

LINFO1252 - LSINC1252

Systèmes Informatiques



Leçon 4 : structure des ordinateurs

Pr. Etienne Rivière

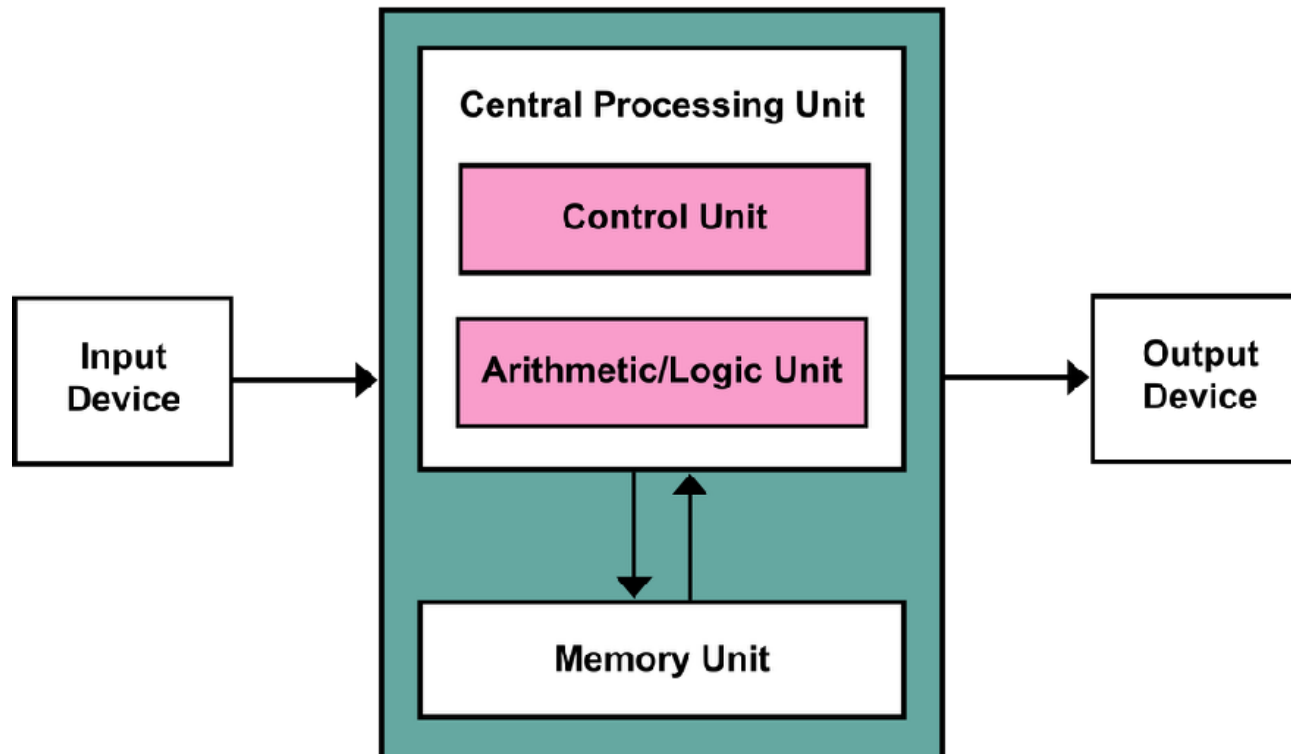
etienne.riviere@uclouvain.be

Objectif du cours

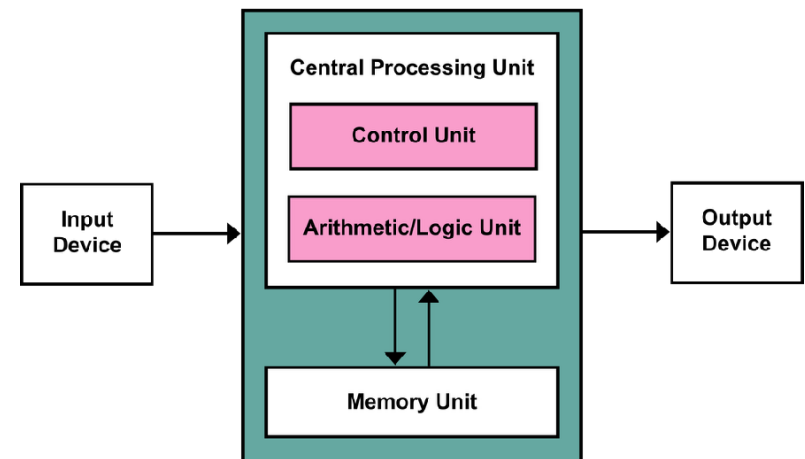
- Approfondir les principes d'organisation d'un système informatique
- Comprendre le rôle et l'utilisation de la hiérarchie mémoire et le principe de cache
- Illustrer le jeu d'instruction IA-32

Architecture de von Neumann (rappel)

- Mémoire principale utilisée pour stocker les instructions ET les données

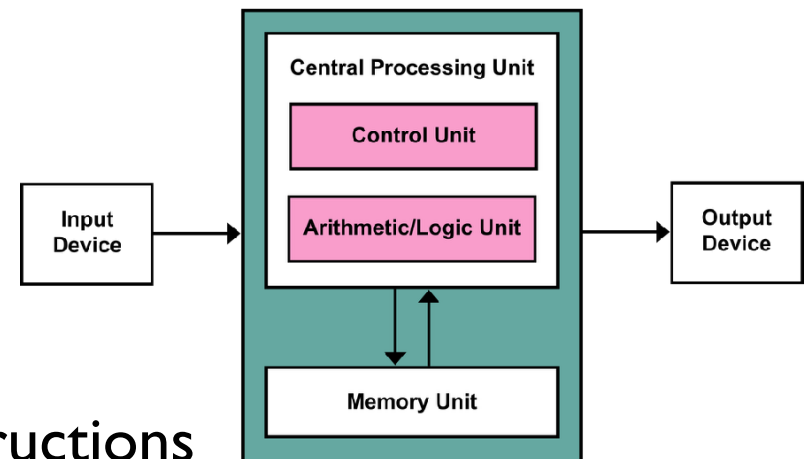


Composants de l'architecture



- Processeur (CPU)
 - Unité arithmétique et logique
 - Opérateurs matériels mettant en œuvre les opérations arithmétiques (+, −, ...) et logiques (&, |, ^, ...)
 - Unité de commande
 - Met en œuvre le cycle fetch/decode/execute des instructions depuis la mémoire

Composants de l'architecture



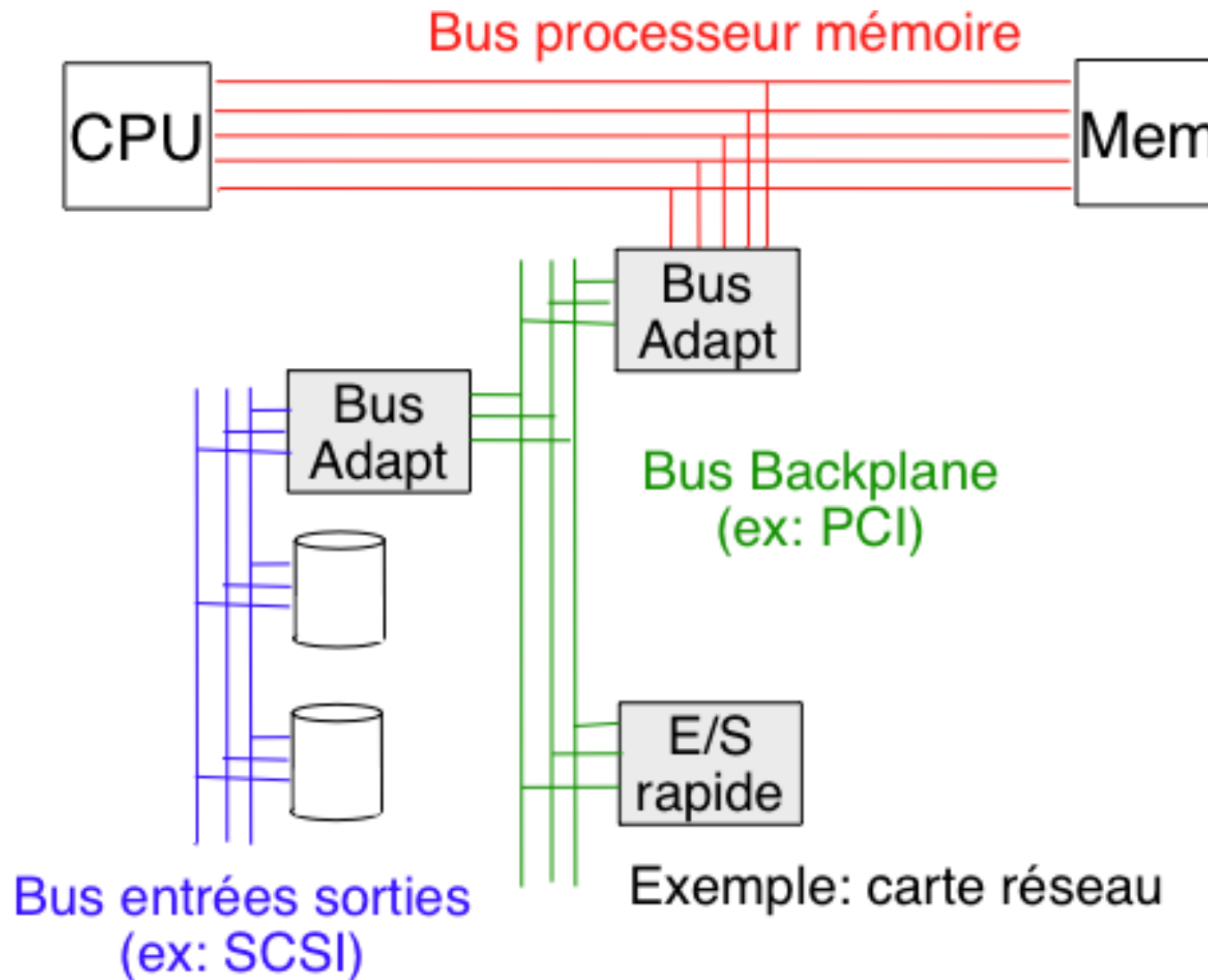
● Mémoire

- Stocke à la fois les données et les instructions
- Organisée en octets d'adresses consécutives
- 2^k octets : adresse sur au moins k bits
- Typiquement, $k = 32$ ou 64 bits
- Un processeur qui utilise des adresses sur k bits peut adresser au plus 2^k octets de mémoire physique
 - 32 bits : $2^{32} = 4 \times 2^{10} \times 2^{10} \times 2^{10} = 4$ Giga-octets de mémoire max. Combien sur votre PC personnel ?
 - 64 bits : $2^{64} = 16 \times (2^{10})^6 = 16$ Exa-octets (milliards de milliards)

Instructions et registres

- Les instructions sont codées en binaire
 - Numéro de l'instruction dans le jeu
 - Opérandes
 - Instructions de taille fixe ou variable (IA32)
 - Pas très important pour le programmeur, important pour le concepteur de processeur
- Un processeur dispose d'un nombre limité (quelques dizaines) de zones mémoire : les registres
 - Directement accessibles par les instructions
 - Utilisés en combinaison avec la mémoire principale

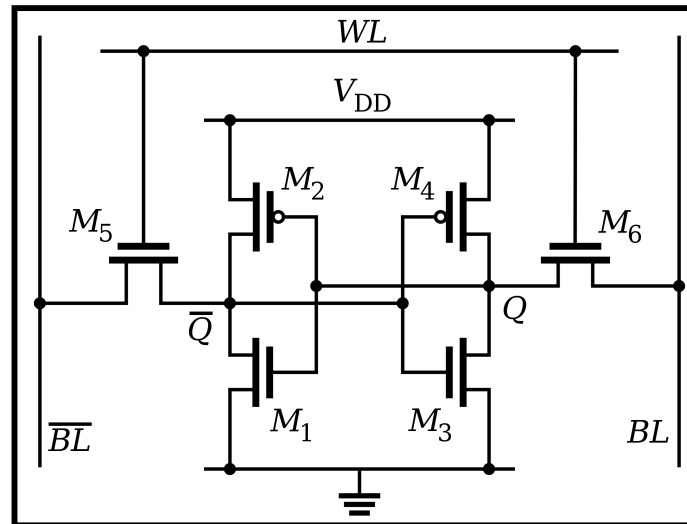
Architecture “traditionnelle”



Technologies mémoire

- Deux grands types de technologies co-existent pour mettre en œuvre de manière électronique une mémoire
 - Mémoire statique : SRAM
 - Mémoire dynamique : DRAM
- Ces deux technologies sont des mémoires dites *volatiles* : leur contenu est perdu lorsque l'alimentation électrique est coupée
 - La technologie utilisée pour votre disque SSD est différente : Les données ne sont (heureusement) pas perdue lorsque votre ordinateur est hors-tension

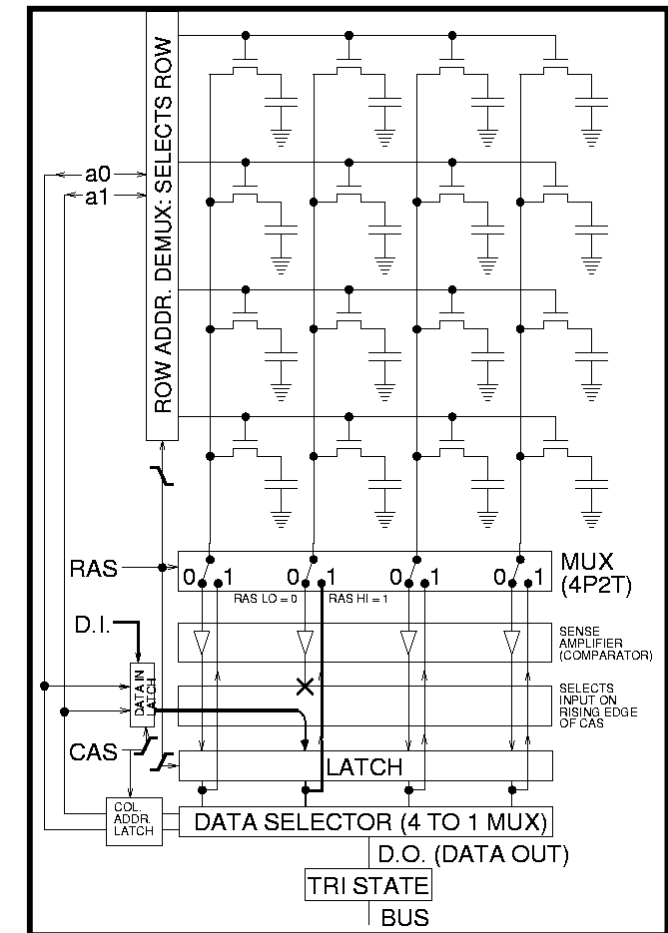
Mémoire statique (SRAM)



- Chaque *bit* d'information est stocké par un circuit électronique de type *bascule*
 - 6 transistors par bit (ou 4 + des résistances)
- Le bit peut être lu en testant le passage (ou non) du courant par la ligne de lecture ; et être mis à 0 ou 1 par les lignes de contrôle
- Valeur reste stockée tant que chip alimenté (e.g. $V_{dd}=5\text{ V}$)

Mémoire dynamique (DRAM)











- Chaque *bit* d'information est stocké par la présence d'une charge dans un condensateur
- Charge d'un condensateur diminue avec le temps
 - Technologie (isolation) imparfaite
 - “Courants de fuite”
- Il faut ‘rafraîchir’ périodiquement la mémoire en balayant l'ensemble des bits de la mémoire et en rechargeant les condensateurs pour les bits à 1
 - D'où l'appellation de mémoire *dynamique*
 - Rafraîchissement aussi nécessaire après chaque lecture
- Lecture ou écriture : sélection de la ligne, sélection de la colonne, lecture
 - Plus efficace pour des données contigües que pour des données réparties aléatoirement !



Technologies de DRAM

Technologie	Fréquence [MHz]	Débit [MB/sec]
SDRAM	200	1064
RDRAM	400	1600
DDR-1	266	2656
DDR-2	333	5328
DDR-2	400	6400
DDR-3	400-667	6400-10600
DDR-4	800-1600	12800-25600

SRAM vs DRAM

Critère	SRAM	DRAM	Année	Accès SRAM	Accès DRAM
Latence			1980	300 ns	375 ns
			1985	150 ns	200 ns
Bande passante			1990	35 ns	100 ns
			1995	15 ns	70 ns
Densité			2000	3 ns	60 ns
			2005	2 ns	50 ns
Consommation d'énergie			2010	1.5 ns	40 ns
Prix par Mo					

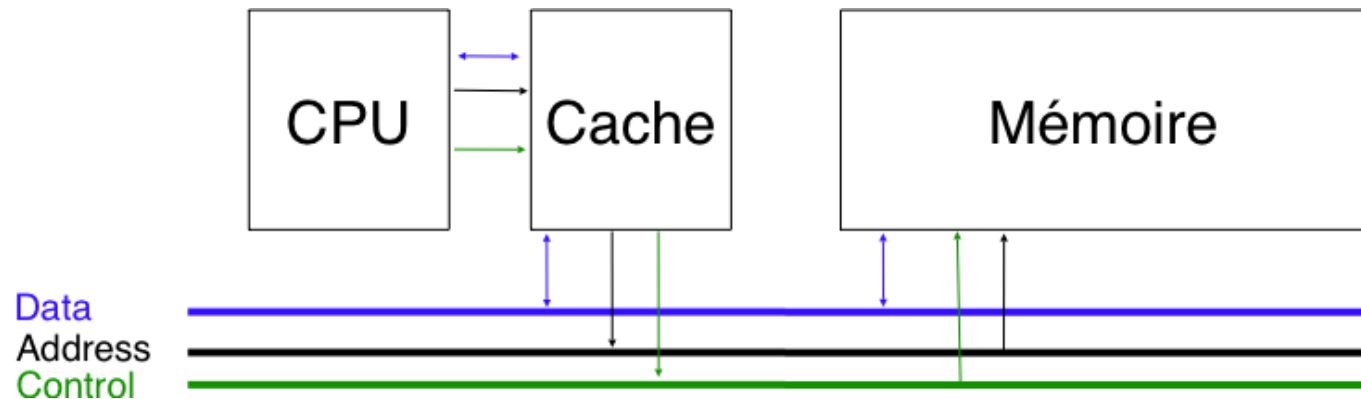
Evolution des performances entre processeur et mémoire

Année	Processeur	Cycle horloge processeur	Latence mémoire SRAM	Latence mémoire DRAM
1980	Intel 8080	1000 ns	300 ns	375 ns
1990	Intel 80386	50 ns	35 ns	100 ns
2000	Pentium III	1.6 ns	3 ns	60 ns

Combiner SRAM et DRAM ?

- Idée : combiner une petite quantité de SRAM (chère) et une grande quantité de DRAM
- Gestion explicite (mémoire séparée en zones, réservation en SRAM et DRAM) ?
 - Complexifie l'écriture des applications
 - Comment choisir quelles données placer et où ?
- Mieux : utiliser la SRAM pour mettre en œuvre le principe de mémoire cache

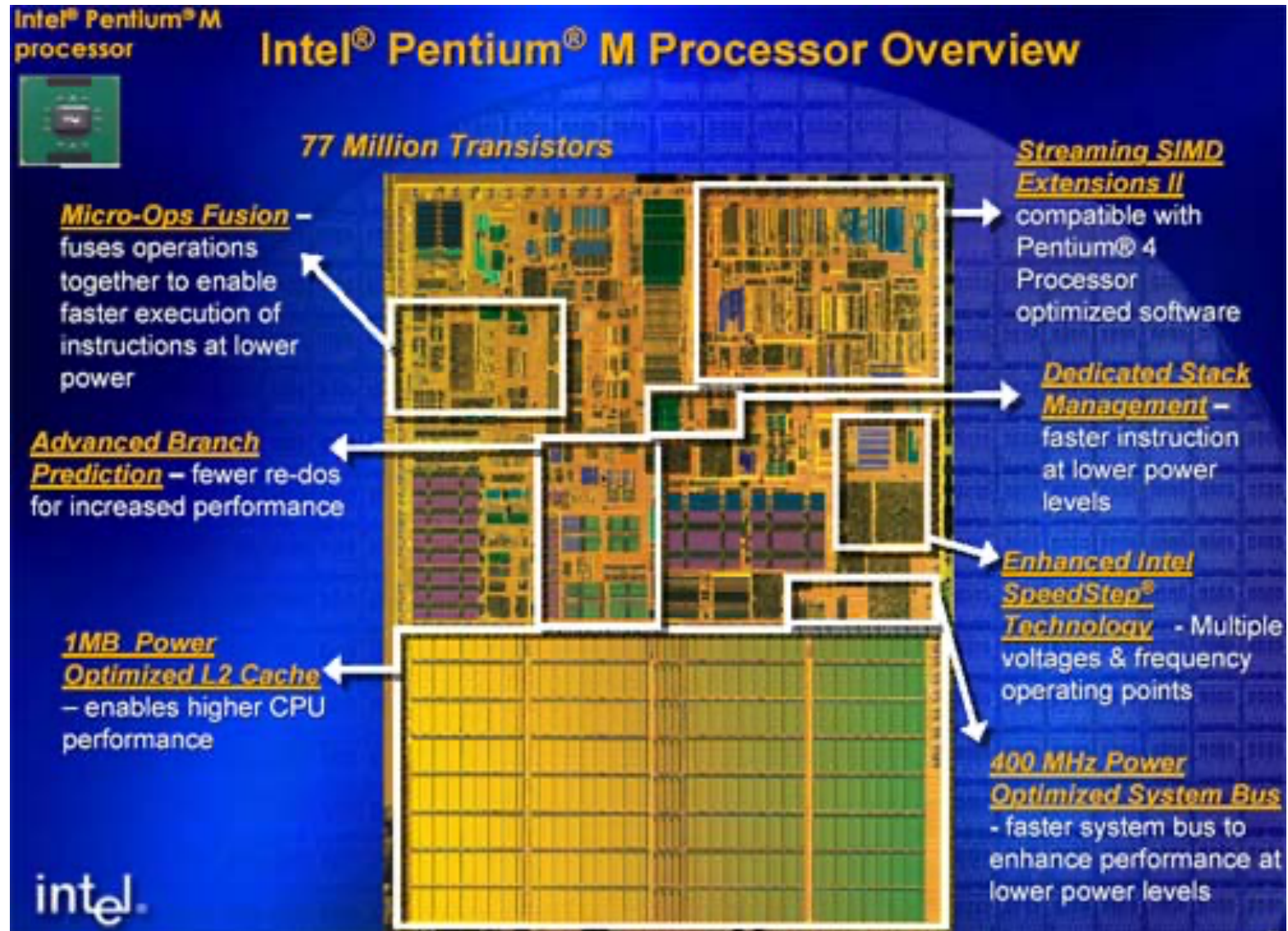
Mémoire cache



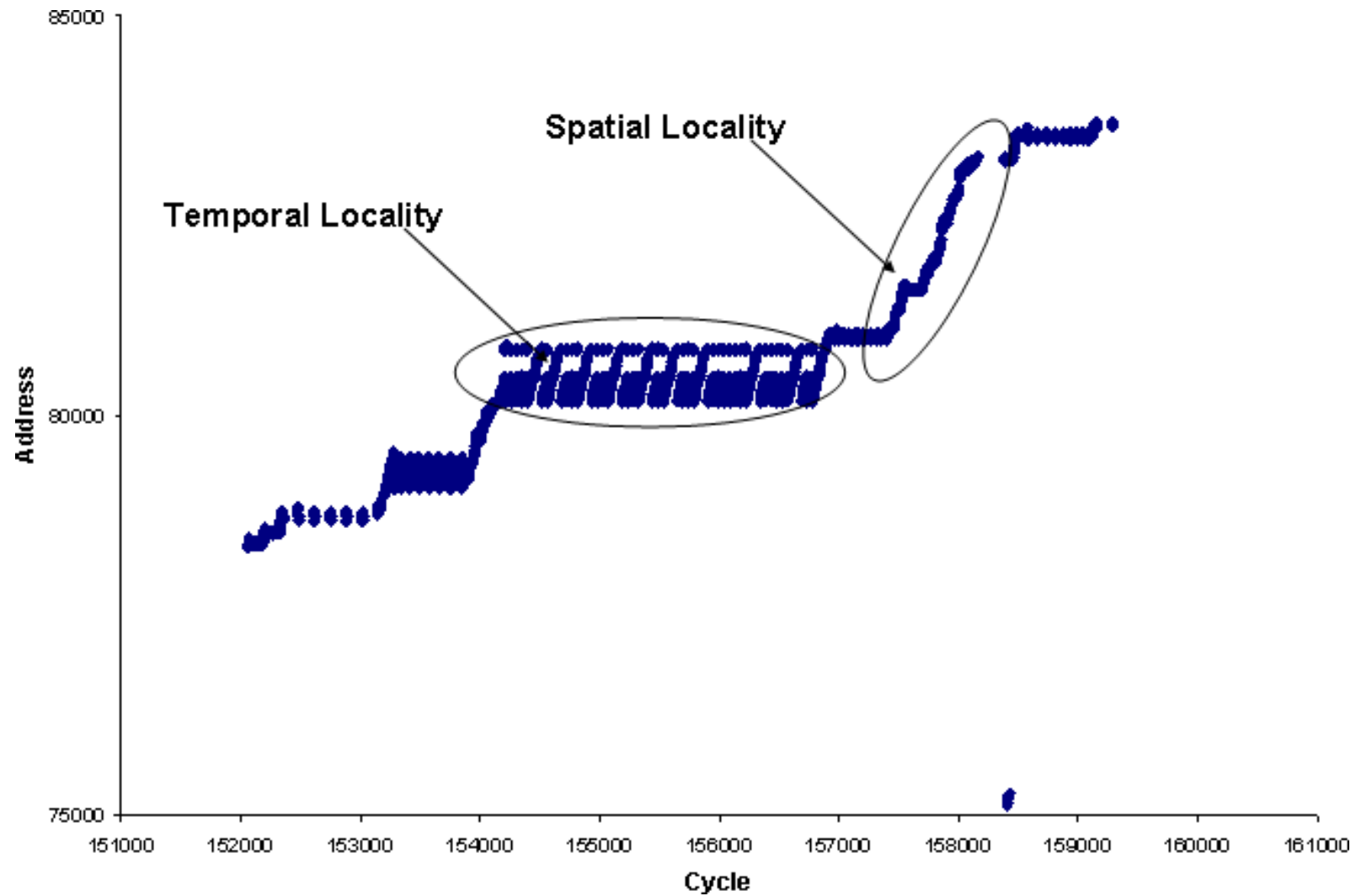
- Intermédiaire entre le processeur et la mémoire principale
 - Faible capacité mais rapide
 - Mémoire de technologie SRAM sur la même puce que le processeur
- Conserve les données récemment accédées
- Granularité = ligne de cache (par exemple 64 octets)

```
cat /proc/cpuinfo | grep cache_alignment | tail -n 1  
cache_alignment : 64
```

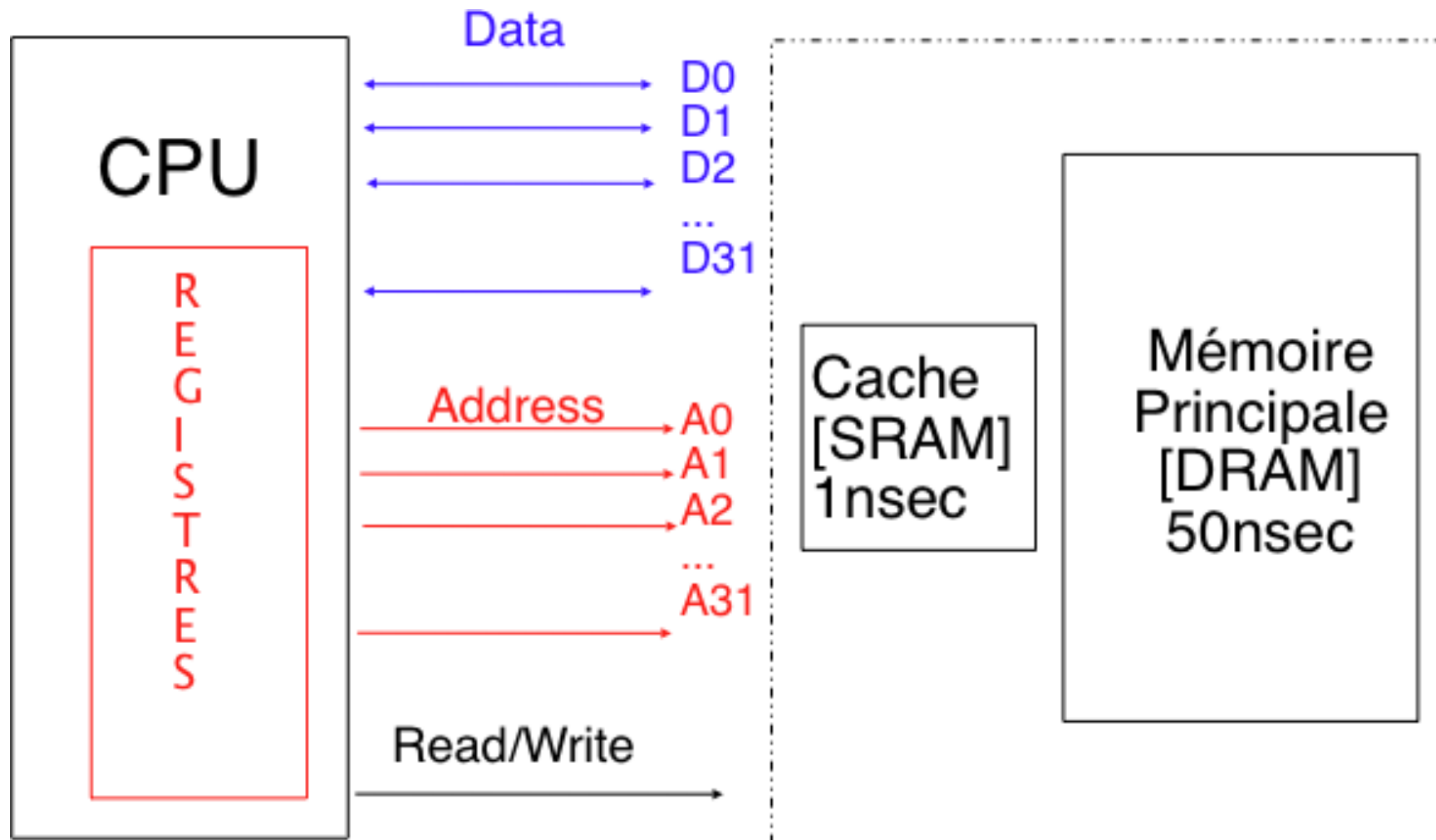
Exemple : Centrino (Pentium-M)



Localités mémoire



Hiérarchie mémoire



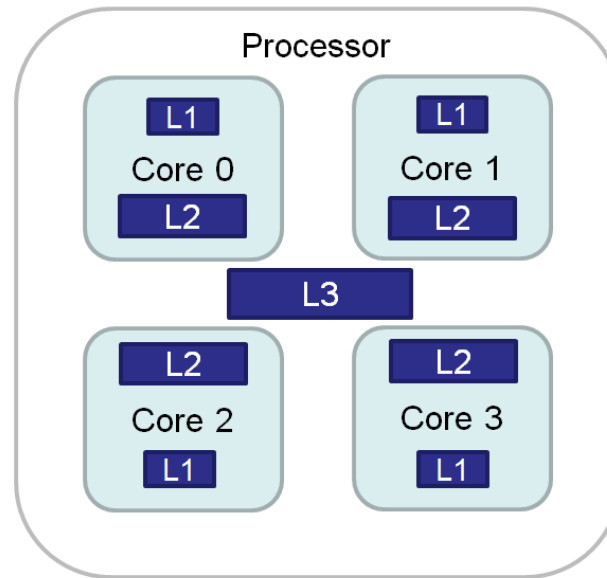
Fonctionnement d'un cache : lecture

- Lecture de l'octet à l'adresse x
- Si le cache contient une copie de l'adresse x :
on lit depuis le cache
- Sinon
 - Le gestionnaire de mémoire cache (hardware)
récupère une copie de la mémoire à l'adresse x
 - Mais pas seulement — on lit une ligne de cache
(par ex. 64 octets) à la fois : localité temporelle et
compatibilité avec les contraintes DRAM
 - La lecture est ensuite effectuée depuis le cache

Fonctionnement d'un cache : écriture

- Écriture de la valeur A à l'adresse x
- Si le cache contient une copie de l'adresse x :
on écrit dans le cache, sinon on récupère
d'abord la ligne de cache correspondante
- Comment propager les modifications ?
 - *Write through* : immédiatement
 - Mais pourquoi payer le prix d'une latence DRAM ?
 - *Write back* : plus tard, lorsque la ligne de cache
sera évincée du cache pour faire de la place

En réalité ...



- Les processeurs modernes sont en réalité des multi-processeurs (multi-cœurs)
- Plusieurs niveau de caches : différents compromis coût / taille / performance ; caches de plus bas niveau séparé entre données et instructions
 - Les principes restent les mêmes !
- En Master :
 - Architecture and performance of computer systems (LINFO2241)
 - Multicore Programming (LINFO2355)

Conclusion intermédiaire

- Un système informatique moderne étend les principes de l'architecture de Von Neumann sans en remettre en cause le principe fondamental
- Composer avec la technologie : compromis coût/performance
 - Principe de cache, important en système (et s'applique aussi dans les systèmes répartis comme Internet !)

Étude de cas : jeu d'instruction IA 32

Objectifs

- Illustrer un jeu d'instruction avec un exemple concret
- IA32 : adresses sur 32 bits (max 4 Go de mémoire physique), registres de 32 bits
 - Nous ne verrons pas l'extension IA32-64
 - Une adresse (un pointeur) et une valeur `int` ont la même taille
- Processeurs : Intel 80386 (1985), Intel 80486 (1989), Intel Pentium (1993), et les équivalents chez d'autres fabricants

Registres

- 8 registres génériques, chacun de 32 bits



utilisés pour la gestion de la pile

- 1 registre stockant le *compteur de programme*

EIP

- + des registres pour traiter les *float* et *double*

Types de données

Type	Taille (bytes)	Suffixe assembleur
<code>char</code>	1	b
<code>short</code>	2	w
<code>int</code>	4	l
<code>long int</code>	4	l
<code>void *</code>	4	l

- Les instructions sont suffixées avec le type de données sur lesquelles elles opèrent

Instruction mov

- Déplacer des données depuis/vers des registres et depuis/vers la mémoire
- Deux arguments :

```
mov src, dest ; déplacement de src vers dest
```

- Une version de mov pour chaque type de donnée : movb, movw, movl
 - Et donc 3 codes instructions différents

Modes d'adressage

- Comment spécifier les arguments
- Premier mode : *registre*
 - Nom du registre préfixé par %
 - %eax, %ebx, etc.

```
movl %eax, %ebx ; déplacement de %eax vers %ebx  
movl %ecx, %ecx ; aucun effet
```

- Second mode : *immédiat*
 - Seulement pour argument source
 - Placer une constante préfixée par \$ dans un registre :

```
movl $0, %eax ; initialisation de %eax à 0  
movl $1252, %ecx ; initialisation de %ecx à 1252
```

3ème mode : *absolu*

- Argument = adresse en mémoire
 - Source ou destination, mais pas les deux (choix pour simplifier la mise en œuvre du processeur)
 - Valeur sans préfixe (ne pas confondre avec `$valeur :immédiat`)

Adresse	Valeur
0x10	0x04
0x0C	0x10
0x08	0xFF
0x04	0x00
0x00	0x04

```

movl 0x04, %eax    ; place la valeur 0x00 (qui se trouve à l'adresse 0x04) dans %eax
movl $1252, %ecx   ; initialisation de %ecx à 1252
movl %ecx, 0x08    ; remplace 0xFF par le contenu de %ecx (1252) à l'adresse 0x08
  
```

4ème mode : *indirect*

- Plutôt que de spécifier une adresse directement, on spécifie un registre contenant une adresse
 - Similaire à l'utilisation d'un pointeur
 - Parenthèses autour du registre source ou destination

Adresse	Valeur
0x10	0x04
0x0C	0x10
0x08	0xFF
0x04	0x00
0x00	0x04

```

movl $0x08, %eax    ; place la valeur 0x08 dans %eax
movl (%eax), %ecx    ; place la valeur se trouvant à l'adresse qui est
                    ; dans %eax dans le registre %ecx : %ecx=0xFF
movl 0x10, %eax      ; place la valeur se trouvant à l'adresse 0x10 dans %eax
movl %ecx, (%eax)    ; place le contenu de %ecx, c'est-à-dire 0xFF à l'adresse
                    ; qui est contenue dans %eax (0x04)
  
```

5ème mode : *indirect avec base et déplacement*

- Extension du mode *indirect*
- Permet d'accéder à une adresse + ou - une valeur donnée
 - Utile pour la manipulation de la pile
 - D(%reg) avec
 - D le déplacement (positif ou négatif)
 - reg le nom du registre

Adresse	Valeur
0x10	0x04
0x0C	0x10
0x08	0xFF
0x04	0x00
0x00	0x04

```

movl $0x08, %eax    ; place la valeur 0x08 dans %eax
movl 0(%eax), %ecx  ; place la valeur (0xFF) se trouvant à l'adresse
                    ; 0x08= (0x08+0) dans le registre %ecx
movl 4(%eax), %ecx   ; place la valeur (0x10) se trouvant à l'adresse
                    ; 0x0C (0x08+4) dans le registre %ecx
movl -8(%eax), %ecx  ; place la valeur (0x04) se trouvant à l'adresse
                    ; 0x00 (0x08-8) dans le registre %ecx
  
```

Instructions arithmétiques et logiques

- Comme pour mov, une variante par taille de données à manipuler (b, w ou l)
- Instructions à un seul argument
 - inc : incrémente la valeur
 - (e.g. compteur de boucle)
 - dec : décrémente
 - not : opération logique “non” bit-à-bit

Instructions à deux arguments

- `add` : addition, `dest + src => dest`
- `sub` : soustraction, `dest - src => dest`
- `mul` : multiplication de nombres entiers non-signés
 - `imul` pour les nombres signés
 - attention aux dépassements de la plage de valeurs ! (non couvert par ce cours)
- `div` : division de nombre entiers non-signés
- `shl` / `shr` : “shift”, décalage logique vers la gauche ou la droite (plus rapide pour multiplier/diviser par multiple de 2!)
- `or` / `xor` / `and` : opérations binaires bit à bit, résultat dans `dest`

Illustration (I)

Adresse	Variable	Valeur
0x0C	c	0x00
0x08	b	0xFF
0x04	a	0x2
0x00	•	0x1

```

movl 0x04, %eax    ; %eax=a
addl $1, %eax      ; %eax++
movl %eax, 0x04    ; a=%eax

```

$a = a + 1$

```

movl 0x08, %eax    ; %eax=b
subl 0x0c, %eax    ; %eax=b-c
movl %eax, 0x04    ; a=%eax

```

$a = b - c$

Illustration (2)

```
int j,k,g,l;
// ...
l=g^j;
j=j|k;
g=l<<6;
```

```
movl    g, %eax    ; %eax=g
xorl    j, %eax    ; %eax=g^j
movl    %eax, l    ; l=%eax
movl    j, %eax    ; %eax=j
orl     k, %eax    ; %eax=j|k
movl    %eax, j    ; j=%eax
movl    l, %eax    ; %eax=l
shll    $6, %eax   ; %eax=%eax << 6
movl    %eax, g    ; g=%eax
```

Registre de drapeaux eflags

- Registre spécial : eflags contient des bits “drapeau” mis à jour lors de l’exécution des instructions (par exemple, addl)
 - **ZF** (Zero Flag) : ce drapeau indique si le résultat de la dernière opération était zéro
 - **SF** (Sign Flag) : indique si le résultat de la dernière instruction était négatif (interprété en comp. à 2)
 - **CF** (Carry Flag) : indique si le résultat de la dernière instruction arithmétique non signée nécessitait plus de 32 bits pour être stocké
 - **OF** (Overflow Flag) : indique si le résultat de la dernière instruction arithmétique signée a provoqué un dépassement de capacité

Instructions de comparaison

- Registre `eflags` mis à jour après une instruction
- Aussi, instructions de comparaison explicite :
 - `cmp` : équivalent de `sub` mais sans stocker le résultat
 - `test` : équivalent de `and`, aussi sans stocker le résultat
- Instructions `set` pour récupérer les valeurs des drapeaux dans un registre (uniquement octet de poids faible)
 - `sete` récupère drapeau `ZF` — égalité
 - `sets` récupère drapeau `SF`
 - `setg` récupère ($\sim SF$ & $\sim ZF$) en gérant dépassement — test $>$
 - `setl` récupère drapeau `SF` en gérant dépassement — test \leq

Illustration

```
r=(h>1);
r=(j==0);
r=g<=h;
r=(j==h);
```

```

cmpl    $1, h           ; comparaison
setg    %al              ; %al est le byte de poids faible de %eax
movzbl  %al, %ecx        ; copie le byte dans %ecx
movl    %ecx, r          ; sauvegarde du résultat dans r

cmpl    $0, j           ; comparaison
sete    %al              ; fixe le byte de poids faible de %eax
movzbl  %al, %ecx        ; sauvegarde du résultat dans r

movl    g, %ecx          ; comparaison entre g et h
cmpl    h, %ecx          ; fixe le byte de poids faible de %eax
setl    %al
movzbl  %al, %ecx
movl    %ecx, r

movl    j, %ecx          ; comparaison entre j et h
cmpl    h, %ecx
sete    %al
movzbl  %al, %ecx
movl    %ecx, r

```

Instructions de saut

- Permettent de modifier la valeur du compteur de programme `%eip`
 - Modifie l'ordre d'exécution des instructions
 - Mise en œuvre des structures de contrôle : conditionnelles, boucles, etc.
 - Prend comme argument une adresse ; comportement peut dépendre du contenu du registre `eflags`
- Construction similaire en C avec `goto`

goto en C

```
int v=0;
for(int i=0;i<SIZE;i++)
    for(int j=0;j<SIZE;j++) {
        if(m[i][j]>MVAL) {
            v=m[i][j];
            goto suite;
        }
    }
printf("aucune valeur supérieure à %d\n",MVAL,v);
goto fin;
suite:
printf("première valeur supérieure à %d : %d\n",MVAL,v);
fin:
return(EXIT_SUCCESS);
```

L'utilisation de goto est découragée
dans un langage de haut niveau !

... mais la seule possibilité en assembleur

Instructions de saut, suite

- Saut inconditionnel : `jmp [etiquette]`
 - le compteur de programme est positionné à l'adresse de la première instruction suivant `[etiquette]`
 - Aussi possible d'utiliser le contenu d'un registre : `jmp *%eax` indique que le programme doit continuer à l'adresse contenue dans `%eax`
- Saut conditionnels :
 - `je` : saut si égal (teste le drapeau ZF) (inverse : `jne`)
 - `js` : saut si négatif (teste le drapeau SF) (inverse : `jns`)
 - `jg` : saut si strictement supérieur (teste les drapeaux SF et ZF et prend en compte un overflow éventuel) (inverse : `jle`)
 - `jge` : saut si supérieur ou égal (teste le drapeaux SF et prend en compte un overflow éventuel) (inverse : `jle`)

Exemple : conditionnelles

```

if(j==0)
    r=1;

if(j>g)
    r=2;
else
    r=3;

if (j>=g)
    r=4;

```

```

if(j!=0) { goto diff; }
    r=1;
diff:
    // suite
if(j<=g) { goto else; }
    r=2;
    goto fin;
else:
    r=3;
fin:
    // suite
if (j<g) { goto suivant; }
    r=4;

suivant:

```

```

    cmpl    $0, j        ; j==0 ?
    jne     .LBB2_2      ; jump si j!=0
    movl    $1, r        ; r=1
.LBB2_2:

    movl    j, %eax      ; %eax=j
    cmpl    g, %eax      ; j<=g ?
    jle     .LBB2_4      ; jump si j<=g

    movl    $2, r        ; r=2
    jmp     .LBB2_5      ; jump fin expression
.LBB2_4:
    movl    $3, r        ; r=3
.LBB2_5:

    movl    j, %eax      ; %eax=j
    cmpl    g, %eax      ; j<g ?
    jl      .LBB2_7      ; jump si j<g
    movl    $4, r        ; r=4
.LBB2_7:

```

Exemple : boucle while

```
while(j>0)
{
    j=j-3;
}
```

```
debut:
    if(j<=0) { goto fin; }
    j=j-3;
    goto debut;
fin:
```

```
.LBB3_1:
    cmpl    $0, j      ; j<=0
    jle     .LBB3_3    ; jump si j<=0
    movl    j, %eax
    subl    $3, %eax
    movl    %eax, j    ; j=j-3
    jmp     .LBB3_1
.LBB3_3:
```

Exemple : boucle for

```
for(j=0;j<10;j++)
{ g=g+h; }
```

```
for(j=9;j>0;j=j-1)
{ g=g-h; }
```

```

    movl    $0, j      ; j=0
.LBB4_1:
    cmpl    $10, j
    jge     .LBB4_4     ; jump si j>=10
    movl    g, %eax     ; %eax=g
    addl    h, %eax     ; %eax+=h
    movl    %eax, g     ; g=%eax
    movl    j, %eax     ; %eax=j
    addl    $1, %eax    ; %eax++
    movl    %eax, j     ; j=%eax
    jmp     .LBB4_1
.LBB4_4:
    movl    $9, j      ; j=9
.LBB4_5:
    cmpl    $0, j
    jle     .LBB4_8     ; jump si j<=0
    movl    g, %eax
    subl    h, %eax
    movl    %eax, g
    movl    j, %eax     ; %eax=j
    subl    $1, %eax    ; %eax--
    movl    %eax, j     ; j=%eax
    jmp     .LBB4_5
.LBB4_8:
```

Manipulation de la pile

- Le sommet de la pile est stocké dans le registre `%esp`
- Instructions spéciales pour ajouter/retirer de l'information au sommet de la pile
 - `pushl %reg` : place le contenu de `%reg` au sommet de la pile, puis décrémente le registre `%esp` de 4 unités
 - `popl %reg` : lit le mot de 32 bits du sommet de la pile et le place dans `%reg`, puis incrémente `%esp` de 4
 - Comme vu la semaine passée : la valeur n'est pas effacée
 - équivalentes à :

```
subl $4, %esp      ; ajoute un bloc de 32 bits au sommet de la pile
movl %ebx, (%esp)   ; sauvegarde le contenu de %ebx au sommet
```

```
movl (%esp), %ecx   ; sauve dans %ecx la donnée au sommet de la pile
addl $4, %esp       ; déplace le sommet de la pile de 4 unités vers le haut
```

Exemple de manipulation de la pile

Adresse	Valeur	Registre	
0x10	0x04	%esp	0x08
0x0C	0x00	%eax	0x02
0x08	0x00	%ebx	0xFF
0x04	0x00	valeurs initiales registres	
0x00	0x00		

contenu de la mémoire initial

```

push %eax ; %esp contient 0x08 et M[0x08]=0x02
push %ebx ; %esp contient 0x04 et M[0x04]=0xFF
pop %eax  ; %esp contient 0x08 et %eax 0xFF
pop %ebx  ; %esp contient 0x0C et %ebx 0x02
pop %eax  ; %esp contient 0x10 et %eax 0x00
  
```

Adresse	Valeur	Registre	
0x10	0x04	%esp	0x10
0x0C	0x00	%eax	0x00
0x08	0x02	%ebx	0x02
0x04	0xFF	valeurs finales registres	
0x00	0x00		

contenu de la mémoire final

Fonctions et procédures

- Procédure : fonction sans valeur de retour
- Étapes pour un appel de fonction
 1. Sauver l'adresse de l'instruction qui suit l'appel de fonction
 2. Positionner le compteur de programme à la première instruction de la fonction appelée
 3. Exécuter la fonction
 4. Positionner le compteur de programme à l'adresse qui suit l'appel, précédemment sauvée
- Instruction `call` combine opérations 1 et 2
 - L'adresse de continuation est placée sur la pile
- Instruction `ret` met en oeuvre 4
- `call` et `ret` modifient la pile et donc `%esp`

```
int g=0;
int h=2;

void increase() {
    g=g+h;
}

void init_g() {
    g=1252;
}

int main(int argc, char *argv[]) {
    init_g();
    increase();
    return(EXIT_SUCCESS);
}
```

Illustration

```

int g=0;
int h=2;

void increase() {
    g=g+h;
}

void init_g() {
    g=1252;
}

int main(int argc, char *argv[]) {
    init_g();
    increase();
    return(EXIT_SUCCESS);
}

```

```

increase:      ; étiquette de la première instruction
                movl    g, %eax
                addl    h, %eax
                movl    %eax, g
                ret     ; retour à l'endroit qui suit l'appel

init_g:        ; étiquette de la première instruction
                movl    $1252, g
                ret     ; retour à l'endroit qui suit l'appel

main:
                (...)
                calll   init_g    ; appel à la procédure init_g
A_init_g:      calll   increase  ; appel à la procédure increase
A_increase:    movl    $0, %eax
                addl    $12, %esp
                ret     ; fin de la fonction main

g:             ; étiquette, variable globale g
                .long   0        ; initialisée à 0

h:             ; étiquette, variable globale g
                .long   2        ; initialisée à 2

```


Illustration (2)

```
int g=0;
int h=2;

void q() {
    g=1252;
}

void p() {
    q();
    g=g+h;
}

int main(int argc, char *argv[]) {
    p();
    return(EXIT_SUCCESS);
}
```

```
q:      movl    $1252, g
        ret                                ; retour à l'appelant

p:      subl    $12, %esp                  ; réservation d'espace sur pile
        calll   q                        ; appel à la procédure q
        movl    g, %eax
        addl    h, %eax
        movl    %eax, g
        addl    $12, %esp                ; libération espace réservé sur pile
        ret                                ; retour à l'appelant
```

la procédure p descend le sommet de pile de 12 octets (et le restaure à la fin) pour respecter une convention qui veut que les adresses de retour soient alignées sur des blocs de 16 octets

Fonction avec un argument

- Les arguments sont placés sur la pile
 - Passage des arguments par valeur !
 - Si la fonction appelée doit modifier une variable locale de l'appelant : il faut passer son adresse (passage de paramètre par pointeur)
- Adresse de retour au sommet de la pile, puis
- Premier argument, second, etc.
- Si la fonction appelée souhaite modifier les registres :
 - pour `%eax`, `%edx`, et `%ecx` : l'appelant doit faire une copie de sauvegarde
 - pour `%ebx`, `%edi`, et `%esi` : l'appelé doit faire une copie et les restaurer
 - convention mais facilite la composition de codes différents

Exemple

```
int g=0;
int h=2;
void init(int i, int j) {
    g=i;
    h=j;
}

int main(int argc, char *argv[]) {
    init(1252,1);
    return(EXIT_SUCCESS);
}
```

```
init:
    subl    $8, %esp           ; réservation d'espace sur la pile
    movl    16(%esp), %eax      ; récupération second argument
    movl    12(%esp), %ecx      ; récupération premier argument
    movl    %ecx, 4(%esp)       ; sauvegarde sur la pile
    movl    %eax, (%esp)        ; sauvegarde sur la pile
    movl    4(%esp), %eax       ; chargement de i
    movl    %eax, g             ; g=i
    movl    (%esp), %eax        ; chargement de j
    movl    %eax, h             ; h=j
    addl    $8, %esp           ; libération de l'espace réservé
    ret

main:
    pushl   %esi
    subl    $40, %esp
    movl    52(%esp), %eax
    movl    48(%esp), %ecx
    movl    $1252, %edx
    movl    $1, %esi
    movl    $0, 36(%esp)
    movl    %ecx, 32(%esp)
    movl    %eax, 24(%esp)
    movl    $1252, (%esp)       ; premier argument sur la pile
    movl    $1, 4(%esp)         ; deuxième argument sur la pile
    movl    %esi, 20(%esp)
    movl    %edx, 16(%esp)
    calll   init                ; appel à init
    movl    $0, %eax
    addl    $40, %esp
    popl    %esi
    ret
```

Fonction et valeur de retour

- `init()` et `sum()` doivent retourner une valeur, que la fonction appelante puisse récupérer après l'exécution (donc après l'instruction `ret`)
- En IA32, la convention est de stocker la valeur de retour dans le registre `%eax`

```
int g=0;
int h=2;

int init() {
    return 1252;
}

int sum(int a, int b) {
    return a+b;
}

int main(int argc, char *argv[]) {
    g=init();
    h=sum(1,2);
    return(EXIT_SUCCESS);
}
```

Exemple

```
int g=0;
int h=2;

int init() {
    return 1252;
}

int sum(int a, int b) {
    return a+b;
}

int main(int argc, char *argv[]) {
    g=init();
    h=sum(1,2);
    return(EXIT_SUCCESS);
}
```

```
init:
    movl    $1252, %eax
    ret

sum:
    subl    $8, %esp           ; réservation d'espace sur la pile
    movl    16(%esp), %eax     ; récupération du second argument
    movl    12(%esp), %ecx     ; récupération du premier argument
    movl    %ecx, 4(%esp)
    movl    %eax, (%esp)
    movl    4(%esp), %eax      ; %eax=a
    addl    (%esp), %eax       ; %eax=a+b
    addl    $8, %esp           ; libération de l'espace réservé
    ret

main:
    subl    $28, %esp
    movl    36(%esp), %eax
    movl    32(%esp), %ecx
    movl    $0, 24(%esp)
    movl    %ecx, 20(%esp)     ; sauvegarde sur la pile
    movl    %eax, 16(%esp)     ; sauvegarde sur la pile
    calll   init
    movl    $1, %ecx
    movl    $2, %edx
    movl    %eax, g
    movl    $1, (%esp)         ; premier argument
    movl    $2, 4(%esp)        ; second argument
    movl    %ecx, 12(%esp)     ; sauvegarde sur la pile
    movl    %edx, 8(%esp)      ; sauvegarde sur la pile
    calll   sum
    movl    $0, %ecx
    movl    %eax, h
    movl    %ecx, %eax
    addl    $28, %esp
    ret
```

Fonctions récursives

```

sumn:
    subl    $28, %esp        ; réservation d'espace sur la pile
    movl    32(%esp), %eax    ; récupération argument
    movl    %eax, 20(%esp)    ; sauvegarde sur pile
    cmpl    $1, 20(%esp)
    jg      .LBB1_2          ; jump si n>1
    movl    20(%esp), %eax    ; récupération n
    movl    %eax, 24(%esp)
    jmp     .LBB1_3

.LBB1_2:
    movl    20(%esp), %eax
    movl    20(%esp), %ecx
    subl    $1, %ecx          ; %ecx=n-1
    movl    %ecx, (%esp)      ; argument sur pile
    movl    %eax, 16(%esp)

recursion:
    calll   sumn
    movl    16(%esp), %ecx    ; récupération de n
    addl    %ecx, %eax        ; %eax=%eax+n
    movl    %eax, 24(%esp)

.LBB1_3:
    movl    24(%esp), %eax
    addl    $28, %esp        ; libération de l'espace réservé sur la pile
    ret

```

Conclusion

Conclusion

- Un processeur effectue des opérations très simples mais très rapidement
- Le principal facteur de (non) performance est la latence d'accès à la mémoire
 - Utilisation de caches
- Correspondance assez directe entre les constructions de C (structures de contrôle, pointeurs, etc.) et leur traduction en instructions
 - Correspond à la volonté de Ritchie lors de la conception du langage