Généralités sur les architectures des ordinateurs

Alain MÉRIGOT

Université Paris Saclay

Constituants d'un ordinateur

Un ordinateur comprend trois parties principales pour :

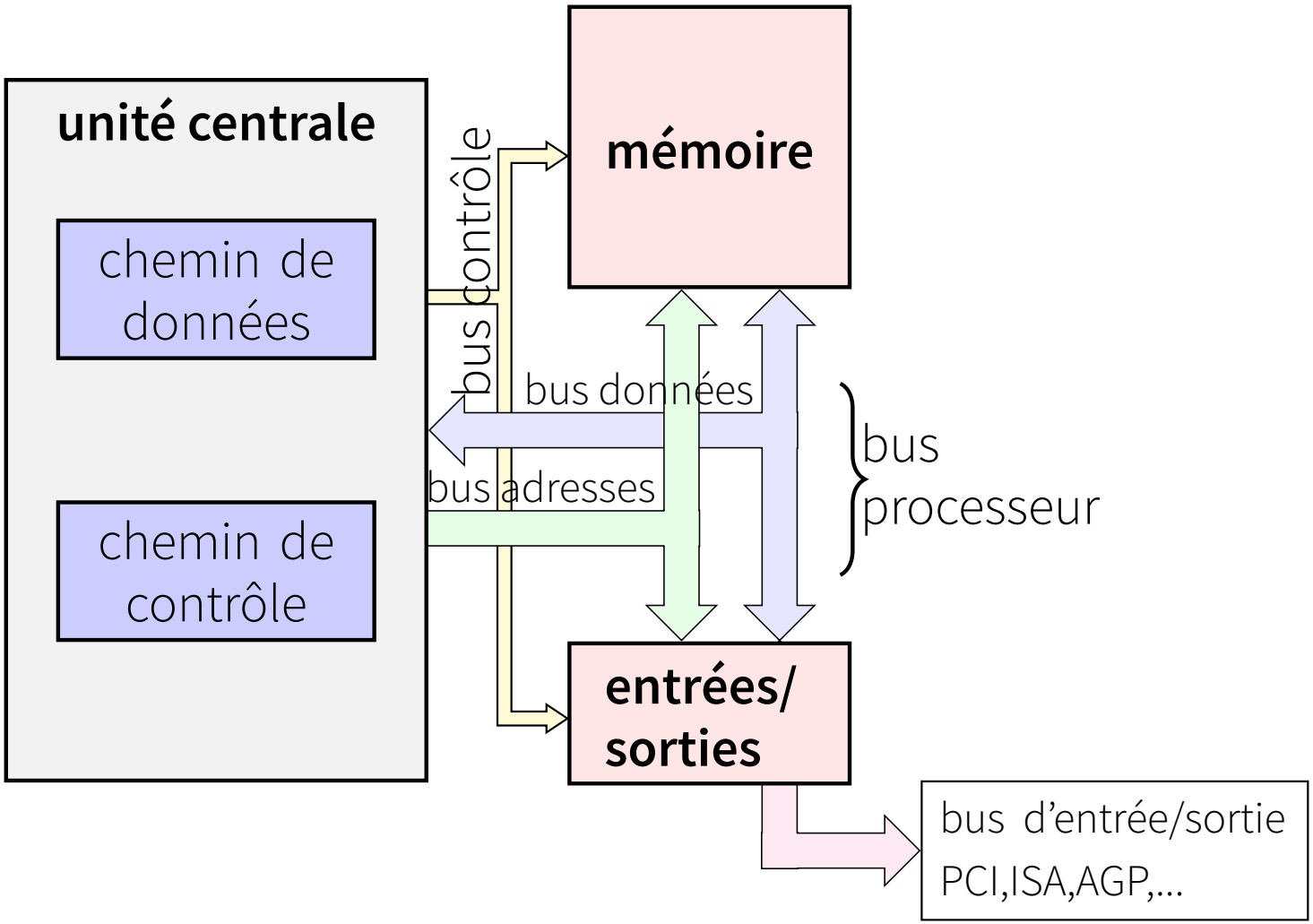
- *traiter* les informations
- *mémoriser*
- et *communiquer* avec l'extérieur.

L'*unité centrale* (UC ou CPU (*Central Processing Unit*)) ou simplement *processeur* transforme les *données* suivant des directives appelées *instructions*.

Ces informations (données et instructions) sont stockées dans l'organe de mémorisation : la *mémoire*

La communication avec l'extérieur (ou les *périphériques*) se fait par le biais des organes d'entrée/sortie.

Constituants d'un ordinateur (cont.)



Constituants d'un ordinateur (cont.)

Ces composants communiquent au moyen du *bus de communication* ou *bus processeur*
Il comprend :

- Le *bus de données* support bidirectionnel sur lequel les informations circulent
- Le *bus d'adresse* qui permet de différencier les informations concernées (emplacement mémoire, organe d'entrée-sortie)
- Le *bus de contrôle* contient des signaux de service pour ces échanges (horloge, sens de transfert mémoire lecture/écriture, indications pour les entrée-sortie, etc.)[1]

[1] On distingue également les **bus d'entrée-sortie** (PCI) qui connectent à des contrôleurs spécialisés, gérant eux même des **bus périphériques** (SATA, IDE, USB, etc).

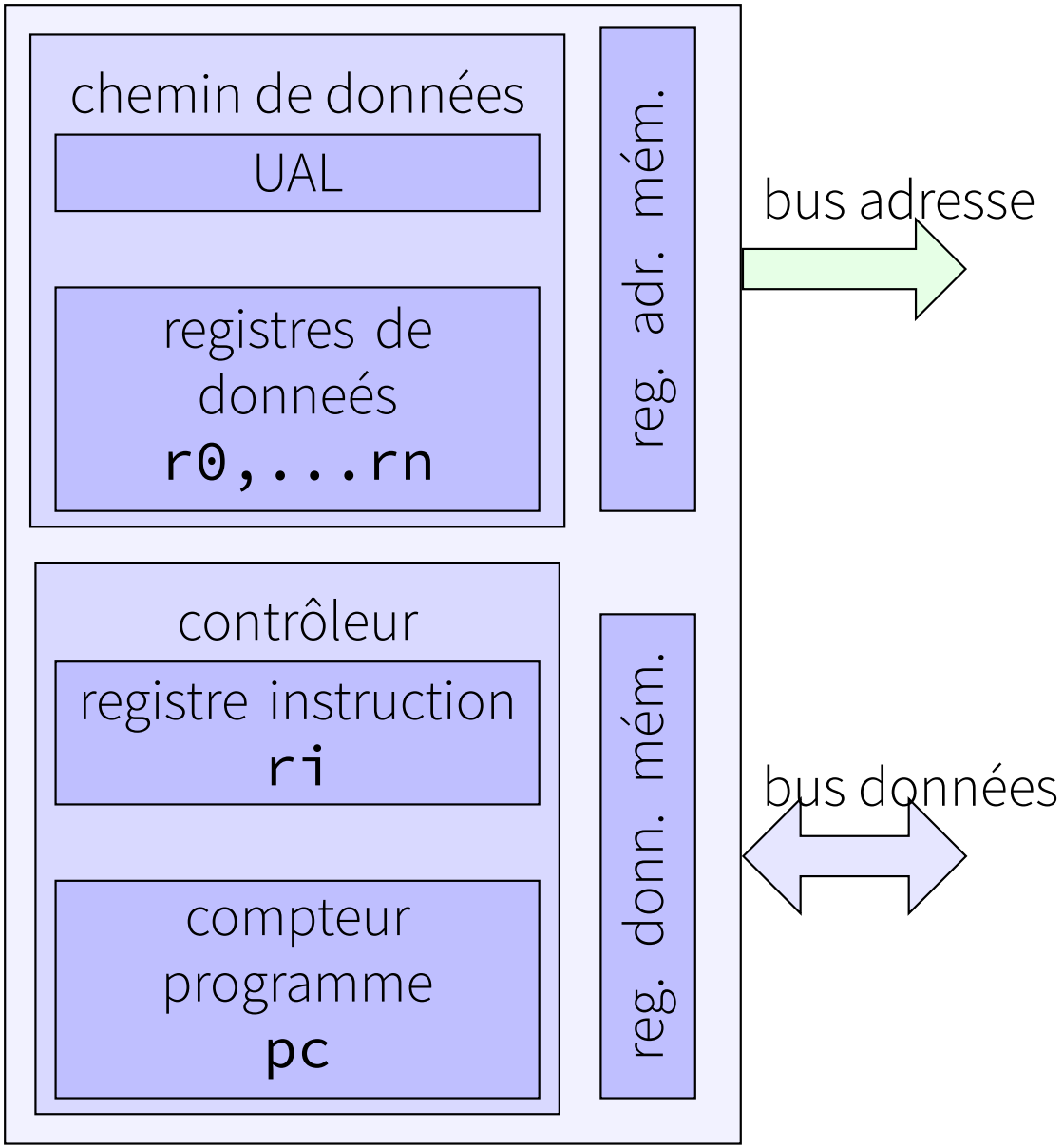
L'unité centrale

Comprend deux parties principales :

- le *séquenceur* (ou *contrôleur*), qui dirige l'exécution des instructions. Il comprend
  - un automate réalisant le séquencement des opérations
  - des registres contenant des informations sur l'instruction en cours d'exécution :
    - registre d'instruction `ri` (contenu de l'instruction)
    - compteur de programme `pc` (adresse en mémoire de l'instruction à exécuter)
- le *chemin de données* qui permet de transformer les données. Il comprend généralement :
  - une *unité arithmétique et logique*
  - des *registres de données* (nommés par exemple `r1`, `r2`, etc)

D'autres registres particuliers ou des registres stockant temporairement l'information (*registres tampons*) peuvent également exister suivant les architectures des processeurs.

L'unité centrale (cont.)



Principaux organes d'une unité centrale

Les instructions d'un processeur

Deux grands types d'instructions :

instructions de modification des données modification de l'état des données stockées dans le processeur et/ou dans la mémoire (calculs, déplacements de données entre la mémoire et le processeur, etc.) *Une fois l'instruction i exécutée, on passe à l'instruction suivante* (celle à l'adresse $i + 1$, ou `pc` +1, `pc` étant le compteur de programme du processeur).

instructions de contrôle de flot Permettent de faire des ruptures du flot d'exécution du programme :

- tests (*if ... then ... else ...*, *switch*)
- boucles (*for*, *while*)
- appels/retour de fonctions

On modifie le compteur de programme `pc`.

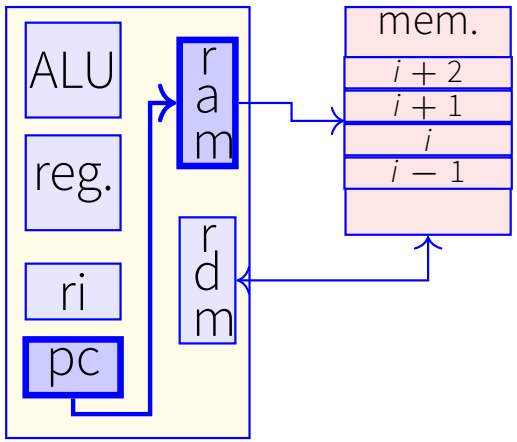
Exécution d'une instruction

Plusieurs actions à réaliser.

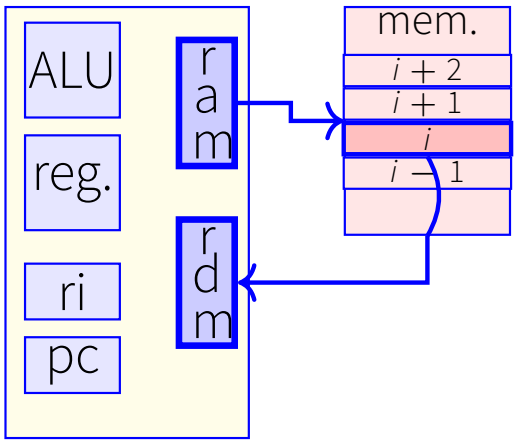
1. *Acquisition de l'instruction*. On va chercher le contenu de la case mémoire dont l'adresse est `pc` et on le copie dans le registre d'instruction `ri`.
2. *Exécution de l'instruction*. On *décode* le contenu du registre `ri`, et on réalise les mouvements de données correspondants à l'instruction.
3. *Préparation de l'instruction suivante*. On met à jour le contenu du compteur de programme `pc`
   - pour une instruction de *modification des données*, on remplace le contenu de `pc` par `pc+1`
   - pour une instruction de *contrôle de flot*, on calcule le nouveau `pc` en fonction de l'instruction.

Exécution d'une instruction (cont.)

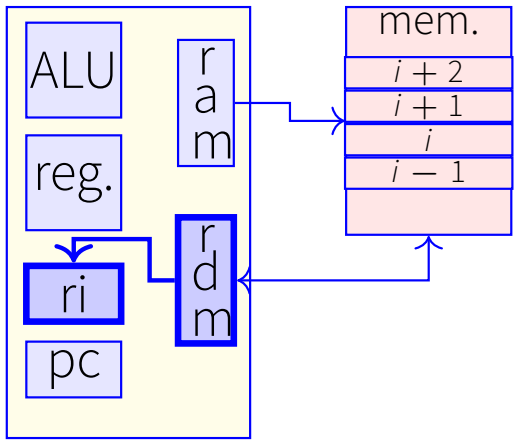
Phase d'acquisition



Transfert de l'adresse de l'instruction à exécuter (`cp`) vers le registre d'adresse mémoire (`ram`).

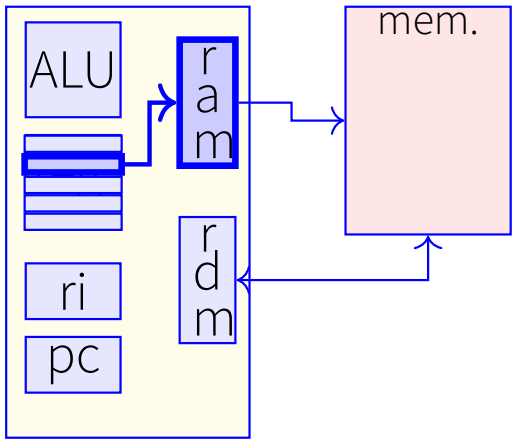


Lecture de la mémoire et recopie du contenu de la case adressée par le registre `ram` dans le registre de données mémoire (`rdm`)

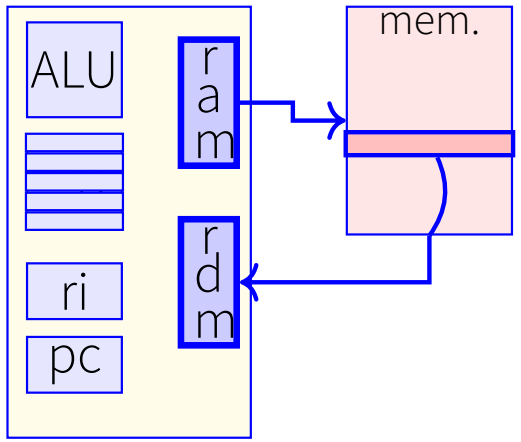


Recopie de l'instruction (dans le registre `rdm`) vers le registre d'instruction `ri`.

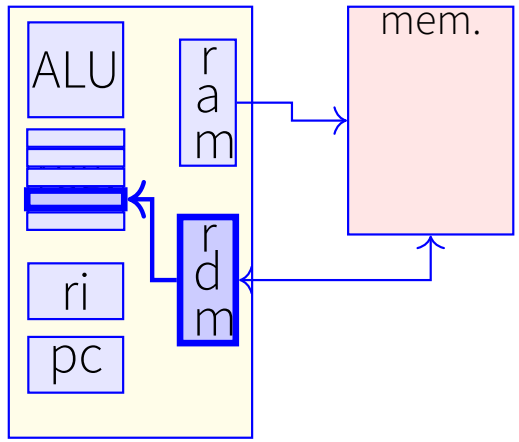
Exemple de l'instruction : `ld r4, (r2)` (registre `r4` ← mémoire[`r2`])



copier `r2` dans le registre `ram`



lire la case mémoire pointée par le registre `ram` et copier le contenu dans le registre `rdm`



copier le contenu du registre `rdm` vers le registre destination `r4`

Grandes familles de processeurs

L'efficacité d'un ordinateur pour exécuter un programme dépend de divers facteurs :

- nombre d'instructions exécutées
- rapidité d'exécution d'une instruction (nombre de cycles)
- nombre d'accès à la mémoire
- fréquence de fonctionnement
- etc

Elle dépend largement des choix du *jeu d'instructions*.

Grandes familles de processeurs (cont.)

Premiers processeurs (1950–1980) :

- programmés principalement en assembleur
- on favorise un jeu d'instruction contenant des instructions sophistiquées : processeurs CISC (*complex instruction set computer*) (IBM370, vax, x86 (intel), 68000 (motorola), etc)

Les instructions CISC peuvent :

- comporter des calculs avec opérandes/résultat en mémoire
- manipuler des zones mémoires de taille quelconque (copie d'une zone mémoire, recherche d'un caractère dans une chaîne, etc).
- itérer une ou plusieurs instructions
- être codées par un nombre quelconque d'octets

Ces caractéristiques posent de nombreux problèmes, sont peu utilisées par les compilateurs et interdisent une mise en oeuvre architecturale efficace.

Grandes familles de processeurs (cont.)

A partir des années 80 :

- la quasi totalité de la programmation se fait en langage de haut niveau
- on favorise un jeu d'instruction simple et efficace, l'optimisation étant faite par le compilateur. Processeurs RISC (*Reduced Instruction Set Computer*) (MIPS, SPARC, PowerPC, Arm, RISC-V, Nios, etc).
  - les instructions de calcul ne manipulent que des registres
  - les accès mémoire ne sont pas associés à des calculs (jeu d'instructions *load–store*)
  - les instructions sont de taille identiques et *alignées* en mémoire

Cette simplification permet de mettre en oeuvre des mécanismes architecturaux accélérant *largement* les traitements : pipeline (lancement d'une instruction/cycle), superscalaire (plusieurs instructions parallèles), etc... Tous les processeurs actuels reposent sur des mécanismes RISC[2].

[2] Le pentium a un jeu d'instructions complexe (appelé x86 ou IA-32), mais un coeur de traitement à base d'instructions RISC (μops)

Caractéristiques des instructions

Adressage des opérandes

Adressage = spécification d'un opérande (registre ou mémoire)

mode	exemple	interprétation	utilisation
registre	`add r1,r2,r3`	`r1←r2+r3`	données en registre
immédiat	`add r1,r2,#5`	`r1←r2+5`	constante entière
direct	`add r1,r2,(1234)`	`r1←r2+mem[1234]`	var. globale (statique)
ind. registre	`add r1,r2, (r3)`	`r1←r2+mem[r3]`	déréférence de pointeur
basé	`add r1,r2,16(r3)`	`r1←r2+mem[r3+16]`	accés aux variables locales. base = pointeur de pile. (calculé à la compilation)
indexé	`add r1,r2,(r3+r4)`	`r1←r2+mem[r3+r4]`	accés systématique aux éléments d'un tableau
ind. mém.	`add r1,r2, @(r3)`	`r1←r2+mem[mem[r3]]`	déréférence de pointeur
étendu	`add r1,r2,4(r3)[r4]`	`r1←r2+mem[4+r3+r4]`	
post-incrémenté	`add r1,r2,(r3)+`	`r1←r2+mem[r3]` `r3←r3+1`	parcours de tableau ou accès a une pile
pré-décrémenté	`add r1,r2,−(r3)`	`r3←r3−1` `r1←r2+mem[r3]`	idem

Tous les types d'adressage n'ont pas la même utilité :

- Registre : indispensable (50% des instructions au moins). Permet des calculs entre registres du processeur.

  Pour les instructions restantes, sont utiles :
- Immédiat : indispensable pour des instructions contenant des constantes (`x=5; if(x>2)...; i++,` etc) (20–40% des instructions)
- Basé : permet notamment la manipulation des données dans la pile (arguments de fonction, variables locales), les accès aux champs d'un *struct*, etc. Très utile (large majorité des instructions d'accès à la mémoire)
- Indirect-registre : cas particulier d'un adressage basé avec base nulle. Indispensable pour déréférencer un pointeur (`*x`)
- Les autres instructions sont moins utiles et rarement présentes dans les processeurs RISC

Les instructions peuvent manipuler des données de différents types : octets (`char`, double octet (`short`), mot de 32 bit (`int` ou `long`), mot de 64 bits (`long long`), flottant simple ou double précision.

Les données *autres que les octets* sont sujettes à des contraintes :

alignement : adresses des données de plus d'un octet

arrangement des octets d'un mot : ordre des octets des données

Les contraintes dépendent des choix architecturaux.

leur non-respect peut conduire à programmes non portables entre architectures différentes.

Caractéristiques des instructions (cont.)

alignement des données

Alignement : Une donnée de 2^k octets doit être à une adresse multiple de 2^k.
Exemple : un `int` (4o) **doit** être à une adresse multiple de 4.
Un `struct` est aligné sur son plus grand champ.
Pour respecter l'alignement, ajout d'octets de remplissage (*padding*).

```
struct s1 { char c1;   // 1 octet + 3 pour l'alignement de i1
            int  i1;   // 4 octets
            char c2;   // 1 octet + 3 pour l'alignement de i2
            int  i2;   // 4 octets
};  // sizeof(struct s1)==16
struct s2 { int  i1;   // 4 octets
            int  i2;   // 4 octets
            char c1;   // 1 octet
            char c2;   // 1 octet + 2 pour alignement de struct s2
};  // sizeof(struct s2)==12
```

L'alignement assure que tous les octets d'une donnée élémentaire sont
- dans la même rangée mémoire
- dans la même ligne de cache

Toutes les architectures récentes nécessitent des données alignées.
Le pentium peut traiter des données non alignées, mais les performances sont meilleures avec alignement.

arrangement des octets d'un mot Différents choix sont possibles pour l'emplacement en mémoire d'un mot sur plusieurs octets.

Exemple : `0xfedcba98` à l'adresse `0x1000`

adr.	`1000`	`1001`	`1002`	`1003`	
	`fe`	`dc`	`ba`	`98`	***big-endian*** ou gros-boutiste (poids fort en tête) (SPARC,MC68000, TCP-IP)
	`98`	`ba`	`dc`	`fe`	***little-endian*** ou petit-boutiste (poids faible en tête) (x86, nios)

Sur de nombreux processeurs modernes, choix *au démarrage* du mode petit-boutiste ou gros-boutiste (Arm, power-PC, Mips, etc) (*bi-endian*).

Attention à la portabilité des programmes !!
On peut tester si `__BYTE_ORDER__` vaut `__ORDER_LITTLE_ENDIAN__` ou `__ORDER_BIG_ENDIAN__`

Caractéristiques des instructions (cont.)

boutisme

Des problèmes peuvent apparaitre lors d'équivalence entre pointeurs vers des types différents.

Exemple : extraction des octets de poids faible et de poids fort d'un mot

```
// ***mauvaise*** extraction des LSB et MSB par équivalence de pointeurs
int ii;
char *cc, lsb, msb;
ii = 0xfedcba98;
cc = (char *) &ii; // A proscrire !!! Équivalence entre char* et int*
lsb = cc[0];
msb = cc[3] ;
// cc[0] et cc[3] ne sont le LSB et MSB qu'en little-endian
// Programme non portable, donnant un "comportement indéfini"
// en C et en C++ car ne respectant pas le "strict aliasing"
```

Mais le boutisme ne concerne que des données *en mémoire*.
Le plus simple et portable est d'utiliser un registre.

```
int ii=0xfedcba98;
char lsb,msb;
// on manipule les données dans un registre : parfaitement portable
lsb=ii&0xff; // ou simplement lsb=ii;
msb=(ii>>24)&0xff ;
```

Instructions de contrôle

Les instruction de contrôle sont de quatre types :

- branchements conditionnels (80–90% des instructions de contrôle)
- sauts inconditionnels
- appels de procédure
- retour de procédure

Les branchements servent à réaliser les tests (`if`) et les boucles (`for`, `while`) des langages de haut niveau. Généralement réalisés par un déplacement *relatif* au compteur de programme.

Ils résultent souvent d'une instruction de comparaison

- instruction explicite de comparaison vers un registre banalisé.
- utilisation d'informations sur les résultats d'opérations positionnées implicitement par l'UAL dans un *registre d'état*
- instructions combinant test et branchement

Loi d'Amdahl

Supposons un programme avec deux parties de temps d'exécution $t = t_1 + t_2$.
Un « dispositif » accélére la partie 2 par un facteur $A > 1$ ($t_2' = t_2/A < t_2$).

L'accélération globale A' du programme sera :

$$\begin{aligned} A' &= \frac{t_1+t_2}{t_1+t_2/A} \\ &= A \times \frac{1}{A \times \frac{t_1}{t_1+t_2} + \frac{t_2}{t_1+t_2}} \end{aligned}$$

Donc $A' < A$, d'autant plus que t_1 est important par rapport à t_2.

Exemple : $t_1 = 90, t_2 = 10, A = 10 \rightarrow A' = 1.098$

Il est donc inutile d'accélérer des fonctions dont l'occurrence est faible.

"Make the common case fast"

Loi de Hennessy-Patterson

$$t_{prog} = n_i \times T_c \times \overline{CPI} \quad (1)$$

Trois facteurs déterminent les performances d'un ordinateur pour l'exécution d'un programme :

- nombre d'instructions nécessaires n_i
- période de l'horloge du processeur T_c,
- nombre moyen de cycles nécessaires pour exécuter une instruction $\overline{CPI}$.

Si on a n, et T et un $\overline{CPI} = 5$ pour un processeur CISC, généralement pour un processeur RISC de même technologie, on aura :

- $n' = 1.2n$ (légère augmentation du nombre d'instructions)
- $T' = 0.7T$ (instructions plus simples $\Rightarrow$ T horloge$\searrow$)
- $\overline{CPI}' = 1.4$ (et même $\overline{CPI} < 1$ pour les processeurs modernes)
  $t' = 1.2 \times 0.7 \times 1.4 \times n \times T = 1.176 \times n \times T < t = 5 \times n \times T$