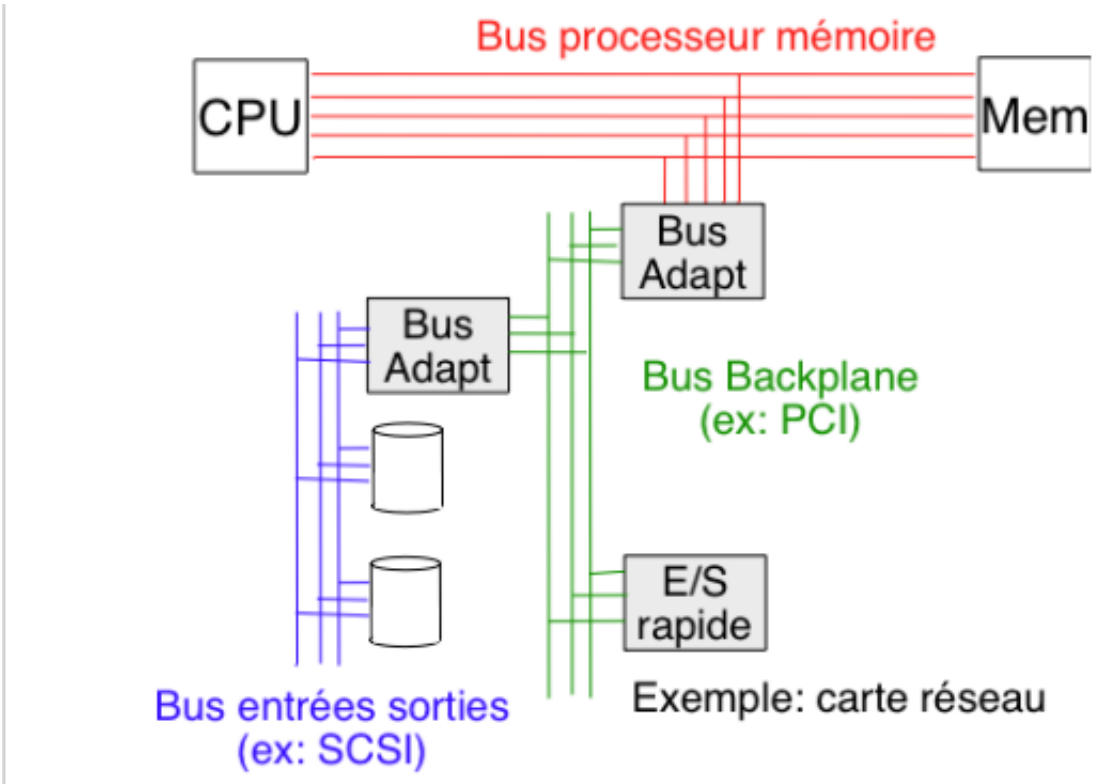
Chapitre 3 : Structure des Ordinateurs

Section 3.1 : Organisation des Ordinateurs

3.1.1 Le modèle VonNeumann

Le modèle VonNeumann est un des modèles qui est toujours d'actualité dans la conception d'un ordinnateur. Il se compose comme suit :

- Unité centrale ou Processeur
  - unité de commande (charge, décode et exécute les instructions)
  - unité arithmétique et logique (+, -, *, /)
- Mémoire
  - données traités par le processeur
  - instructions du programme



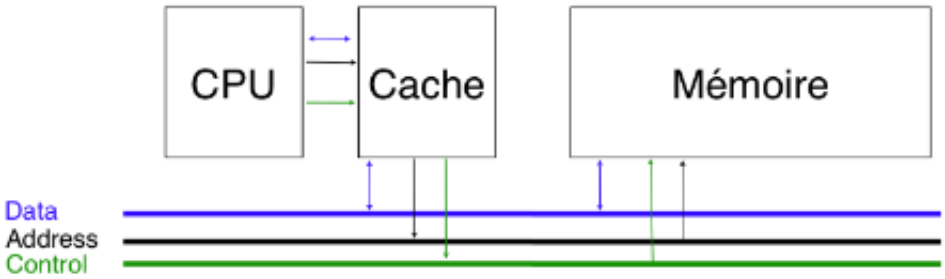
info: Une mémoire permettant de sotcker 2^k bytes de données utilisera au minimum k bits pour représenter son adresse en mémoire.

3.1.2 DRAM et SRAM

- SRAM
  - plus gourmand (-)
  - plus cher (-)
  - plus petites capacitées (-)
  - plus rapide (+)
- Dram
  - moins gourmand (+)
  - moins cher (+)
  - plus grandes capacitées (+)
  - moins rapide (-)

3.1.3 Mémoire Cache

Pour permettre d'augmenter la vitesse d'exécution maintenant bridée par la vitesse de la mémoire, les constructeurs ont implémenté dans les processeurs actuels, des mémoires de faibles de capacités mais rapides. Il s'agit de la **mémoire cache. Elle récupère des données des mémoires à plus grandes capacités. et sert d'interface entre le processeur et la mémoire principale.



Une **mémoire cache** est une mémoire de faible capacité mais rapide.

Les Mémoires cache utilisent les principes de localités :

- **Localité temporelle** : Si le processeur à besoin de l'information à l'adresse x au moment t, il est fort probable qu'il en ai besoin à l'instant t+1.
- **Localité Spatiale**: Si le processeur à besoin de l'information à l'adresse x à l'instant t, il est fort probable qu'il ai besoin d'accéder à l'adresse x+1 à l'instant t+1.

vitesses relatives

- **cache** (SRAM): 1nsec
- **Mémoire Principale** (DRAM): 50nsec

Acces en Lecture

Quand le Processeur à bsoin de lire une donnée à une adresse, il fournit l'adresse au cache. Si la donnée est présente, elle est directement renvoyée. Dans le cas contraire, la mémoire cache interroge la RAM, se mets à jour et ensuite fournit l'info au processeur (localité temporelle). Lors du chargement depuis la RAM, le cache charge directement une lignen de cache (jusqu'à quelques dizaines d'adresses consécutives à la fois) en mémoire. Les RAM modernes sont optimisées pour fournir des débits élevées pour des adresses consécutives.

Acces en Ecriture

L'écriture est un peu plus compliquée car il faut que la ram et le cache soient tous les deux mis à jour. Voici deux techniques:

- **Write Trough** : une demande en écriture donne lieu à une écriture en cache et en RAM
  - Les performances du cache et de la RAM sont limitées à celle de la RAM
- **Write Back**: Une demande d'écriture donne lieu à une écriture directe dans le cache qui sera ensuite réécrite dans la ram lors de la suppression de celle-ci du cache.
  - De meilleures performances mais attentions aux erreurs de synchronisation avec par exemple les ddr ou les cartes réseaux qui ont accès direct à la ram

Section 3.2 : Etude de cas :Architecture IA32

Les Registres

Un processeur IA32 dispose de **huit** registres génériques de 32 bits pouvant contenir des int ou des adresses.

- **eax** :
- **ebx** :
- **ecx** :
- **edx** :

- **esi** :
- **edi** :
- **ebp** : Gestion de la pile
- **esp** : Gestion de la pile
- **eip** : Compteur de programme (Program counter)
- **eflags** : Drapeaux regroupés

Types de donneés supportées

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

3.2.1 Les instructions mov

Les instructions **mov** permettent de **déplacer des données** (entre registres, depuis la mémoire, d'un registre vers la mémoire)

```
    mov src, dest; //déplace src vers dest
```

> movb pour les byte
> movw pour un mot de 16 bits
> movl pour un mot de 32 bits

(mode d'adressage :) Les registres sont utilisées avec le préfixe **%** (ex: *%eax*)

Mode d'adressage: `movl %eax, %ebx; déplacement de %eax vers %ebx`

Mode immédiat: `movl $0, %ecx; initialisation de %ecx à 0`

Mode absolu:
```
movl 0x04, %eax; place la valeur contenue à l'adresse 0x04
dans %eax
```

Mode indirect
```
movl (%eax), %edx; place la valeur contenue à l'adresse elle
même contenue dans %eax dans %edx
```

Mode avec une base et un déplacement

```
movl 4(%eax), %edx; place la valeur contenue à l'adresse elle
même contenue dans %eax(+4) dans %edx
```

3.2.2 Instructions arithmétiques et logiques

Opérations arithmétiques à un registre / une adresse

```
    inc %eax    ;       incrémente et save le resultat du regis
    dec %eax    ;       idem supra mais décremente
    not %eax    ;       applique l'opération logique not et sau
```

Opérations à deux registres

```
    add     addition
    sub     soustraction
    mul multiplication entiers non-signés
    imul    multiplication entiers signés
    div     division entiers non-signés
    shl     décalage logique vers la droite
    shr     décalage logique vers la gauche
    xor     ou exclusif     (xor a, b) == b^a
    and conjonction logique
```

attention, les multiplications peuvent provoquer un dépassement des 32 bits de mémoire.

3.2.3 Les instructions de comparaison

```
    cmp     soustraction et mise a jour de CF et SF sans sauveg
    test    conjonction et mise à jour des flags sans sauvegard
```

Les flags (drapeaux)

```
    ZF (Zero Flag) :    resultat dernière opération était 0
    SF (Sign Flag):     resultat dernière opération était négat
    CF (Carry Flag):    der. opé. non-sign. nécéssitait plus de
    SF (Sign Flag):     der. opé. sign. a provoqué un dépasseme
```

3.2.4 Les instructions de saut

Les instructions de saut permettent de modifier %eip (pile), elles sont nécessaires pour implémenter des tests, des boucles et des appels de fonctions. Prohibée dans

la plus part des langages modernes, elle est indispensable en assembleur.

Dans certains languages, l'instruction goto permets d'effectuer de tels sauts.

> Le c continent l'instruction goto, mais son utilisation est fortement découragée.

```
    LABEL:
    goto LABEL;
```

jmp

```
    jmp *%eax   ;   saute à l'adresse contenue au registre %eax
```

IF

```
.LBB2_1:
    movl    j, %eax ;   %eax=j
    cmpl    g, %eax ;   j<=g
    jle     .LBB2_2 ;   jump si j<=g
    movl    $2, r   ;   r=2
    jmp .LBB2_3 ;   jump fin
.LBB2_2:
    movl    $3, r   ;   r=3
.LBB2_3:
```

While

```
.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 ;   boucle
.LBB3_3:
```

for

```
    movl    $0, j   ;   j=0
.LBB4_1:
    cmpl    $10, j  ;   j<=0
    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;
```

```
    movl    %eax,j  ;
    jmp .LBB4_1 ;   boucle
.LBB4_4:
```

3.2.5 Manipulation de la pile

```
    pushl   %reg    ;
    popl    %reg    ;
```

- **Push**: Place le contenu du registre au sommet de la pile et décrémente le registre %esp de 4 unités
- **Pop**: Retire le mot de 32 bits du sommet de la pile, sauvegarde le résultat dans le registre et incrémente le registre %esp de 4 unités

3.2.6 Les fonctions et procédures

une procédure est un ensemble d'instructions qui peuvent être appelées depuis n'importe quel endroit dans le programme.

Les procédures (sans arguments)

dans cette exemple, on modifira les variables globales

```
void proc(){...}
...
proc();
```

- **call**: instruction de saut qui transfert l'exécution à l'adresse du label et sauvegarde l'adresse de l'instruction qui suit dans la pile
- **ret**: retire l'adresse au sommet de la pile et y jump.

```
    proc:   ...
            ret         ;
    main:
            ...
            calll proc  ;
    A_proc  ...         ;   instruction quelqonque
```

variable globale

```
    g:
        .long   0
```

suite cfr p 79 à 82 du livre partie 3