.

Inconvénients

● RISC

- Programmes volumineux,
- Compilation complexe,
- Seules les instructions `load` et `store` accèdent à la mémoire.

● CISC

- Beaucoup d'instructions et de modes d'adressage différents,
- Processeur complexe (surtout décodeur),
- Exécution complexe et peu performante.

Jeu d'instructions

Ainsi, ces types de processeur possèdent chacun différents jeux d'instructions (ou langages d'assemblage).

Le jeu d'instructions définit :

- Quelles sont les instructions supportées par le processeur,
- Quels sont les registres du processeur manipulables par le programmeur,
- Comment les instructions et les opérandes sont représentées en mémoire.

On peut néanmoins préciser que les architectures (et donc les jeux d'instructions) de type RISC sont en nette supériorité par rapport à celles de type CISC, celles-ci étant tombées en désuétude après la migration de x86 de conception initiale CISC vers une conception RISC, soit à la sortie du microprocesseur Pentium Pro (1995).

Jeu d'instructions

On a vu que le processeur lit les instructions *séquentiellement* selon la lecture du compteur ordinal. Les programmes haut-niveau sont alors mis sous *forme linéaire* en Assembleur.

En Assembleur, on décompose tout !

- Plus aucun type de donnée ⇒ Organiser en mémoire les données
- Plus aucune variable ⇒ Quelques registres (Pile et tas) et de la mémoire
- Plus aucune structure de contrôles ou de données ⇒ Les réinventer avec les mnémoniques assembleur

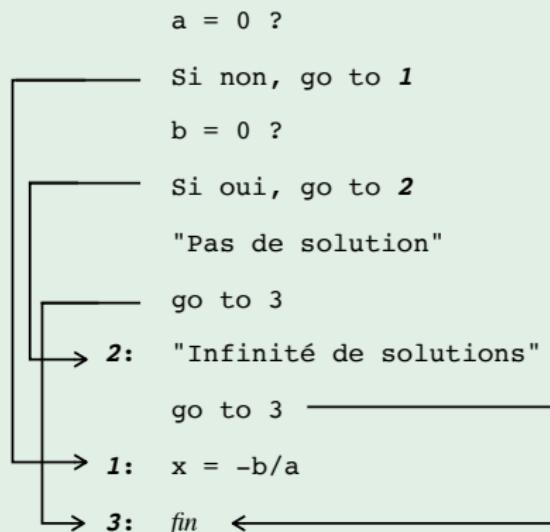
Les *instructions de branchement* permettent d'indiquer l'adresse de la prochaine instruction à exécuter et de simuler les structures de contrôle connues.

Jeu d'instructions

Exemple

Soit la forme linéaire de l'algorithme de résolution de l'équation du premier degré $ax + b = 0$.

Les symboles **1**, **2** et **3** désignent une adresse mémoire dans la séquence d'instructions, on les appelle des «*étiquettes*». Lors de la traduction en langage machine d'un programme, le programme d'assemblage associe à chaque étiquette l'adresse mémoire correspondante et effectue une instruction de branchement. Dans notre exemple, les branchements sont conditionnels.



Jeu d'instructions

Soient les registres \$d, \$s, \$t.

Les instructions arithmétiques

Nom	Commande	Opération
Addition	add \$d, \$s, \$t	\$d \leftarrow \$s + \$t
Soustraction	sub \$d, \$s, \$t	\$d \leftarrow \$s - \$t
Multiplication	mul \$d, \$s, \$t	\$d \leftarrow \$s \times \$t
Division	div \$s, \$t	Hi \leftarrow \$s % \$t Lo \leftarrow \$s / \$t

On récupère le contenu de Hi et Lo grâce à **mfhi** \$r et **mflo** \$r.

Jeu d'instructions

Comment traiter une expression arithmétique ?

Soit l'expression arithmétique suivante :

$$E = \frac{(a + b)^2 - c}{d}$$

On suppose que $(a + b)^2 < 2^{32}$ et que l'on procède à une division entière.

On considère le contenu des registres suivants :

Aff. reg.		Val. var. prog
\$r1	\leftarrow	a
\$r2	\leftarrow	b
\$r3	\leftarrow	c
\$r4	\leftarrow	d

On déposera l'expression E dans un registre \$r0.

Jeu d'instructions

Une réponse possible

```
add $r1, $r1, $r2 # a + b
mul $r1, $r1, $r1 # (a + b)2
sub $r1, $r1, $r3 # (a + b)2 - c
div $r1, $r4      #  $\frac{(a+b)^2 - c}{d}$ 
mflo $r0          # On récupère le quotient dans lo
```

Oui, c'est plus long...

Jeu d'instructions

Soient les registres \$d, \$s, \$t.

Les instructions logiques

Nom	Commande	Opération
And	<code>and \$d, \$s, \$t</code>	$$d \leftarrow \$s \& \$t$
Or	<code>or \$d, \$s, \$t</code>	$$d \leftarrow \$s \$t$
Xor	<code>xor \$d, \$s, \$t</code>	$$d \leftarrow \$s \wedge \$t$
Set on less than	<code>slt \$d, \$s, \$t</code>	$$d \leftarrow (\$s < \$t)$
Décalage à gauche	<code>sll \$d, \$s, \$t</code>	$$d \leftarrow \$s \ll \$t$
Décalage à droite	<code>srl \$d, \$s, \$t</code>	$$d \leftarrow \$s \gg \$t$

Jeu d'instructions

Il existe plusieurs modes d'adressage qui permettent de spécifier la localisation des données :

- *Adressage immédiat*
- *Adressage par registre*
- Adressage direct
- *Adressage indirect par registre*
- *Adressage indirect avec déplacement*
- Adressage indirect indexé

On détaille certains parmi les plus utilisés en assembleur Mips.

Jeu d'instructions

L'adressage *immédiat* est le plus simple, ce mode est utilisé pour charger les constantes. Soient une instruction i , $\$r_1$ et $\$r_2$ des registres et n une donnée quelconque dépendant de la signature de i . Une signature par adressage immédiat est de la forme :

$$i : \$r_1 \leftarrow n \text{ ou } i : \$r_2 \leftarrow \$r_1, n.$$

Exemple

En Mips, une instruction par adressage immédiat est en général spécifiée par la lettre 'i' dans son nom :

- **addi** : $R \leftarrow R, \mathbb{N}$ (Addition immediate)
- **li** : $R \leftarrow \mathbb{N}$ (Load immediate)

li \$a0,0
addi \$a0,\$a0,1

Jeu d'instructions

L'adressage *par registre* est utilisé pour les instructions de transfert de contenu. Soient une instruction i , et deux registres $\$r_1, \r_2 , une signature par adressage par registre est de la forme :

$$i : \$r_1 \leftarrow \$r_2$$

Exemple

- **move** : $R \leftarrow R$ (Transfert registre)
 $move \$r1, \$r2 \iff addi \$r1, \$r2, 0$
- **la** : $R \leftarrow \text{étiquette}$ (Load address)

```
.data
chaine : .asciiz ":)"
.text
main    : la      $t0, chaine
          move   $t1, $t0      # t1 = t0
```

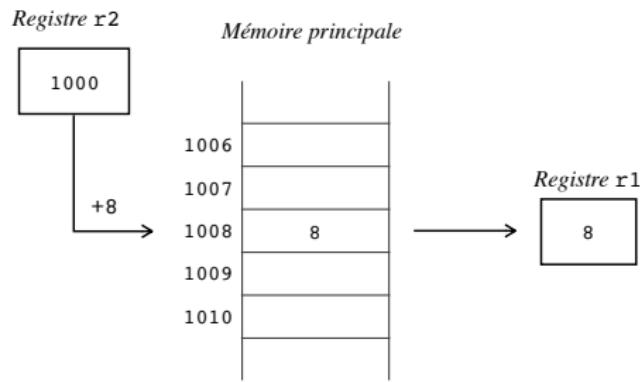
Jeu d'instructions

L'adressage *indirect par registre* est fortement utilisé dans le cadre des chaînes de caractères et des tableaux.

On spécifie l'adresse mémoire d'une donnée enregistrée dans registre $\$r_2$ afin de transférer la donnée enregistrée à cette adresse dans un registre $\$r_1$. À l'adresse $\$r_2$ peut s'ajouter un décalage de C octets en mémoire.

La signature d'une instruction i d'adressage indirect par registre [avec décalage] est de la forme :

$$i : \$r_1 \leftarrow RAM[\$r_2 + C]$$



Jeu d'instructions

Les instructions d'adressage

Nom	Commande	Opération
Load Adress	la \$d, etiq	$\$d \leftarrow \text{etiq}$
Move	move \$d, \$s	$\$d \leftarrow \s
Load byte	lb \$d, C(\$s)	$\$d \leftarrow \text{RAM}[\$S + C]$
Store byte	sb \$d, C(\$s)	$\text{RAM}[\$S + C] \leftarrow \d

Les commandes **lb** et **sb** s'adapte aussi à un mot (**lw** et **sw**).

Jeu d'instructions

Exemple

- Soit un tableau `tab[]` d'entiers de 32 bits dont l'adresse est référencée par l'étiquette `tab`.
- Soit une variable `x` associée au registre `$v2`.

Quel est l'équivalent en Assembleur de l'instruction C :

```
tab[1] = x + tab[5];
```

Jeu d'instructions

Exemple

- Soit un tableau `tab[]` d'entiers de 32 bits dont l'adresse est référencée par l'étiquette `Tab`.
- Soit une variable `x` associée au registre `$v2`.

Quel est l'équivalent en Assembleur de l'instruction C :

```
tab[1] = x + tab[5];
```

```
la $v1, Tab
lw $t, 20($v1)      # 20 = 4 × 5 (i.e. taille_donnée × indice)
add $v2, $t, $v2
sw $v2, 4($v1)
```

Jeu d'instructions

Comment effectuer des tests et branchements ?

On utilise des instructions de branchement et des sauts sur étiquette qui manipulent le compteur ordinal.

Instructions similaires aux fameux `go to`.

C'est ces instructions qui vont nous permettre de simuler toutes les instructions telles que :

- If ... then ... [else ...]
- for(i : 0...(n-1)) {...}
- while(...) {...}

Jeu d'instructions

Instruction de branchement

Nom	Commande	Opération
Branch. =	<code>breq \$d, \$t, etiq</code>	Si ($d == t$) alors $\$PC \leftarrow etiq$
Branch. \neq	<code>bneq \$d, \$t, etiq</code>	Si ($d != t$) alors $\$PC \leftarrow etiq$
Branch. $>$	<code>bgt \$d, \$t, etiq</code>	Si ($d > t$) alors $\$PC \leftarrow etiq$
Branch. \geq	<code>bge \$d, \$t, etiq</code>	Si ($d \geq t$) alors $\$PC \leftarrow etiq$
Branch. $<$	<code>blt \$d, \$t, etiq</code>	Si ($d < t$) alors $\$PC \leftarrow etiq$
Branch. \leq	<code>ble \$d, \$t, etiq</code>	Si ($d \leq t$) alors $\$PC \leftarrow etiq$

Où le registre $\$PC$ désigne le compteur ordinal.

Jeu d'instructions

L'instruction If ... then ...

Soit le code :

```
if (x < 0)
    x = 0;
    x++;
```

En considérant que **x** est contenu dans un registre **\$r**. Quel est l'équivalent de ce code en Assembleur Mips ?

Jeu d'instructions

L'instruction if ... then ...

Soit le code :

```
if (x < 0)
    x = 0;
    x++;
```

En considérant que `x` est contenu dans un registre `$r`. Quel est l'équivalent de ce code en Assembleur Mips ?

```
bge $r, $zero, Suite
li $r, 0
Suite : addi $r, $r, 1
```

Jeu d'instructions

Et si on ajoute l'instruction `else` ?

On va utiliser l'instruction `j` de branchement inconditionnel.

Jump	j etiq	\$PC ← etiq
------	--------	-------------

L'instruction `if ... then ... else`

Soit le code :

```
if (x < 0)
    bge $r, $zero, Else
    li $r, 0
    j Suite
x = 0;
Else : addi $r, $r, 1
else
Suite : ...
x++;
```

Jeu d'instructions

L'instruction `while(...)`

Soit le code :

```
while (tab[i++] >= 0)  
    n++;
```

Soient les assignations suivantes :

- L'adresse du tableau `tab` est repérée par l'étiquette `Tab`,
- Le registre `$ind` contient la valeur de la variable `i`,
- Le registre `$n` contient la valeur de la variable `n`,
- Les registres `$t1`, `$t2`, `$t3` contiennent respectivement :
 - la valeur 4,
 - la valeur intermédiaire $4 \times i$,
 - la i -ème valeur de `tab`.

Quel est l'équivalent de ce code en Assembleur Mips ?

Jeu d'instructions

L'instruction while(...)

Soit le code :

```
while (tab[i++] >= 0)  
  
    x++;  
-----
```

```
        la $t0, Tab          # $t0 ← @tab[0]  
While : mul $t2, $t1, $int   # $t2 ← 4 × i  
                    add $t3, $t0, $t2  # $t3 ← @tab[i]  
                    lw $t3, ($t3)     # $t3 ← tab[i]  
                    blt $t3, $zero, Suite # test logique nié  
                    addi $n, $n, 1       # incrémentation  
                    addi $ind, $ind, 1    # incrémentation  
                    j While  
  
Suite : ...
```

Jeu d'instructions

L'instruction `for(i... (n-1))`

Soit le code :

```
for (i = 0; i < n; i++) {  
    ...  
}
```

Soient les assignations suivantes :

- \$r stocke la valeur i,
- \$n stocke la valeur n,

Quel est l'équivalent de ce code en Assembleur Mips ?

Jeu d'instructions

L'instruction `for(i... (n-1))`

Soit le code :

```
for (i = 0; i < n; i++) {  
    ...  
}
```

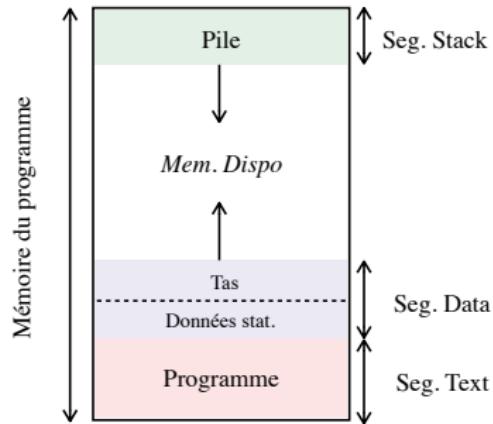
```
-----  
        li $r, 0          # $r ← 0  
For :   bge $r, $n, Suite  # branchemet si i >= n  
        ...                # Instructions de la boucle  
        addi $r, $r, 1      # Incrémentation  
        j For              # Retour à la boucle  
  
Suite :
```

Jeu d'instructions

Comment l'ordinateur fait-il pour distinguer en mémoire le programme et les données ?

On utilise différents segments mémoire :

- Le segment texte contient les instructions du programme.
- Le segment données contient les données statiques et dynamiques



Segments mémoire et pile

Pour se repérer dans la RAM, le CPU utilise des registres spéciaux dédiés à la gestion mémoire. Le segment de la *Pile* :

- Permet de décharger les registres.
- De les charger par d'autres données.
- De restituer leur état antérieur.
- Permet la récursivité (!).

Jeu d'instructions

Indiquer le segment mémoire

- `.text [adr]` : la donnée sera dans le segment texte.
- `.data [adr]` : la donnée sera stockée dans le segment des données.
- `.stack` : la donnée sera sur la pile.

Décrire et enregistrer les données

- `.space n` : définit un espace de n octets consécutifs dans le segment des données.
- `.ascii ch` : enregistre en mémoire la chaîne de caractères ch avec le caractère de fin de chaîne
- `.byte b1, ..., bn` : enregistre un tableau de n octets.
- `.double d1, ..., dn` : tableau de doubles.
- `.float f1, ..., fn` : tableau de flottants.
- etc.

Jeu d'instructions

Peut-on utiliser comme on veut chacun des 32 registres MIPS ?

Nom	Numéro	Fonct. conv.
\$zero	\$0	Constante 0
\$at	\$1	Rés. assembleur
\$v0-\$v1	\$2-\$3	Val. retour et Rés. fonction
\$a0-a3	\$4-\$7	Argument fonct.
\$t0-\$t7	\$8-\$15	Temp.
\$s0-\$s7	\$16-\$23	Temp. sauv.
\$t8-\$t9	\$24-\$25	Temp
\$k0-\$k1	\$26-\$27	Rés. noyau SE
\$gp	\$28	Pointeur global
\$sp	\$29	Pointeur Pile
\$fp	\$30	Pointeur Bloc
\$ra	\$31	Adresse retour

Eh bien, pas vraiment...

- Conventions logicielles déterminant le bon usage de l'architecture.
 - Programme
 - Compilation
- Contraintes matérielles.
- Contraintes du SE.

Jeu d'instructions

Et les E/S élémentaires ?

Méthodologie d'appel

- ① Écrire l'instruction pour charger le service voulu dans le registre \$v0.
- ② Charger les arguments nécessaires dans les registres \$a0 [et \$a1].
- ③ Écrire l'instruction `syscall`.
- ④ [Écrire les instructions pour `syscall` récupérer la valeur de retour de l'appel `syscall` dans \$a0]

Service	code Service	Arguments	Valeur de retour
Afficher entier	1	\$a0 : Stocke l'entier à print	
Afficher string	4	\$a0 : Stocke l'entier à print	
Lire entier	5		\$v0 : entier Lu
Lire string	8	\$a0 : Adresse du tampon \$a1 : Nb. max car. lus	
Exit	10		

Jeu d'instructions

Exemple de programme complet

```
##### Nombre de caractères d'une chaîne #####
        .data
Chaine : .asciiz "Hello world"
        .text
        .globl Main          # Visibilité du programme
Main :   la    $t0, Chaine # Chargement de la chaîne
        li    $a0, 0          # Initialisation
For :    lb    $t1, ($t0)  # Chargement indirect chaîne
                # P.S : 1 lettre = 1 octet
        beqz $t1, Fin       # Branchement conditionnel
        addi $a0, $a0, 1      # Incrémentation du nombre de caractères
        addi $t0, $t0, 1      # Incrémentation du tableau
        j     For             # Saut inconditionnel
Fin :    li    $v0, 1          # Appel procédure print_int
        syscall            # Impression du contenu de $a0
        li    $v0, 10         # Appel procédure fin de programme
        syscall
```

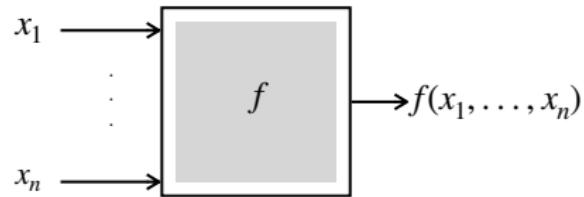
Récursion et Pile

Qu'est-ce qu'une fonction ?

C'est une boîte noire :

- Avec des entrées spécifiques (i.e. typées).
- Qui produit une certaine valeur de retour.

Ces caractéristiques nous suffisent et définissent une interface pour l'extérieur.



Concepts et architecture

- ➊ Type, variable, fonction sont des concepts de programmation sans réel analogue en architecture.
- ➋ Possibilité de créer de toute pièce en assembleur le concept de fonction :
 - Conserver cette notion de “boîte noire”.
 - Conserver cette notion d’interface.
 - Définit un mécanisme logiciel/matériel effectif repenant ces deux notions.

Récursion et Pile

Comment émuler des fonctions en Assembleur Mips?

```
int valAbs(int x) {
    if(x < 0) {
        return -x;
    }
    else
        return x;
}

int main(void) {
    int val, res;
    printf("Valeur ?\n");
    scanf(" %d", &val);
    res = valAbs(val) ←
    printf("|%d| = %d\n", val, res);
    return 0;
}
```

```
ValAbs :
inst_ValAbs_1
inst_ValAbs_1
...
inst_ValAbs_1
j suiv ←

Main :
inst_Main_1
inst_Main_2
...
j ValAbs ←

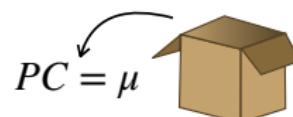
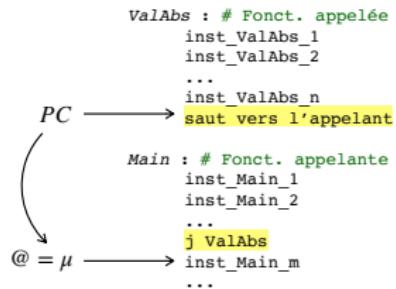
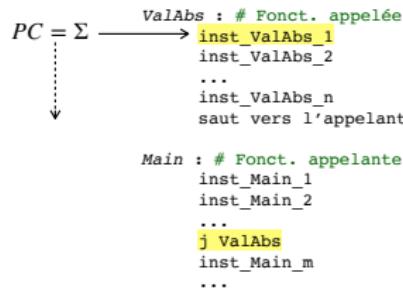
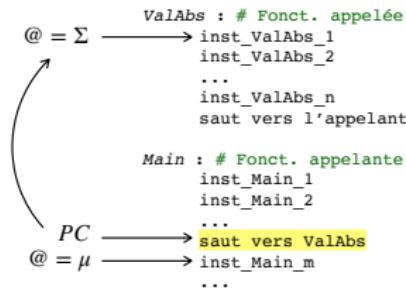
Suiv :
inst_Main_m
...
```

Notre fonction assembleur a connaissance de qui l'a appellée...

- ≠ boite noire
- Ce n'est pas le concept d'une fonction !

Récursion et Pile

Que doit faire le processeur pour réaliser un appel de fonction ?



Récursion et Pile

Comment l'ordinateur sait où revenir ?

Sauvegarde dans un registre l'adresse contenue dans le \$PC lors de l'appel.

Nom	Commande	Opération
Jump and link	<code>jal etiq</code>	$\$ra \leftarrow \$PC + 4$ $\$PC \leftarrow etiq$
Jump register	<code>jr \$r</code>	$\$PC \leftarrow \r

Utilisée avec \$ra, la commande jr permet de retourner à un point d'appel.

Récursion et Pile

Main :

```
inst_Main_1  
inst_Main_2  
...  
jal ValAbs  
inst_Main_m  
...
```

ValAbs :

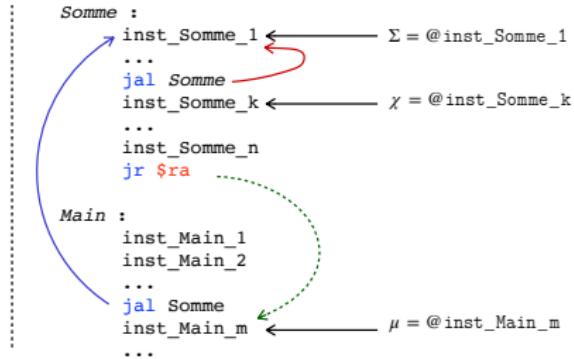
```
inst_ValAbs_1  
inst_ValAbs_2  
...  
inst_ValAbs_n  
jr $ra
```

Mais que se passe-t-il si la fonction appelée en appelle à son tour une autre ?

Récursion et Pile

```
int somme(int n) {
    if(n == 0)
        return 0;
    else
        return n + somme(n-1);
}

int main(void) {
    int n, res;
    printf("Entier ?\n");
    scanf(" %d", &n);
    res = somme(n);
    printf("Sum(%d) = %d", n, res);
    return 0;
}
```



- L'adresse de retour du premier appel est écrasée !!
⇒ Un saut avec liaison est insuffisant.
- Comment garder la trace des appels précédents ?
⇒ Définir et sauvegarder un contexte d'exécution.

- Premier appel :

$$\text{jal somme} \Leftrightarrow \begin{cases} \$ra \leftarrow \mu \\ \$PC \leftarrow \Sigma \end{cases} \quad \checkmark$$

- Deuxième appel :

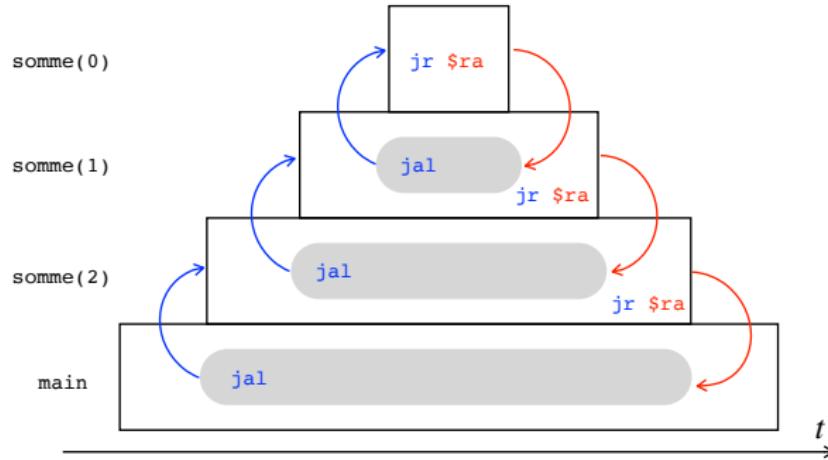
$$\text{jal somme} \Leftrightarrow \begin{cases} \$ra \leftarrow \chi \\ \$PC \leftarrow \Sigma \end{cases} \quad \times$$

Récursion et Pile

Comment organiser en mémoire ces contextes d'exécution ?

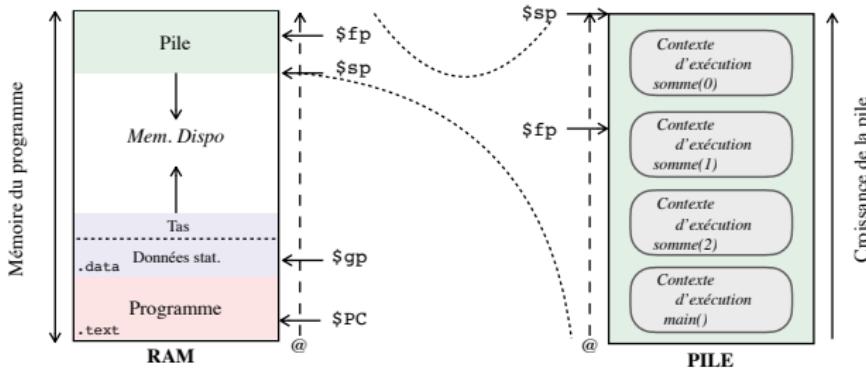
Une fonction se termine lorsque les appels internes de cette fonction sont tous complétés.

« Dernier entré, premier servi » \iff c'est le principe de la Pile !



Récursion et Pile

Où mettre / récupérer un contexte d'exécution de fonction ? \Rightarrow En RAM !



Pile : registre **\$fp** / **\$sp**

- **\$fp** définit l'adresse du bloc mémoire pile de la fonction courante
- **\$sp** désigne l'adresse du sommet de la pile.

Définir un bloc de pile

- ➊ $\$fp \leftarrow \sp
- ➋ $\$sp \leftarrow \$sp - \text{TailleBloc}$
où **TailleBloc** est un multiple de 8 octets.

Récursion et Pile

Exemple de la somme en récursif

```
.data
.text
.globl main
main :    li $t0, 10                  # Nombre n
            jal sum
            move $a0, $v0
            jal print_int
            li $v0, 10
            syscall
sum :     beqz $t0, base_case      # Si n = 0 -> cas de base
            subi $sp, $sp, 8        # On réserve 2 mots en pile
            sw $ra, ($sp)          # On sauve l'adresse de retour sur la pile
            sw $t0, 4($sp)          # On sauve l'adresse et la valeur actuel de n
            subi $t0, $t0, 1        # On décremente n
            jal sum                # On relance l'appel récursif
            lw $t0, 4($sp)          # On charge la valeur de n sauvé
            add $v0, $v0, $t0       # On l'ajoute à la somme
            lw $ra, ($sp)          # On récupère l'adresse de retour
            addi $sp, $sp, 8        # On libère la pile
            jr $ra                 # On retourne à l'appelant
base_case : li $v0, 0
            jr $ra                 # On retourne à l'appelant

# Routine d'impression des entiers
print_int : li $v0, 1
            syscall
            jr $ra
```

Optimisations CPU

Une façon de mesurer les performances d'un processeur est le *MIPS* (*Million Instruction Per Second*). On utilise aussi le *temps d'exécution d'un programme* (*CPU time*) pour mesurer les performances d'un compilateur.

Pour simplifier un peu :

- On considère une architecture monocoeur.
- On considère une durée moyenne d'instruction, à savoir que certains types d'instructions peuvent se révéler plus coûteux, par exemple les opérations sur les flottants.

Dans le cas où de tels éléments doivent être considérés, il faut penser à pondérer les équations !

- Le MIPS n'est pas une mesure parfaite !
- Le CPU time d'un programme est bien plus fiable si l'on cherche à mesurer la performance d'un compilateur.

Optimisations CPU

Définition (Mesure de performance du processeur)

Soit un ensemble d'instructions \mathcal{I} correspondant à l'exécution d'un programme, le *MIPS* est donné par :

$$\text{MIPS} = \frac{\text{nb instructions}}{\text{CPU time} \times 10^6} = \frac{f}{\text{CPI} \times 10^6}$$

avec :

- CPU time = $\frac{\text{nb instructions} \times \text{CPI}}{f}$, le temps d'exécution du programme.
- CPI = $\sum_{i \in \mathcal{I}} p_i \times C_i$, (*Cycle Per Instruction*) moyen. Où :
 - p_i , la proportion d'instruction i au sein de \mathcal{I} . On a $\sum_{i \in \mathcal{I}} p_i = 1$
 - C_i , le nombre moyen de cycles horloge que dure l'instruction i .
- $f = \frac{1}{P}$ est la fréquence (*clock rate*) exprimé en *Hz* (et P , la période).

Optimisations CPU

Exemple

Soit une machine dont la période P d'un cycle horloge est de 0.45ns et \mathcal{I} , ensemble d'instructions correspondant à l'exécution d'un programme. On considère que $|\mathcal{I}| = n$. On donne la table correspondant à \mathcal{I} suivante :

Instruction i	Proportion p_i	Nb cycles C_i
Opérations UAL	0.35	1
Chargement	0.15	2
Stockage	0.1	2
Branchement	0.15	1
Décodage	0.25	2

- Calcul de la fréquence :
$$P = 0.45 \times 10^{-9} \text{s} \Leftrightarrow f = \frac{1}{0.45} \times 10^9 = 2,2\text{GHz}$$
- Calcul du CPI moyen :
$$\text{CPI} = 0,35 + 0,15 \times 2 + 0,1 \times 2 + 0,15 + 0,25 \times 2 = 1,5$$

- Calcul du MIPS : $\text{MIPS} = \frac{2,2 \times 10^9}{1,5 \times 10^6} \approx 1467$
- La performance du programme : $\text{CPU time} = 0,675n \times 10^{-9}$

Optimisations CPU

De ces équations, on en déduit que pour augmenter les performances d'un processeur, on peut :

- Diminuer le *CPI*,
 - Conception du processeur de façon à diminuer le temps mis pour l'exécution d'une instruction. Dépend de l'implémentation des circuits logiques et des logiques matérielles.
 - Adapter la fréquence d'utilisation des composants, le nombre de cycles et privilégier les instructions rapides. Dépend en grande partie de la conception du compilateur.
- Augmenter la fréquence *f*,
 - Utilisation de composants plus rapides, de transistors plus petits.
 - Augmenter la fréquence du bus FSB (Front Side Bus) pour agir sur celle du processeur : Overclocking.
- Paralléliser les instructions : *Pipeline* et *Multithread*.

Pourquoi le parallélisme ?

- Pendant longtemps, le facteur principal d'augmentation des performances des processeurs reposait sur l'augmentation de la fréquence d'horloge, rendue possible par l'amélioration de la finesse de gravure du silicium (*Loi de Moore*).
- Cependant, la telle finesse de gravure des circuits ($< 110nm$) pose le problème des courants de fuite, ce qui entraîne une chauffe accrue du matériel. Ceci a entraîné une stagnation pendant plusieurs années des fréquences maximales entre 3 et 4GHz.
- Les géants du processeur comme Intel ou IBM ont donc décidé de se tourner vers le parallélisme pour améliorer malgré tout les performances de leurs produits.

Définition (Pipeline)

Le *pipeline* est un mécanisme permettant d'accroître la vitesse d'exécution des programme dans un micro-processeur. L'idée générale est d'appliquer le principe du travail à la chaîne à l'exécution des instructions.

- Chaque tâche est décomposée en plusieurs instructions machine appelées *étages*.
- Dans un processeur doté de pipeline, chaque étage possède son propre circuit ce qui permet au processeur de paralléliser le traitement.
- Le processeur commence une nouvelle tâche avant d'avoir fini la précédente. Plusieurs instructions se trouvent donc simultanément en cours d'exécution au cœur du processeur.

Ainsi, on augmente le débit du processeur, c'est-à-dire le nombre d'instructions exécutées par unité de temps.

Optimisations CPU

La décomposition en étages suit le cycle d'exécution des instructions au sein du processeur.

Dans notre cas, et pour des raisons de simplicité, on considérera ici une décomposition \mathcal{E} en cinq étages telle que :

- *Instruction Fetch (IF)* : l'instruction suivante est placée dans le registre d'instruction depuis la mémoire ;
- *Instruction Decode (ID)* : les registres nécessaires sont lus ;
- *Execution (EXE)* : l'opération de calcul de l'instruction est effectuée (= l'ALU pour une opération arithmétique ou logique, calcul d'adresse mémoire, etc.) ;
- *Memory Access (MEM)* : si l'instruction concerne la mémoire (écriture/lecture), celle-ci est exécutée ;
- *Write Back (WB)* : les registres-cibles sont mis à jour.

Optimisations CPU

On suppose sur les schémas suivants que : $\forall e \in \mathcal{E}, C_e = 1$.

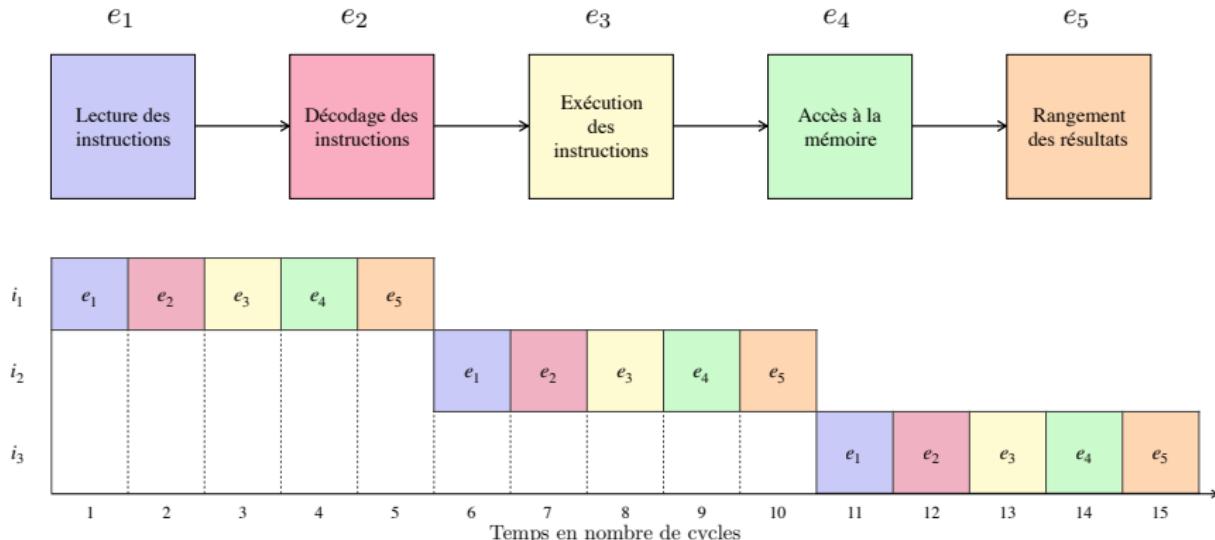


FIGURE 6 (a) - Diagramme d'exécution sans pipeline

Optimisations CPU

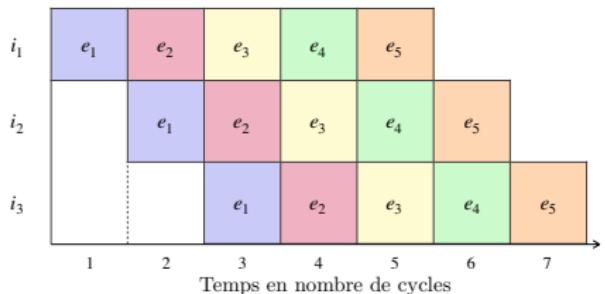


FIGURE 6 (b) - Diagramme d'exécution avec pipeline à 5 étages

- Chaque unité dispose de ressources matérielles propres,
- On peut donc exécuter plusieurs instructions en parallèle.
- On va ainsi recouvrir l'exécution des instructions,
 - Si le pipeline d'exécution contient p étage, une instruction dont l'exécution a commencé au cycle k se termine au cycle $k + p$.
 - L'instruction i_{k+1} commencera donc son exécution avant la fin effective de i_k !

Optimisations CPU

Pour exécuter n instructions impliquant p étages, il faut :

- $n \times p$ cycles horloge pour une exécution séquentielle normale.
- p cycles horloge pour la première instruction puis $n - 1$ cycles pour les $n - 1$ instructions restantes.

Dès lors, le gain $\gamma(n, p)$ en fonction du nombre d'instructions n et du nombre d'étages p est de :

$$\gamma(n, p) = \frac{n \times p}{p + n - 1}$$

On peut voir que $\lim_{n \rightarrow +\infty} \gamma(n, p) = p$.

Dès lors, l'idée est d'adopter une structure pipeline maximisant le nombre d'étages p .

⇒ Philosophie de l'Intel Pentium 4 Prescott® (2004) avec 31 étages!!

Optimisations CPU

Commencer l'instruction i_{n+1} avant la fin de i_n pose problème

- On ne respecte plus forcément la sémantique du programme original, car le résultat de i_{n+1} peut dépendre de i_n !
- On appelle ce type de situation des *aléas de pipeline*.
- Ces aléas réduisent les performances, car on n'exécute alors plus systématiquement une nouvelle instruction à chaque cycle

Pour garantir le respect de la sémantique, on sera parfois forcé de « geler» le pipeline et créer des *bulles* qui font perdre un cycle.

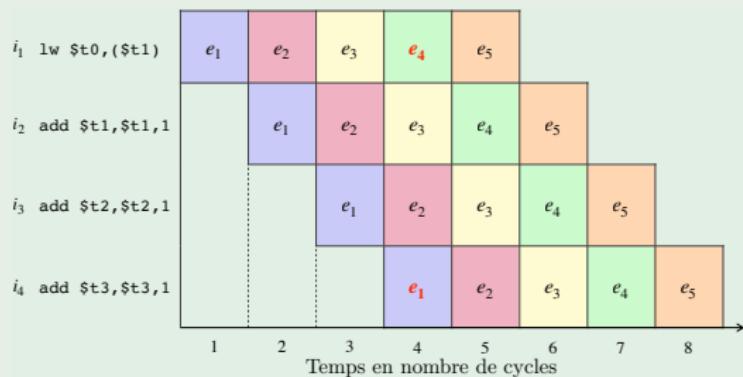
Aléas de pipeline

On distingue trois types d'*aléas* au sein des pipelines :

- Aléa *structurel* : Deux instructions dans des étages différents du pipeline nécessitent la même ressource.
⇒ Utilisation de mémoires distinctes entre données et programme.
- Aléa de *données* : Une instruction nécessite une donnée qui n'a pas encore été calculée par une instruction précédente.
⇒ Technique de *forwarding* permettant à l'aide des chemins de données d'obtenir les résultats de l'instruction i dès la fin de l'étape e_3 pour les opérations UAL et e_4 pour les transferts.
- Aléa de *branchement* : Une instruction peut-être évaluée avant même de connaître l'emplacement effectif du saut (càd de la prochaine valeur inscrite au compteur ordinal).
⇒ Utilisation des *branchements prédictifs*.

Optimisations CPU

Exemple (Aléa structurel)

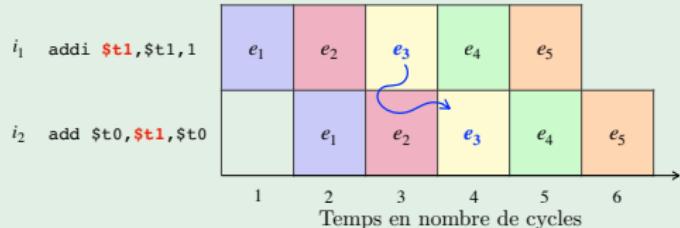
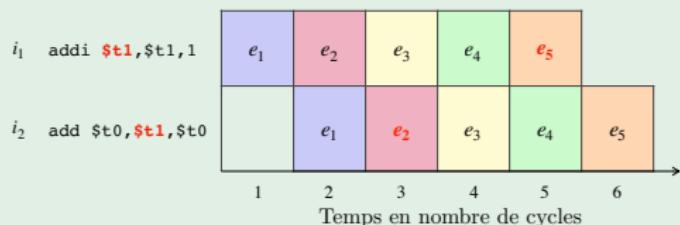


- Au cycle 4, l'instruction i_1 demande une lecture mémoire et l'instruction i_4 de charger une instruction en même temps. Ces deux étapes nécessitent simultanément l'accès à la mémoire
- Il s'agit d'un *aléa structurel*. Comme cela est impossible, l'instruction i_4 et celles qui suivent devront être retardées d'un cycle.

Optimisations CPU

Exemple (Aléa de données)

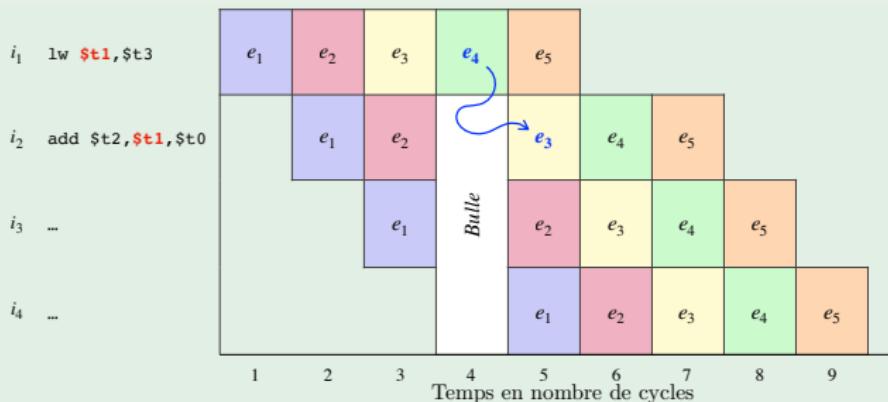
- Le résultat de i_1 est écrit dans le registre $\$t1$ après la lecture de ce même registre par i_2 . La valeur utilisée par i_2 est alors erronée.
- Pourtant, le résultat de i_1 est disponible dès la fin de l'étape d'exécution e_3 de celle-ci et il est utilisé seulement à l'étape e_3 de i_2 .
⇒ On fournit le résultat de i_1 en entrée de l'additionneur à la place de la valeur lue dans $\$t1$ par i_2 : Réalisé automatiquement par le compilateur en usant du *forwarding*.



Optimisations CPU

Exemple (Aléa de données)

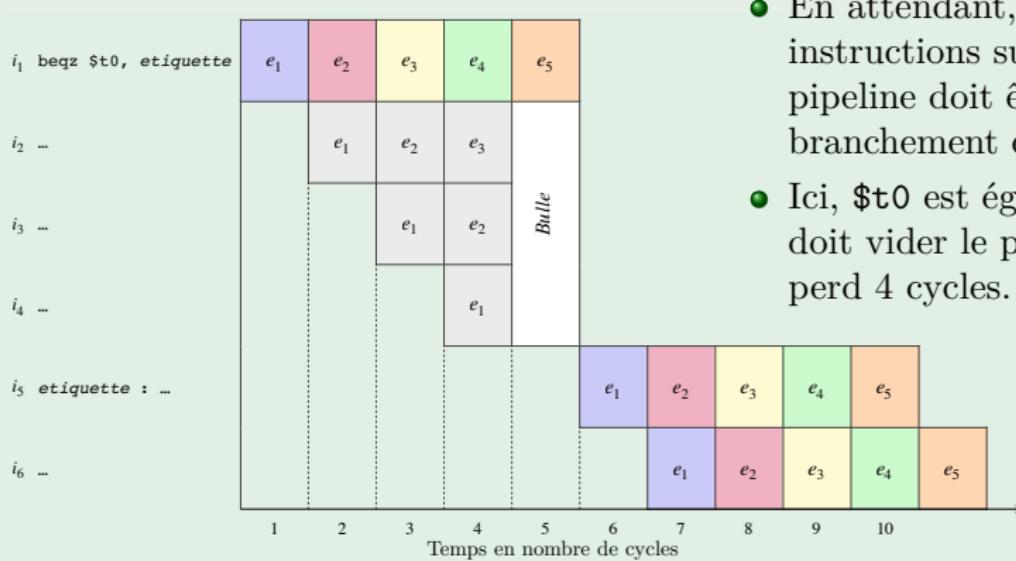
- Ici, i_1 est une instruction de transfert ; le résultat n'est pas disponible avant l'étape e_4 . Comme cette étape a lieu après l'étape e_3 de i_2 , le forwarding ne suffit pas pour faire disparaître l'aléa.
- Il faut retarder i_2 . On introduit une *bulle* dans le pipeline.



Optimisations CPU

Exemple (Aléa de branchement)

- Lors d'un branchement, l'adresse de la prochaine instruction effective est calculée à l'étape e_3 et rangée dans le compteur ordinal à l'étape e_5 .



- En attendant, les instructions suivent et le pipeline doit être vidé si le branchement est effectif.
- Ici, $$t0$ est égale à 0, on doit vider le pipeline. On perd 4 cycles.

Conclusions

Pour terminer cette partie, assez longue, bien que loin d'être exhaustive. J'aimerais revenir sur chaque point abordé et dégager les notions importantes.

- Un ordinateur, c'est une machine qui traite l'information à l'aide d'une *suite d'instructions* que l'on appelle *programme*.
 - Un programme est écrit dans un langage formel puis compilé ou/et interprété.
 - Le *compilateur* est un élément important dans l'optimisation aujourd'hui : Optimisation du code, ordonnancement des instructions, etc.
 - On s'attarde ici sur le résultat de la compilation : les langages d'assemblage.
- Les machines actuelles utilisent le modèle d'architecture dit de *Von Neumann* (plus ou moins évoluée et optimisée).
 - On a une *unité de contrôle*, une *unité arithmétique et logique*, une *mémoire*, des dispositifs d'*E/S* et des *bus* chargés de la communication.
- Un processeur possède un *jeu d'instructions* qui définit les opérations supportées, les registres manipulables et les différents modes d'adressage.
 - On oppose souvent les jeux d'instructions *CISC* et *RISC*.

Conclusions

- On a vu les rudiments de l'assembleur MIPS.
- En Assembleur, les programmes deviennent plats.
 - On doit les mettre sous forme linéaire pour bien programmer.
 - On programme à l'aide des instructions `go to` qui sont des sauts et branchements.
- On ne possède pas de variables mais uniquement des registres.
 - On a 32 registres, mais on ne peut pas tous les utiliser comme veux
 - Gérer implicitement par le compilateur et les conventions d'architecture.
- Pour faire de la récursivité, on utilise le segment de la pile en RAM.
 - On enregistre les adresses au sein de la pile pour pouvoir les sauvegarder.
 - On les dépile par la suite.
- Pour optimiser le processeur, on exécute des tâches en parallèle via des pipelines
 - Gain fort de performances mais pose des aléas dûs à la dépendances des instructions
 - Base du parallélisme et des optimisations des processeurs modernes.