ARCHITECTURE DES ORDINATEURS

Notes de cours (30 h cours +TD)

MIAGE – Formation Continue

Université de Paris-Sud, 1994-1995

Michel CRUCIANU

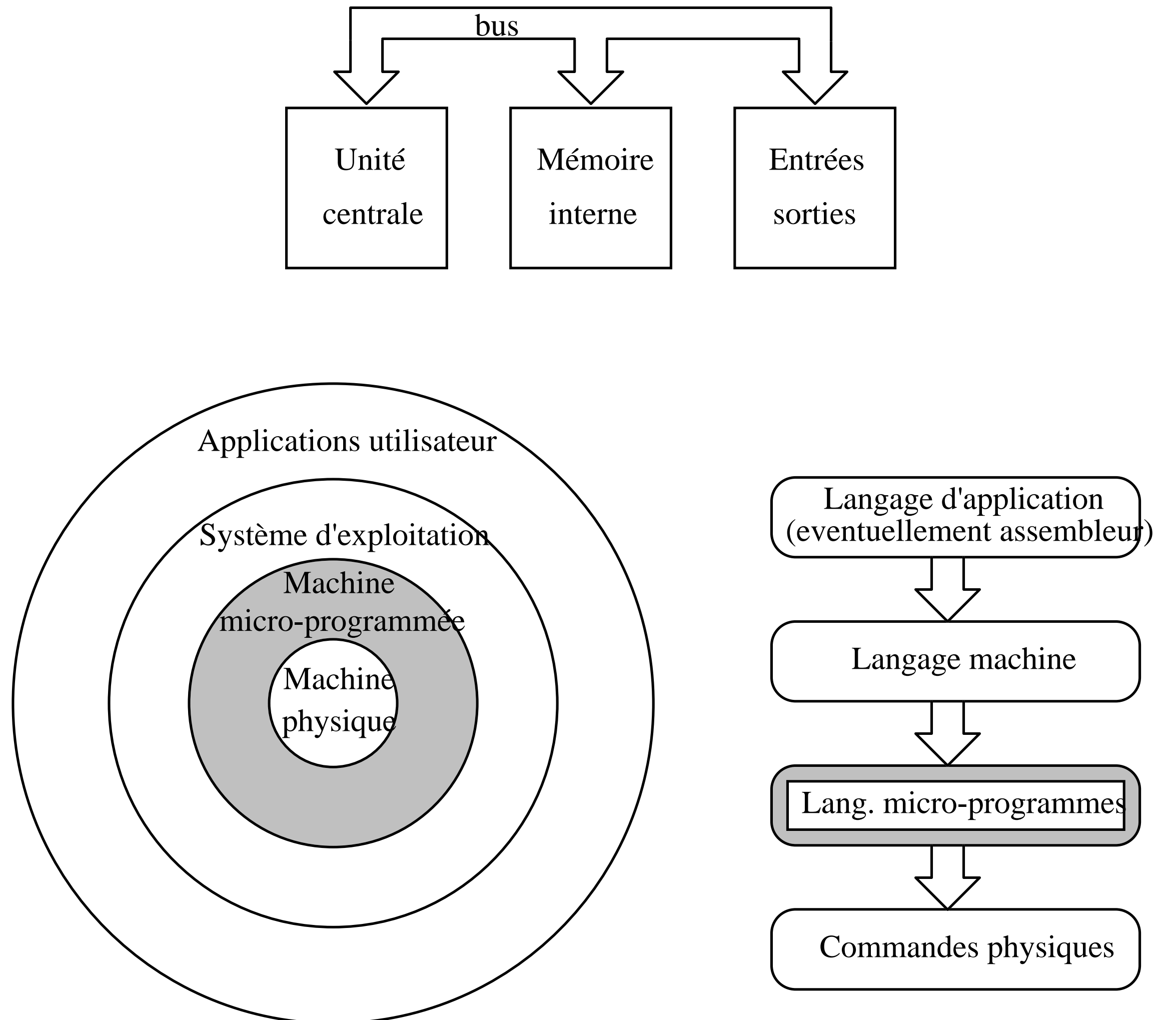
Table des matières

Introduction 5
Structure générale d'un ordinateur 5
1. Représentation de l'information. Arithmétique en binaire. 5
1.1. Information 5
1.2. Représentation des nombres 5
1.2.1. Système de numération en base 2 5
1.2.2. Représentations en "virgule fixe" 7
1.2.2.1. Nombres sans signe : 7
1.2.2.2. Nombres avec signe : 7
1.2.3. Représentations en "virgule flottante" 8
1.3. Arithmétique en binaire 9
1.3.1. Nombres en binaire, sans signe 9
1.3.1.1. L'addition — exemples 9
1.3.1.2. La soustraction (A - B) — exemples 9
1.3.1.3. La multiplication — exemple 10
1.3.1.4. La division — exemple 10
1.3.2. Nombres en complément à 2 10
1.3.2.1. Obtenir le complément à 2 10
1.3.2.2. L'addition — exemples 11
1.3.3. Nombres en DCBN compacte 11
1.3.3.1. L'addition — exemples 11
1.3.3.2. La soustraction (A - B) — exemples 11
1.4. Représentation des caractères 12
2. Circuits combinatoires 12
2.1. Algèbre de Boole et opérateurs combinatoires 12
2.2. Synthèse combinatoire 14
2.2.1. Formes canoniques 14
2.2.2. Simplification de l'écriture d'une fonction logique 15
2.2.2.1. Méthode algébrique 15
2.2.2.2. Méthode de Karnaugh 15
2.2.3. Synthèse avec circuits élémentaires (portes) 15
2.2.4. Synthèse avec mémoires mortes (*Read Only Memory, ROM*) 17
2.2.5. Synthèse avec réseaux programmables (*Programmable Logic Array, PLA*) 18
2.3. Bus et circuits "à trois états" (*Tri-State Logic*) 18
3. Circuits séquentiels 18
3.1. Bascules 19
3.1.1. Bascules RS 19
3.1.2. Bascules JK 19
3.1.3. Bascules D 20
3.2. Registres 21
3.2.1. Registres parallèles 21
3.2.2. Registres de déplacement 21
3.3. Compteurs 21
3.4. Synthèse d'automates synchrones 22
4. Circuits complexes 24
4.1. Unité Arithmétique et Logique (UAL, *ALU*) 24
4.1.1. Additionneurs 24
4.1.2. Additionneur/soustracteur 25
4.1.2. Unité Arithmétique et Logique 26
4.2. Mémoire vive (*Random Access Memory, RAM*) 27
4.2.1. Mémoires RAM statiques (*SRAM*) 27
4.2.2. Mémoires RAM dynamiques (*DRAM*) 29
4.2.2. Augmentation de la capacité : utilisation de plusieurs circuits 30
5. Structure et fonctionnement global d'un ordinateur 31
5.1. Structure et fonctionnement 31
5.1.1. Structure simplifiée 31

5.1.2. Fonctionnement ... 31
5.1.2.1. Exécution d'une instruction ... 31
5.1.2.2. Exécution des programmes ... 32
5.2. Architecture et performances ... 32
5.2.1. Temps d'exécution ... 32
5.2.2. Amélioration des accès mémoire ... 32
5.2.3. Modèles d'exécution et réduction du temps d'exécution ... 33
6. Structure et fonctionnement de l'unité centrale ... 35
6.1. Structure et fonctionnement d'une unité centrale simple ... 35
6.2. Structure et fonctionnement de l'unité centrale SPARC ... 36
6.2.1. Registres internes ... 36
6.2.2. Bus, *pipeline*, contrôle ... 37
7. Les instructions du SPARC ... 39
7.1. Registres et types de données ... 39
7.2. Types et formats d'instructions. Modes d'adressage ... 41
7.3. Instructions de transfert registres UC ↔ mémoire ... 42
7.4. Instructions arithmétiques, logiques et de translation (*shift*) ... 43
7.5. Instructions de transfert de contrôle ... 44
Appel d'une procédure : ... 45
Retour d'une procédure : ... 46
Exemples d'utilisation : ... 47
7.7. Autres instructions ... 49
8. Organisation et gestion de la mémoire ... 50
8.1. Hiérarchie de mémoires ... 50
8.2.1. La mémoire cache et sa gestion ... 50
8.2.2. Mémoire principale et mémoire virtuelle ... 52
9. Les entrées/sorties (E/S, *I/O*) et leur gestion ... 53
9.1. Types de dispositifs d'E/S ... 53
9.2. L'interface avec l'UC ... 54
9.3. Dialogue avec les périphériques, interruptions ... 54
9.4. Accès direct à la mémoire (*Direct Memory Access, DMA*) ... 54
9.5. L'interface avec la mémoire ... 55

Introduction

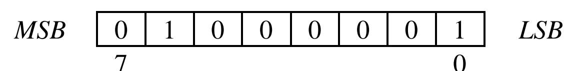
Structure générale d'un ordinateur



1. Représentation de l'information. Arithmétique en binaire.

1.1. Information

Unité d'information : le ***bit*** — permet de lever une ambiguïté élémentaire (oui/non)
Groupe ordonné de 8 bits : **octet** (***byte***)



Groupe ordonné de 4 bits : **quartet**
Unités employées couramment : 1 Koctet = 1024 octets (= 2^{10} octets)
1 Moctet = 2^{10} Koctets = 2^{20} octets

1.2. Représentation des nombres

1.2.1. Système de numération en base 2

Si b est un nombre entier plus grand que 1, tout nombre N peut se mettre sous la forme

$$N = a_n \cdot b^n + a_{n-1} \cdot b^{n-1} + \cdots + a_1 \cdot b^1 + a_0 \cdot b^0 + a_{-1} \cdot b^{-1} + a_{-2} \cdot b^{-2} + \cdots \text{, avec } 0 \le a_i < b.$$

En base *b*, *N* s'écrit alors

$$N = a_n a_{n-1} \cdots a_2 a_1 a_0 , a_{-1} a_{-2} \cdots .$$

Nous utiliserons des représentations en base 2 (en binaire), 8 (en octal), 16 (en hexadécimal). En hexadécimal, les chiffres qui composent les représentations des nombres sont

$$a_i \in \{0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ \mathrm{A}(10),\ \mathrm{B}(11),\ \mathrm{C}(12),\ \mathrm{D}(13),\ \mathrm{E}(14),\ \mathrm{F}(15)\}$$

La partie entière $a_n a_{n-1} \cdots a_2 a_1 a_0$ d'un nombre *N* en base *b* peut s'écrire :

$$a_n \cdot b^n + a_{n-1} \cdot b^{n-1} + \cdots + a_1 \cdot b^1 + a_0 \cdot b^0,$$

ou $$b \cdot \left(a_n \cdot b^{n-1} + a_{n-1} \cdot b^{n-2} + \cdots + a_1\right) + a_0,$$

ou $$b \cdot \left(b \cdot \left(b \cdots \left(b \cdot \left(a_n\right) + a_{n-1}\right) + \cdots\right) + a_1\right) + a_0,$$

donc $a_0, a_1, \cdots, a_n$ sont les restes successifs de la division de *N* par la base *b*.

Par exemple, pour coder $(56)_{10}$ en base 2, il faut effectuer des divisions successives par 2 et garder les restes successifs ainsi que le dernier quotient :

```
56 | 2
 0 | 28 | 2
      0 | 14 | 2
           0 |  7 | 2
                1 | 3 | 2
                    1 | 1
```

donc on obtient $(56)_{10} = (111000)_2$.

La partie fractionnaire $a_{-1} a_{-2} a_{-3} a_{-4} \cdots$ de *N* en base *b* peut s'écrire :

$$a_{-1} \cdot b^{-1} + a_{-2} \cdot b^{-2} + a_{-3} \cdot b^{-3} + a_{-4} \cdot b^{-4} + \cdots,$$

ou $$b^{-1} \cdot \left(a_{-1} + b^{-1} \cdot \left(a_{-2} + b^{-1} \cdot \left(a_{-3} + b^{-1} \cdot \left(a_{-4} + \cdots\right)\right)\right)\right),$$

donc $a_{-1}, a_{-2}, a_{-3}, a_{-4}, \cdots$ sont les chiffres qui "glissent" successivement à gauche de la virgule lors de multiplications successives par la base.

Par exemple, écrivons $(0{,}3125)_{10}$ en base 2 :

0,3125 × 2 = **0**,625
0,625 × 2 = **1**,25
0,25 × 2 = **0**,5
0,5 × 2 = **1**, donc $(0{,}3125)_{10} = (0{,}0101)_2$.

Codage en hexadécimal, en octal et en binaire des nombres de 0 à 15 :

Décimal	Hexa	Octal	Binaire
0	0	0	0000
1	1	1	0001
2	2	2	0010
3	3	3	0011
4	4	4	0100
5	5	5	0101
6	6	6	0110
7	7	7	0111
8	8	10	1000
9	9	11	1001
10	A	12	1010
11	B	13	1011
12	C	14	1100
13	D	15	1101
14	E	16	1110
15	F	17	1111

Conversion binaire → octal : nous séparons les chiffres en groupes de 3, en partant de la virgule, et nous remplaçons chaque groupe par son équivalent en octal (voir les 8 premières lignes du tableau précédent). Par exemple, pour 11010101001,01011 :

011 010 101 001, 010 110

3 2 5 1 , 2 6

donc $(11010101001{,}01011)_2 = (3251{,}26)_8$.

Conversion binaire → hexadécimal : nous séparons les chiffres en groupes de 4, en partant de la virgule, et nous remplaçons chaque groupe par son équivalent en hexadécimal (voir le tableau précédent). Par exemple, pour 11010101001,01011 :

0110 1010 1001, 0101 1000

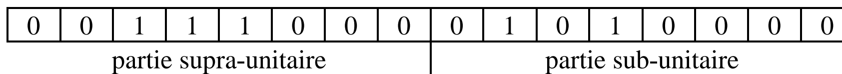
6 A 9 , 5 8

donc $(11010101001{,}01011)_2 = (6A9{,}58)_H$.

1.2.2. Représentations en "virgule fixe"

1.2.2.1. Nombres sans signe :

1° Habituellement, un nombre est représenté par sa traduction en binaire, sur un nombre fixé de bits (en général 8, 16, 32 ou 64) ; la position de la virgule est fixe. Par exemple, si 8 bits sont alloués à la partie supra-unitaire et 8 bits à la partie sub-unitaire du nombre à représenter, $(56{,}3125)_{10}$ est représenté par



0	0	1	1	1	0	0	0	0	1	0	1	0	0	0	0
partie supra-unitaire								partie sub-unitaire							

La valeur maximale que nous pouvons représenter est $2^8\text{-}2^{-8}$ (11111111,11111111 en binaire) et la valeur minimale 2^{-8} (0,00000001). Nous constatons que la fidélité de la représentation (le nombre de chiffres significatifs gardés) dépend directement de la valeur à représenter : plus le nombre à représenter est petit, moins on peut garder de chiffres significatifs. Ce désavantage se manifeste pour toutes les représentations en "virgule fixe".

2° Une autre possibilité, moins courante, est d'employer une représentation de type Décimal Codé Binaire Naturel (*Binary Coded Decimals*). Dans ce cas, chaque chiffre du nombre décimal à représenter est traduite individuellement en binaire sur 4 bits, et le nombre est représenté par la concaténation de ces groupes de 4 bits (représentation BCD compacte). Par exemple, pour $(56{,}3125)_{10}$:

5 6 , 3 1 2 5

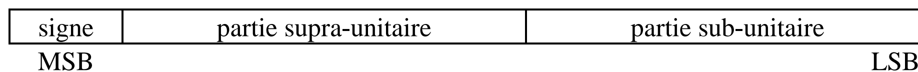
0101 0110, 0011 0001 0010 0101

En version non compacte, chaque quartet qui code un chiffre décimal constitue le quartet le moins significatif d'un octet, l'autre quartet étant 0000 (ex. : 5 → 0101 → 00000101). Le nombre est représenté par la concaténation de ces octets.

Cette technique de représentation est moins économique, mais facilite la traduction.

1.2.2.2. Nombres avec signe :

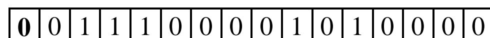
1° Une première possibilité : le premier bit (le plus significatif) est réservé au signe (0 si $N \geq 0$, 1 sinon), les autres contiennent la traduction en binaire de la valeur absolue :



signe	partie supra-unitaire	partie sub-unitaire
MSB		LSB

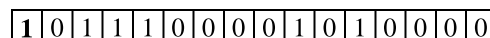
Par exemple, avec 7 bits pour la partie supra-unitaire et 8 bits pour la partie sub-unitaire, nous obtenons :

$(+56{,}3125)_{10} = (+111000{,}0101)_2$ →



0	0	1	1	1	0	0	0	0	1	0	1	0	0	0	0

$(-56{,}3125)_{10} = (-111000{,}0101)_2$ →



1	0	1	1	1	0	0	0	0	1	0	1	0	0	0	0

2° Complément à 1 ($C_1(N)$) : le premier bit (le plus significatif) est réservé au signe (0 si $N \geq 0$, 1 sinon), les autres contiennent la traduction en binaire de la valeur si le nombre est positif, ou les chiffres opposés (0 ↔ 1) à

ceux de la traduction si le nombre est négatif. Par exemple, avec 7 bits pour la partie supra-unitaire et 8 bits pour la partie sub-unitaire, nous obtenons :

$(+56{,}3125)_{10} = (+111000{,}0101)_2 \rightarrow$

0	0	1	1	1	0	0	0	0	1	0	1	0	0	0	0

$(-56{,}3125)_{10} = (-111000{,}0101)_2 \rightarrow$

1	1	0	0	0	1	1	1	1	0	1	0	1	1	1	1

3° Complément à 2 ($C_2(N)$, ou complément **vrai**). Le premier bit (le plus significatif) est réservé au signe (0 si $N \geq 0$, 1 sinon). Considérons que n des bits suivants sont réservés à la partie supra-unitaire des nombres. Alors la représentation — signe mis à part — contient la traduction en binaire de la valeur si le nombre est positif, ou la différence entre 2^n et le résultat de cette traduction si le nombre est négatif. Par exemple, avec 7 bits pour la partie supra-unitaire et 8 bits pour la partie sub-unitaire, nous obtenons :

$(+56{,}3125)_{10} = (+111000{,}0101)_2 \rightarrow$

0	0	1	1	1	0	0	0	0	1	0	1	0	0	0	0

$(-56{,}3125)_{10} = (-111000{,}0101)_2 \rightarrow 2^7 - 111000{,}0101 = 1000111{,}10110000 =$

1	1	0	0	0	1	1	1	1	0	1	1	0	0	0	0

Le plus grand nombre positif représentable : 01111111.11111111 → +127,99609375.
Le plus petit nombre positif représentable : 00000000.00000001 → +0,00390625.
Le plus grand nombre négatif représentable : 11111111.11111111 → -0,00390625.
Le plus petit nombre négatif représentable : 10000000.00000000 → -128.
Ecart minimal entre deux nombres représentables : 0,00390625 (constant).

4° Codage par excédent (employé pour des nombres sans partie fractionnaire) : la représentation contient la traduction en binaire de la somme entre le nombre à représenter et une valeur positive fixe (choisie telle que le résultat soit toujours positif pour les nombres qu'on veut représenter). Par exemple, avec 8 bits, par excédent à 128 ($= 2^7$), nous obtenons :

$(+56)_{10} = (+111000)_2 \rightarrow 2^7 + 111000 = 10111000 =$

1	0	1	1	1	0	0	0

$(-56)_{10} = (-111000)_2 \rightarrow 2^7 - 111000 = 01001000 =$

0	1	0	0	1	0	0	0

Nous constatons que le premier bit ne correspond plus à la convention de signe employée jusqu'ici.

1.2.3. Représentations en "virgule flottante"

Les nombres sont d'abord mis sous forme normale :

$$N = (\pm 0, a_{-1}a_{-2}a_{-3}\cdots) \times b^{\pm n}, \text{ avec } a_{-1} \neq 0 \ (\textit{normalisation}).$$

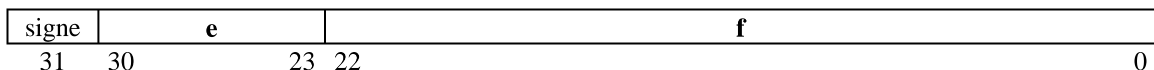
$a_{-1}a_{-2}a_{-3}\cdots$ s'appelle mantisse et n exposant.

Par exemple :

$(\pm 56{,}3125)_{10} = (+111000{,}0101)_2 = \pm 0{,}1110000101 \times 2^6$.

Des bits sont réservés pour représenter le signe, l'exposant et la mantisse.

1° En simple précision (IEEE 754), 32 bits sont employés pour la représentation. Ainsi, le nombre $1,\mathbf{f} \times 2^{\mathbf{e}-127}$ est représenté sous la forme suivante :



Par exemple :

1 10000001 01000000000000000000000 représente :

signe = 1 ⇒ nombre négatif
$\mathbf{e} - 127 = (10000001)_2 - 127 = 129 - 127 = 2$
$\mathbf{f} = (0{,}01)_2 = 0{,}25$
donc le nombre représenté est $-1{,}25 \times 2^2 = -5$.

$+0{,}25 = (0{,}01)_2$ est représenté par :

nombre positif ⇒ signe = 0
$(0{,}01)_2 = 1{,}0 \times 2^{-2} = 1{,}0 \times 2^{125-127}$

donc +0,25 est représenté par 0 01111101 00000000000000000000000

2° En double précision (IEEE 754), 64 bits sont employés pour la représentation. Le nombre $1,\mathbf{f} \times 2^{\mathbf{e}-1023}$ est représenté sous la forme suivante :

signe	e		f	
63	62	52	51	0

Interprétation complète des codes possibles :

e	**f**	**représente**
0	0	±0
0	≠ 0	$\pm 0,f \times 2^{-127}$ ou* $\pm 0,f \times 2^{-1023}$
$0 < e < e_{max}$	∀ f	$\pm 1,f \times 2^{e-127}$ ou* $\pm 1,f \times 2^{e-1023}$
e_{max} (255 ou* 2047)	0	±∞
e_{max} (255 ou* 2047)	≠ 0	NaN (*Not a Number*)

(*simple ou double précision)

NaN est le résultat, par exemple, de $\sqrt{-1}$ ou de log(−1) ; le résultat d'une opération dont un des arguments est NaN doit être NaN. ±∞ est le résultat, par exemple, des divisions par 0 ou de log(0) ; ±∞ peut intervenir dans des calculs, par exemple 1/±∞ = ±0.

1.3. Arithmétique en binaire

1.3.1. Nombres en binaire, sans signe

1.3.1.1. L'addition — exemples (nombres représentés sur 8 bits) :

1° pas de retenue (*transport* du MSB)

```
  0100 1010   +
  1000 1100
 ----------
  1101 0110
```

2° retenue (transport du MSB) ⇒ dépassement du format

```
  0100 1010   +
  1100 1100
 ----------
 10001 0110
```

1.3.1.2. La soustraction (A - B) — exemples (nombres représentés sur 8 bits) :

1° A ≥ B : pas de retenue (transport vers le MSB)

```
  1000 1010   -
  0100 0100
 ----------
  0100 0110
```

2° A < B : retenue (transport vers le MSB) ⇒ résultat négatif, à représenter sur *plus de 8 bits*

```
  0100 0100   -        68  -
  1000 1010           138
 ----------          -----
  1011 1010           - 70
```

$(70)_{10} = (0100\ 0110)_2$, $(-70)_{10} = (1011\ 1010)_2$ en complément à 2, sur 8 bits

mais :

```
  0100 0100   -        68  -
  1100 1010           202
 ----------          -----
  0111 1010          - 134
```

$(134)_{10} = (1000\ 0110)_2$, $(-134)_{10}$ ne peut pas être représenté en complément à 2 sur 8 bits, 9 bits sont nécessaires : $(-134)_{10} = (\mathbf{1}\ 0111\ 1010)_2$

1.3.1.3. La multiplication — exemple (facteurs représentés sur 8 bits) :

```
1°        00101011 ×
          00001001
      ----------------
          00101011
     00101011
  --------------------
       110000011(9 bits nécessaires)

2°                11111111 ×
                  11111111
              ----------------
                  11111111
                 11111111
                11111111
               11111111
              11111111
             11111111
            11111111
           11111111
      ------------------------
         1111111000000001          (16 bits nécessaires)
```

Il faut réserver au produit le double du nombre de bits réservés aux facteurs.
Cas particulier : multiplication par 2^n = déplacement à gauche de *n* positions.

1.3.1.4. La division — exemple (dividende représenté sur 8 bits, diviseur sur 4 bits) :

```
10000101 | 1001
 1001    |------
-----    | 1110
  1111   |
  1001
  -----
   1100
   1001
   -----
    111
```

correspond à :

$133 = 9 \times 14 + 7$

Cas particulier : division par 2^n = déplacement à droite de *n* positions.

1.3.2. Nombres en complément à 2

1.3.2.1. Obtenir le complément à 2 (pour nombres entiers) :

Nous considérons que la représentation du nombre se fait sur *n* bits, dont 1 bit de signe.
1° $C_2(N) = 2^n - N$, par exemple
positif → négatif : 0100 1000 → 1 0000 0000 - 0100 1000 = 1011 1000 (signe inclus)
négatif → positif : 1011 1000 → 1 0000 0000 - 1011 1000 = 0100 1000 (signe inclus)

2° $C_2(N) = C_1(N) + 1$, le $C_1(N)$ étant obtenu en inversant toutes les chiffres de la représentation en binaire du nombre (0 ↔ 1), par exemple
positif → négatif : 0100 1000 → 1011 0111, +1 → 1011 1000 (signe inclus)
négatif → positif : 1011 1000 → 0100 0111, +1 → 0100 1000 (signe inclus)

1.3.2.2. L'addition — exemples (nombres représentés sur 8 bits, dont 1 bit de signe) :

1° pas de retenue, pas de dépassement du format

```
  0100 1010  +
  0010 1100
  ---------
  0111 0110
```

⇒ résultat correct

2° pas de retenue, *mais dépassement du format*

```
  0100 1010  +
  0100 1100
  ---------
  1001 0110
```

⇒ résultat incorrect sur 8 bits (la somme de deux nombres positifs ne peut pas être un nombre négatif)

3° retenue, *mais pas de dépassement du format*

```
  0100 1010  +
  1100 1100
  ---------
  0001 0110
```

⇒ résultat correct (la retenue n'est pas prise en compte)

4° retenue et dépassement du format

```
  1000 1010  +
  1100 1100
  ---------
  0101 0110
```

⇒ résultat incorrect sur 8 bits (la somme de deux nombres négatifs ne peut pas être un nombre positif)

Observation : le format n'est jamais dépassé quand les termes ont des signes opposés.

Règle pour savoir si un dépassement a eu lieu :

dépassement ⇔ (transport vers le MSB ≠ transport du MSB).

Pour la soustraction (A - B), on construit le complément de B et on additionne le résultat à A.

1.3.3. Nombres en DCBN compacte

Les quartets sont entre 0000 (correspondant à 0)et 1001 (correspondant à 9).

1.3.3.1. L'addition — exemples (nombres à deux chiffres décimales, représentés sur 8 bits) :

En général, nous effectuons l'addition en binaire sans prendre en compte le caractère particulier du codage, et ensuite nous corrigeons le résultat de la façon suivante : s'il y a une retenue dans le quartet ou si le quartet contient une valeur supérieure à 9 (1001), on ajoute 6 (0110) au quartet (et on propage l'éventuelle nouvelle retenue vers le quartet supérieur).

Par exemple :

```
  0010 1001  +        29  +
  0100 0100           44
  ---------          ---
  0110 1101           73
```

quartet inférieur supérieur à 1001, donc la correction doit être appliquée → 0111 0011 (73),

ou

```
  0000 1001  +         9  +
  0100 1000           48
  ---------          ---
  0101 0001           57
```

retenue obtenue dans le quartet inférieur, la correction doit être appliquée → 0101 0111 (57).

1.3.3.2. La soustraction (A - B) — exemples (nombres à 2 chiffres décimales, sur 8 bits) :

En général, nous effectuons l'addition en binaire sans prendre en compte le caractère particulier du codage, et ensuite nous corrigeons le résultat de la façon suivante : nous soustrayons 6 (0110) de tout quartet qui a demandé une retenue.

Par exemple :

```
    0010  0000   -        20  -
    0000  1001             9
  ------------        ------
    0001  0111            11
```

retenue demandée par le quartet inférieur, donc la correction doit être appliquée à ce quartet → 0001 0001 (11).

1.4. Représentation des caractères

Standard ASCII 8 (8 bits) :

Dec	➔	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
↓	Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	nul	□		0	@	P	`	p	Ç	É	á	□	□	□	α	≡
1	1	☺	□	!	1	A	Q	a	q	ü	æ	í	□	□	□	β	±
2	2	□	×	"	2	B	R	b	r	é	Æ	ó	□	□	□	Γ	≥
3	3	♥	!!	#	3	C	S	c	s	â	ô	ú	□	□	□	π	≤
4	4	♦	¶	$	4	D	T	d	t	ä	ö	ñ	□	□	□	Σ	⌠
5	5	♣	§	%	5	E	U	e	u	à	ò	Ñ	□	□	□	σ	⌡
6	6	♠	□	&	6	F	V	f	v	å	û	□	□	□	□	µ	÷
7	7	•	□	'	7	G	W	g	w	ç	ù	□	□	□	□	τ	≈
8	8	□	↑	(	8	H	X	h	x	ê	ÿ	¿	□	□	□	Φ	°
9	9	○	↓	)	9	I	Y	i	y	ë	Ö	□	□	□	□	θ	•
10	A	□	→	*	:	J	Z	j	z	è	Ü	¬	□	□	□	Ω	•
11	B	□	←	+	;	K	[	k	{	ï	¢	½	□	□	□	δ	√
12	C	□	□	,	<	L	\	l	¦	î	£	¼	□	□	□	∞	n
13	D	□	Ö	–	=	M	]	m	}	ì	¥	¡	□	□	□	φ	²
14	E	□	s	.	>	N	^	n	~	Ä	□	«	□	□	□	∈	z
15	F	⌖	t	/	?	O	_	o	•	Å	□	»	□	□	□	∩	□

Exemple : A = 41 en hexadécimal (premier quartet sur la deuxième ligne, deuxième quartet sur la deuxième colonne)
A = 65 en décimal (= 64 + 1, somme entre la valeur sur la première ligne et celle sur la première colonne)

2. Circuits combinatoires

2.1. Algèbre de Boole et opérateurs combinatoires

Variable logique (binaire) : $x \in \{0, 1\}$
Fonction logique à n variables : $f : \{0, 1\}^n \rightarrow \{0, 1\}$
Nombre de fonctions logiques à n variables : 2^{2^n}

Opérateur de négation (NON, *NOT*) :

x	$\overline{x}$
0	1
1	0

Opérateurs OU et NON-OU (*OR* et *NOR*)

x	y	$x + y$	$\overline{x+y}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

Opérateurs ET et NON-ET (*AND* et *NAND*)

x	y	$x \cdot y$	$\overline{x \cdot y}$
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

Les opérateurs ET, OU, NON-ET et NON-OU se généralisent pour un nombre quelconque de variables, par exemple :

x	y	z	$x \cdot y z$	$\overline{x \cdot y \cdot z}$
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	0

Opérateur OU-EXCLUSIF (somme modulo 2)

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

Conséquences des définitions :

$\overline{\overline{x}} = x$

$x + 0 = x$

$x + 1 = 1$

$x + x = x$

$x + \overline{x} = 1$

$x\,0 = 0$

$x\,1 = x$

$x\,x = x$

$x\,\overline{x} = 0$

Commutativité :

$x + y = y + x$

$x\,y = y\,x$

Associativité :

$(x + y) + z = x + (y + z)$

$(xy)z = x(yz)$

Distributivité :

$x(y + z) = xy + xz$

$x + yz = (x + y)(x + z)$

Lois de De Morgan :

$$\overline{x+y} = \overline{x} \cdot \overline{y}$$
$$\overline{x \cdot y} = \overline{x} + \overline{y}$$

x	y	$\overline{x+y}$	$\overline{x} \cdot \overline{y}$	$\overline{x \cdot y}$	$\overline{x} + \overline{y}$
0	0	1	1	1	1
0	1	0	0	1	1
1	0	0	0	1	1
1	1	0	0	0	0

2.2. Synthèse combinatoire

2.2.1. Formes canoniques

Considérons la fonction logique $f(x, y, z, t)$ donnée dans le tableau suivant :

x	y	z	t	f
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

Forme canonique disjonctive (f.c.d.) :

$$f(x, y, z, t) = \overline{x}\,\overline{y}\,\overline{z}\,\overline{t} + \overline{x}\,\overline{y}z\overline{t} + \overline{x}y\overline{z}t + x\overline{y}\,\overline{z}\,\overline{t} + x\overline{y}\,\overline{z}t + x\overline{y}z\overline{t} + x\overline{y}zt + xy\overline{z}t$$

Forme canonique conjonctive (f.c.c.) :

$$f(x, y, z, t) = (x + y + z + \overline{t})(x + y + \overline{z} + \overline{t})(x + \overline{y} + z + t)(x + \overline{y} + \overline{z} + t)$$
$$(x + \overline{y} + \overline{z} + \overline{t})(\overline{x} + \overline{y} + z + t)(\overline{x} + \overline{y} + \overline{z} + t)(\overline{x} + \overline{y} + \overline{z} + \overline{t})$$

2.2.2. Simplification de l'écriture d'une fonction logique

2.2.2.1. Méthode algébrique

On procède par des regroupements qui permettent (en utilisant les propriétés des opérateurs logiques) d'éliminer des variables :

$f = \overline{x}\,\overline{y}\,\overline{z}\,\overline{t} + \overline{x}\,\overline{y}z\overline{t} + \overline{x}y\overline{z}t + x\overline{y}\,\overline{z}\,\overline{t} + x\overline{y}\,\overline{z}t + x\overline{y}z\overline{t} + x\overline{y}zt + xy\overline{z}t$

$f = \overline{x}\,\overline{y}\,\overline{t}\,(\overline{z} + z) + \overline{x}y\overline{z}t + x\overline{y}\,\overline{z}\,(\overline{t} + t) + x\overline{y}z(\overline{t} + t) + xy\overline{z}t$

$f = \overline{x}\,\overline{y}\,\overline{t} + \overline{x}y\overline{z}t + x\overline{y}\,\overline{z} + x\overline{y}z + xy\overline{z}t$

$f = \overline{x}\,\overline{y}\,\overline{t} + x\overline{y}\,(\overline{z} + z) + (\overline{x} + x)y\overline{z}t$

$f = (\overline{x}\,\overline{t} + x)\overline{y} + y\overline{z}t$

$f = (\overline{t} + x)\overline{y} + y\overline{z}t$

donc $f(x, y, z, t) = \overline{y}\,\overline{t} + x\overline{y} + y\overline{z}t.$

2.2.2.2. Méthode de Karnaugh

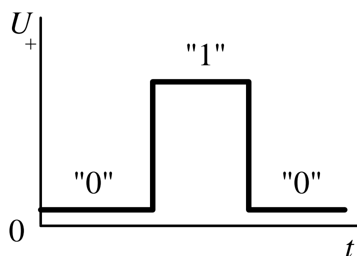
Les valeurs de la fonction sont introduites dans le diagramme de Karnaugh et des regroupements sont effectués, permettant de simplifier l'expression de la fonction :

zt \ xy	00	01	11	10
00	1	0	0	1
01	0	1	1	1
11	0	0	0	1
10	1	0	0	1

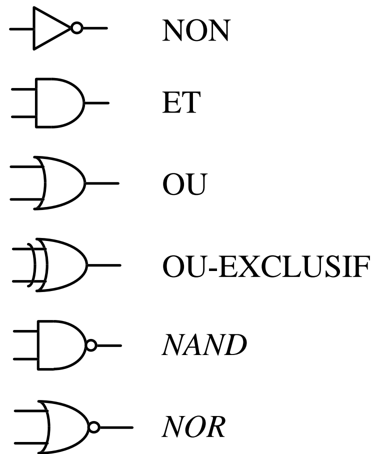
donc $f(x, y, z, t) = y\overline{z}t + x\overline{y} + \overline{y}\,\overline{t}$.

2.2.3. Synthèse avec circuits élémentaires (portes)

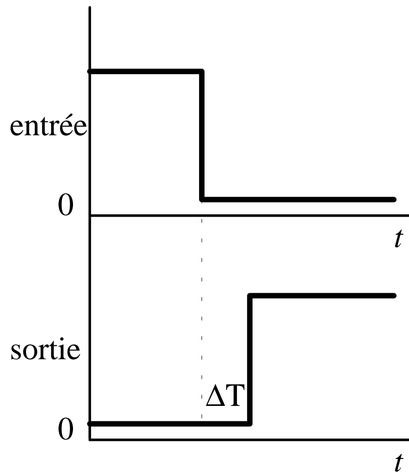
Convention positive : le potentiel positif le plus élevé est associé à la valeur logique 1



Symboles employés pour les circuits élémentaires :



Délais de propagation : par exemple, pour une porte NON



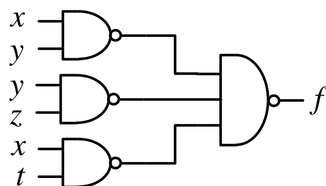
Toute fonction logique peut être obtenue soit avec des circuits *NAND*, soit avec des circuits *NOR*. Par exemple, avec *NAND* :

1° $\overline{x} = \overline{x \cdot x}$

2° $f = x\,y + y\,z + x\,t$

$f = \overline{\overline{x \cdot y + y \cdot z + x \cdot t}}$

$f = \overline{\overline{xy} \cdot \overline{yz} \cdot \overline{xt}}$

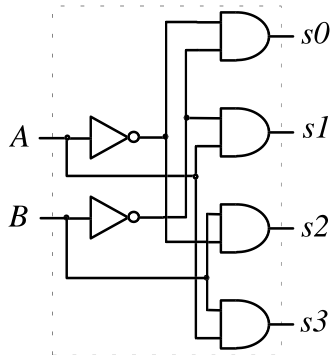


2.2.4. Synthèse avec mémoires mortes (*Read Only Memory, ROM*)

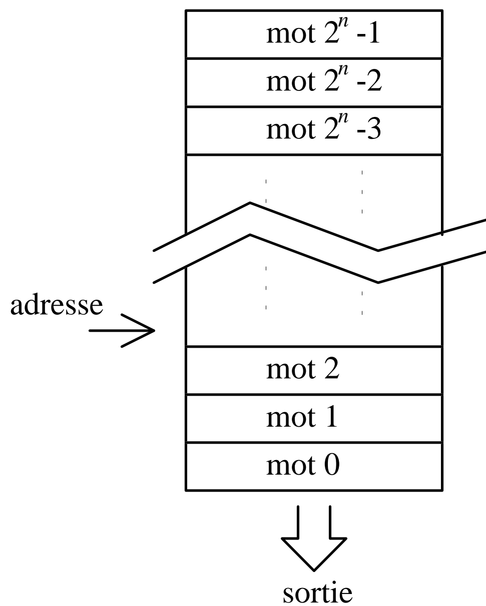
Décodeurs : n entrées d'adresse et 2^n sorties, dont une seule est active à la fois

entrées		sorties			
A	*B*	*s3*	*s2*	*s1*	*s0*
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

donc : $s0 = \overline{B} \cdot \overline{A}$, $s1 = \overline{B} \cdot A$, $s2 = B \cdot \overline{A}$, $s3 = B \cdot A$.



ROM à n bits d'adresse ⇒ capacité 2^n mots

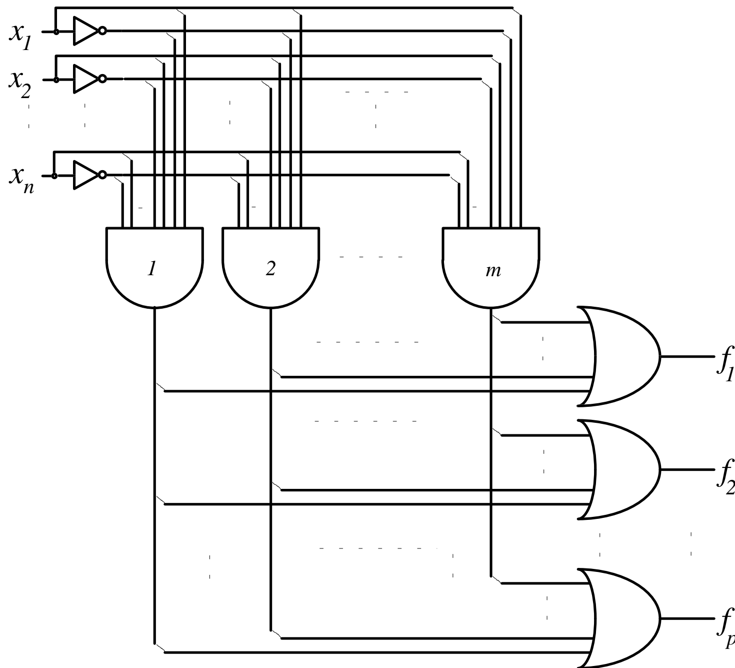


n bits d'adresse, mots de m bits ⇒ au maximum m fonctions à n variables

x_1	…	x_n	f_1	…	f_m
0	…	0	1	…	0
0	…	1	0	…	0
… …	… …	… …	… …	… …	… …
1	…	1	0	…	1

2.2.5. Synthèse avec réseaux programmables (*Programmable Logic Array, PLA*)

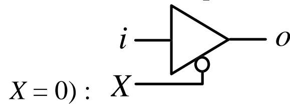
Directement à partir de la f.c.d. :

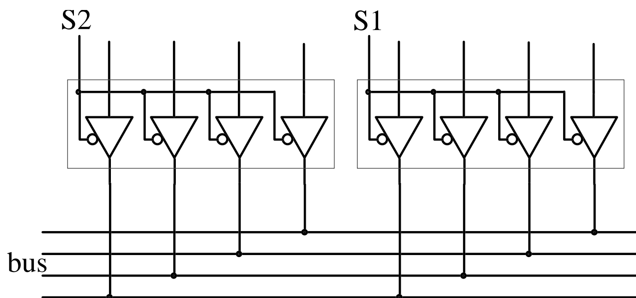


Programmation : les fusibles sont brûlés là où il est nécessaire d'éliminer des dépendances.

2.3. Bus et circuits "à trois états" (*Tri-State Logic*)

Troisième état : équivalent à la déconnexion de la sortie du circuit du fil. Symbole (ici, l'entrée X est active quand $X = 0$) :





Condition qui assure l'absence de conflits sur le bus : S1 + S2 = 0.

3. Circuits séquentiels

Circuits combinatoires : sortie (t) = F (entrée (t)) (absence de *mémoire*).
Circuits séquentiels : sortie (t) = F (entrée (t), entrée (t-1), … entrée (0)).

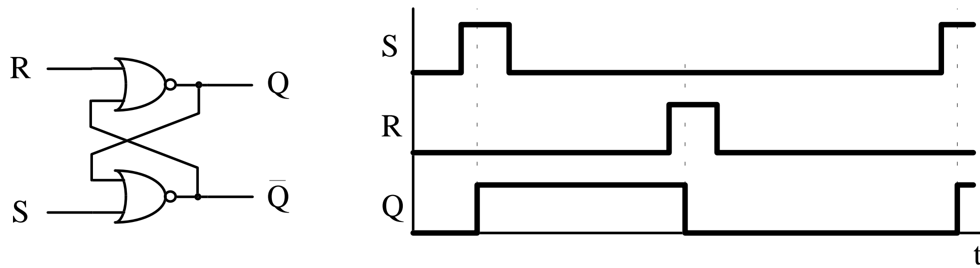
3.1. Bascules

3.1.1. Bascules RS

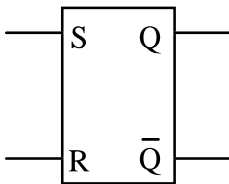
Q_{n-1} = état antérieur, Q_n = nouvel état

R	S	Q_n
0	0	Q_{n-1}
0	1	1
1	0	0
1*	1*	?

*combinaison interdite



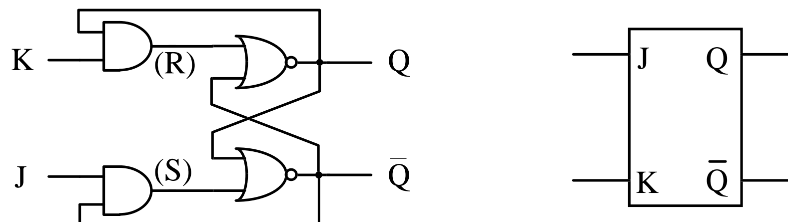
Condition de bon fonctionnement : R·S = 0.



3.1.2. Bascules JK

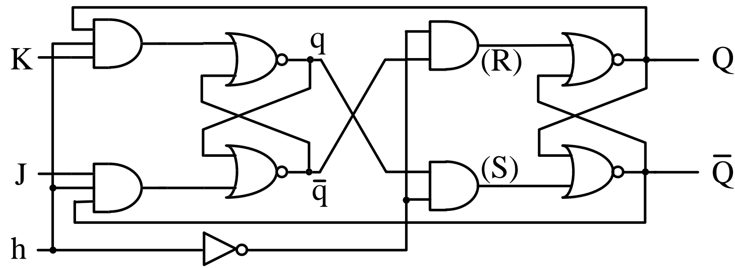
But : éviter les combinaisons interdites en entrée

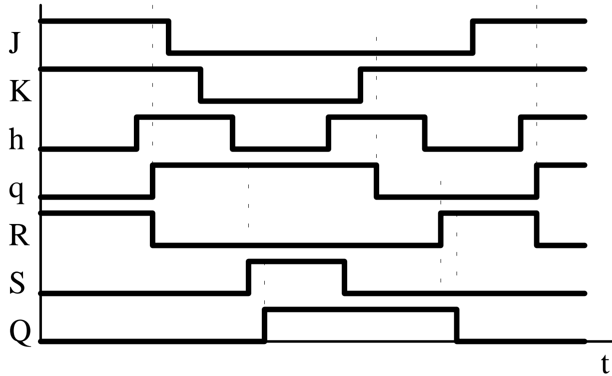
K	J	Q_n
0	0	Q_{n-1}
0	1	1
1	0	0
1	1	$\overline{Q_{n-1}}$

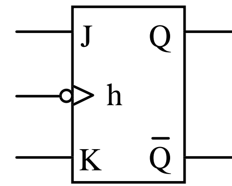


$R \cdot S = (K \cdot Q) \cdot (J \cdot \overline{Q}) = 0 \Rightarrow$ la condition de bon fonctionnement de la bascule RS est satisfaite.

Bascule JK synchrone "maître-esclave" :



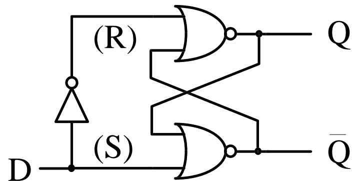




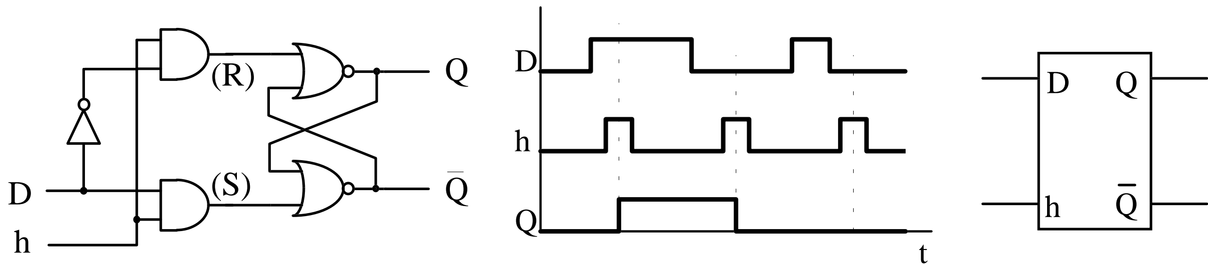
Le changement d'état se manifeste à la sortie (Q) après que le signal d'horloge revient à 0.

3.1.3. Bascules D

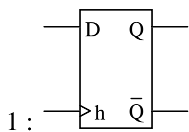
D	Q_n
0	0
1	1



Bascule D synchronisée par un horloge *h* (change d'état quand l'entrée *D* est "visible", c'est à dire quand *h* est à 1) :



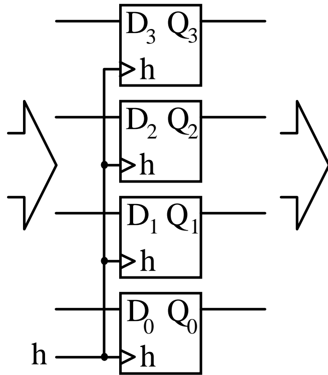
Sont utilisées couramment les bascules D qui changent d'état durant la montée du signal d'horloge de 0 à 1 :



3.2. Registres

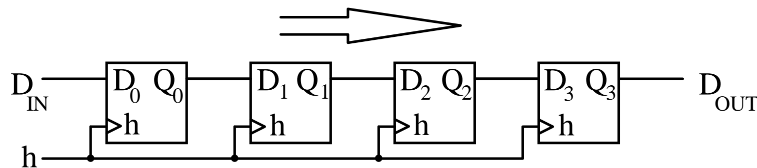
3.2.1. Registres parallèles

(entrée/sortie des données en parallèle)

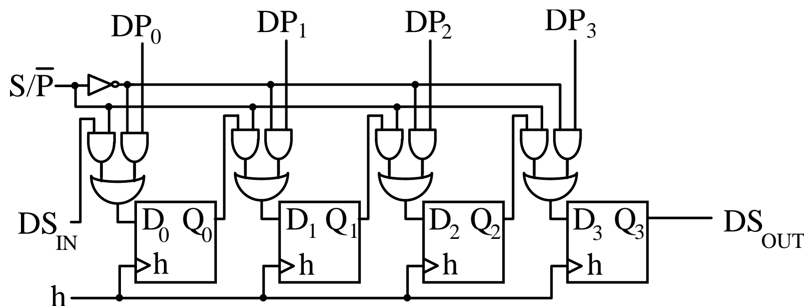


3.2.2. Registres de déplacement

Entrée série, sortie série ou parallèle :

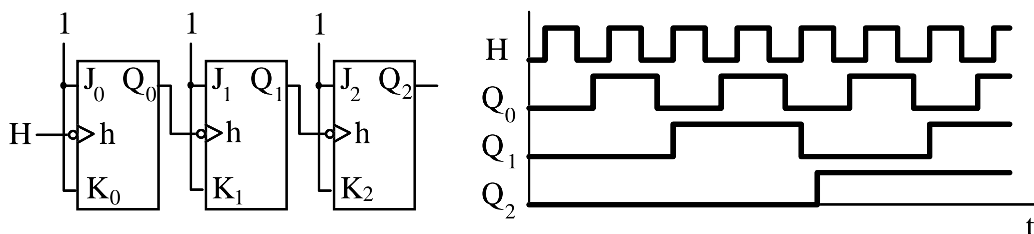


Entrée série ou parallèle, sortie série ou parallèle :



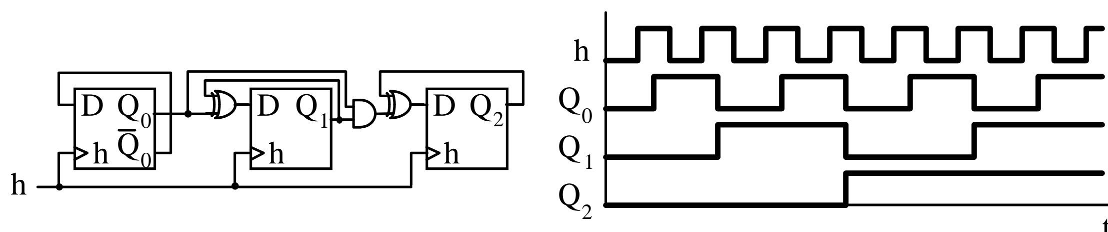
3.3. Compteurs

Compteur binaire asynchrone avec bascules JK "maître-esclave" :



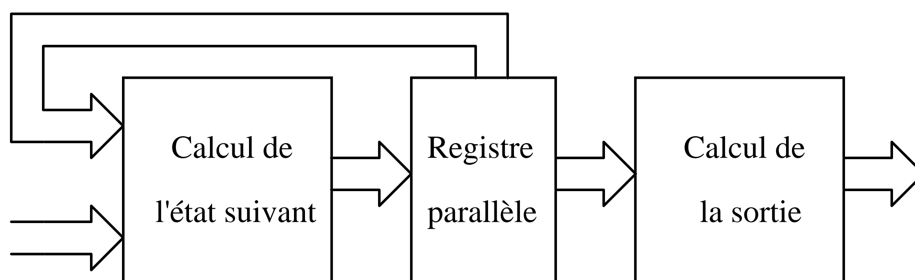
Q_2^{n-1}	Q_1^{n-1}	Q_0^{n-1}	Q_2^n	Q_1^n	Q_0^n
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

Compteur binaire synchrone avec bascules D :



3.4. Synthèse d'automates synchrones

$$\begin{cases} \text{Etat}(t_n) = F[\text{Etat}(t_{n-1}), \text{Entrée}(t_{n-1})] \\ \text{Sortie}(t_n) = G[\text{Etat}(t_n)] \end{cases}$$

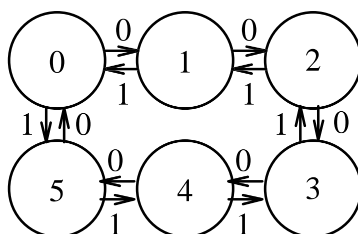


Exemple : synthèse d'un compteur 0 à 5, en ordre croissant/décroissant, synchrone

entrée : sens de comptage (S)

sortie : code en binaire du chiffre

1° construction du graphe de transition



2° codage des états et construction de la table de transition et de sortie

code = valeur en binaire (4 bits nécessaires) du nombre décimal

S	Q_2^{n-1}	Q_1^{n-1}	Q_0^{n-1}	Q_2^n	Q_1^n	Q_0^n
0	0	0	0	0	0	1
0	0	0	1	0	1	0
0	0	1	0	0	1	1
0	0	1	1	1	0	0
0	1	0	0	1	0	1
0	1	0	1	0	0	0
1	0	0	0	1	0	1
1	0	0	1	0	0	0
1	0	1	0	0	0	1
1	0	1	1	0	1	0
1	1	0	0	0	1	1
1	1	0	1	1	0	0

3° synthèse des fonctions de transition et des fonctions de sortie

Q_2

Q_1Q_0 \ SQ_2	00	01	11	10
00	0	1	0	1
01	0	0	1	0
11	1	X	X	0
10	0	X	X	0

Q_1

Q_1Q_0 \ SQ_2	00	01	11	10
00	0	0	1	0
01	1	1	0	0
11	0	X	X	1
10	1	X	X	0

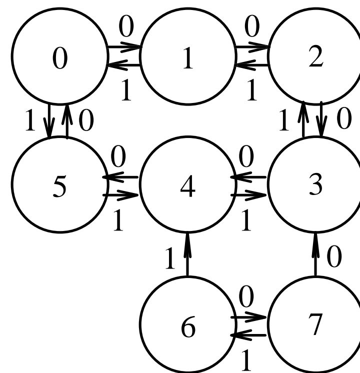
Q_0

Q_1Q_0 \ SQ_2	00	01	11	10
00	1	1	1	1
01	0	0	0	0
11	0	X	X	0
10	1	X	X	1

$$D_2 = \overline{S} \cdot Q_2 \cdot \overline{Q}_0 + S \cdot Q_2 \cdot Q_0 + S \cdot \overline{Q}_2 \cdot \overline{Q}_1 \cdot \overline{Q}_0 + \overline{S} \cdot Q_1 \cdot Q_0$$
$$D_1 = \overline{S} \cdot \overline{Q}_1 \cdot Q_0 + S \cdot Q_2 \cdot \overline{Q}_0 + S \cdot Q_1 \cdot \overline{Q}_0 + \overline{S} \cdot Q_1 \cdot Q_0$$
$$D_0 = \overline{Q}_0$$

4° (éventuellement) construction du graphe de transition complet, modification afin d'assurer l'entrée en fonctionnement normal à la mise sous tension

S	Q_2^{n-1}	Q_1^{n-1}	Q_0^{n-1}	Q_2^n	Q_1^n	Q_0^n
0	0	0	0	0	0	1
0	0	0	1	0	1	0
0	0	1	0	0	1	1
0	0	1	1	1	0	0
0	1	0	0	1	0	1
0	1	0	1	0	0	0
0	1	1	0	1	1	1
0	1	1	1	1	0	0
1	0	0	0	1	0	1
1	0	0	1	0	0	0
1	0	1	0	0	0	1
1	0	1	1	0	1	0
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	0	1	1
1	1	1	1	1	1	0



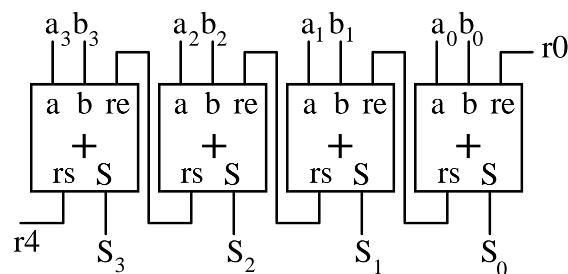
Même si l'état initial ne fait pas partie des états désirés, après au maximum 2 impulsions d'horloge l'automate entre en cycle normal.
5° (éventuellement) re-synthèse à partir du graphe complet

4. Circuits complexes

4.1. Unité Arithmétique et Logique (UAL, *ALU*)

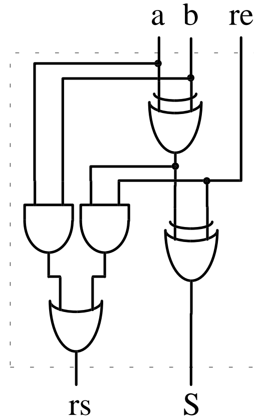
4.1.1. Additionneurs

Addition sur 4 bits :



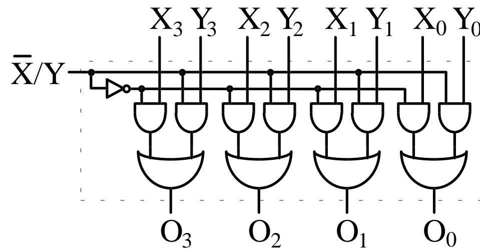
a	b	re	S	rs
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$S = \overline{a} \cdot \overline{b} \cdot re + \overline{a} \cdot b \cdot \overline{re} + a \cdot \overline{b} \cdot \overline{re} + a \cdot b \cdot re = a \oplus b \oplus re$$
$$rs = a \cdot b + re \cdot (a \oplus b)$$

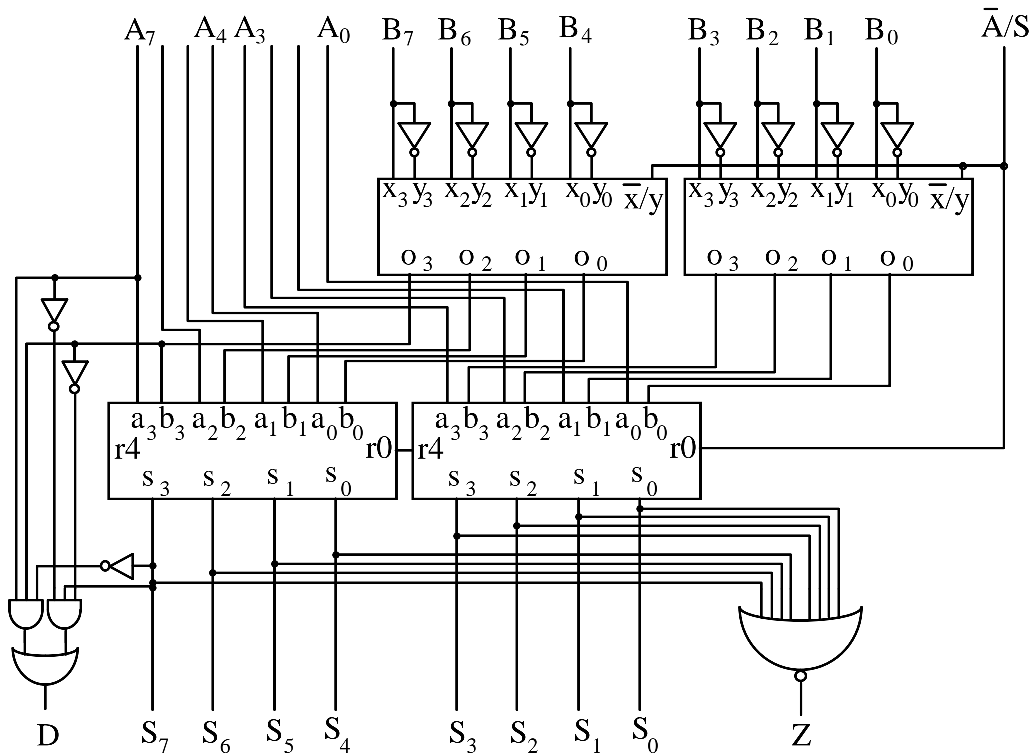


4.1.2. Additionneur/soustracteur

Multiplexeur 2 à 1 (sur 4 bits) :



Additionneur/soustracteur pour nombres représentés en complément à 2 :

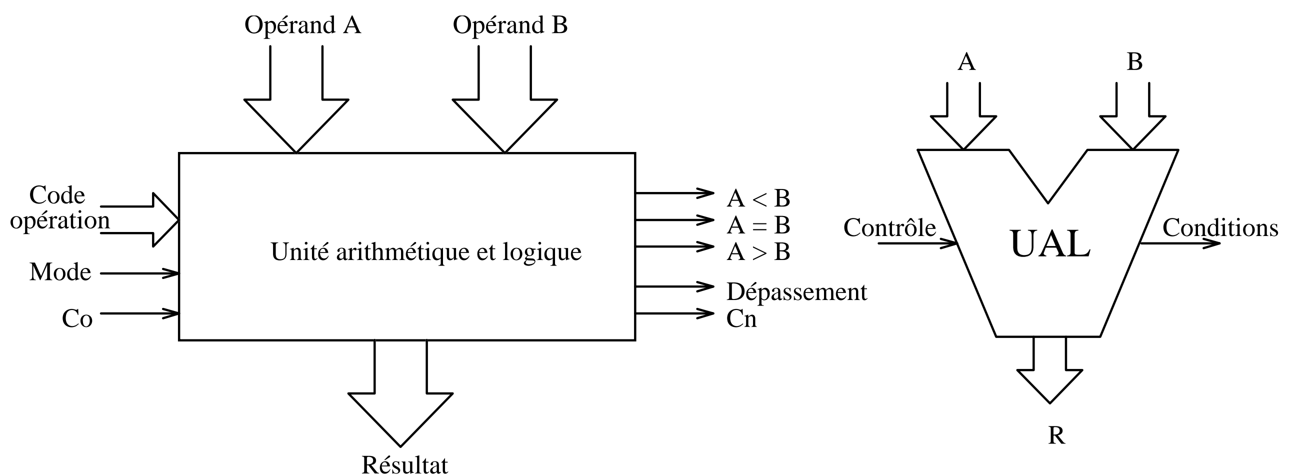


Addition : $\overline{A}/S = 0$, soustraction : $\overline{A}/S = 1$; D = 1 ⇒ débordement (dépassement du format).

Comparaison : $\overline{A}/S = 1$ (soustraction A-B)

A < B ⇔ S7 = 1 et Z = 0
A = B ⇔ Z = 1
A > B ⇔ S7 = 0 et Z = 0.

4.1.2. Unité Arithmétique et Logique



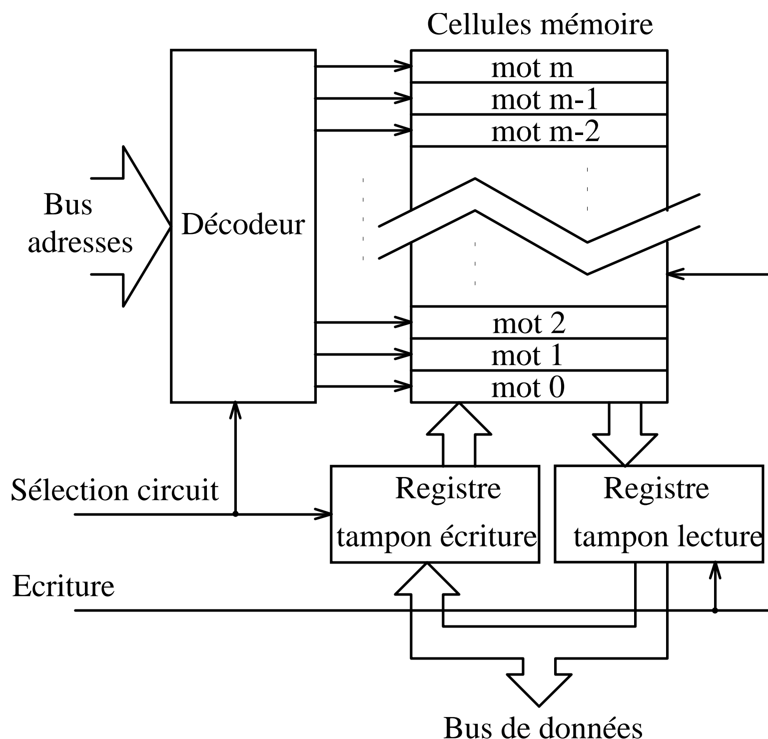
Code opération	Mode = 1 opérations logiques	Mode = 0 — opérations arithmétiques	
		Co = 1	Co = 0
0 0 0 0	$F = \overline{A}$	$F = A$	F = A plus 1
0 0 0 1	$F = \overline{A + B}$	$F = A + B$	F = (A + B) plus 1
0 0 1 0	$F = \overline{A} \cdot B$	$F = A + \overline{B}$	$F = (A + \overline{B})$ plus 1
0 0 1 1	$F = 0$	F = -1 (compl. à 2)	F = 0
0 1 0 0	$F = \overline{A \cdot B}$	$F = A$ plus $A \cdot \overline{B}$	$F = A$ plus $A \cdot \overline{B}$ plus 1
0 1 0 1	$F = \overline{B}$	$F = (A + B)$ plus $A \cdot \overline{B}$	$F = (A + B)$ plus $A \cdot \overline{B}$ plus 1
0 1 1 0	$F = A \oplus B$	F = A moins B moins 1	F = A moins B
0 1 1 1	$F = A \cdot \overline{B}$	$F = A \cdot \overline{B}$ moins 1	$F = A \cdot \overline{B}$
1 0 0 0	$F = \overline{A} + B$	$F = A$ plus $A \cdot B$	$F = A$ plus $A \cdot B$ plus 1
1 0 0 1	$F = \overline{A \oplus B}$	F = A plus B	F = A plus B plus 1
1 0 1 0	$F = B$	$F = (A + \overline{B})$ plus $A \cdot B$	$F = (A + \overline{B})$ plus $A \cdot B$ plus 1
1 0 1 1	$F = A \cdot B$	$F = A \cdot B$ moins 1	$F = A \cdot B$
1 1 0 0	$F = 1$	F = A plus A	F = A plus A plus 1
1 1 0 1	$F = A + \overline{B}$	F = (A + B) plus A	F = (A + B) plus A plus 1
1 1 1 0	$F = A + B$	$F = (A + \overline{B})$ plus A	$F = (A + \overline{B})$ plus A plus 1
1 1 1 1	$F = A$	F = A moins 1	F = A

("‾ ", "·" et "+" désignent respectivement le complément, le ET logique et le OU logique bit par bit ; "plus " désigne l'addition arithmétique).

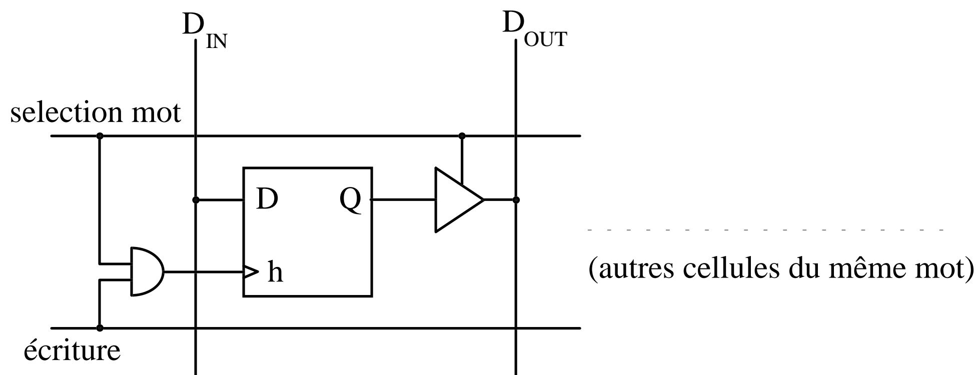
4.2. Mémoire vive (*Random Access Memory, RAM*)

4.2.1. Mémoires RAM statiques (*SRAM*)

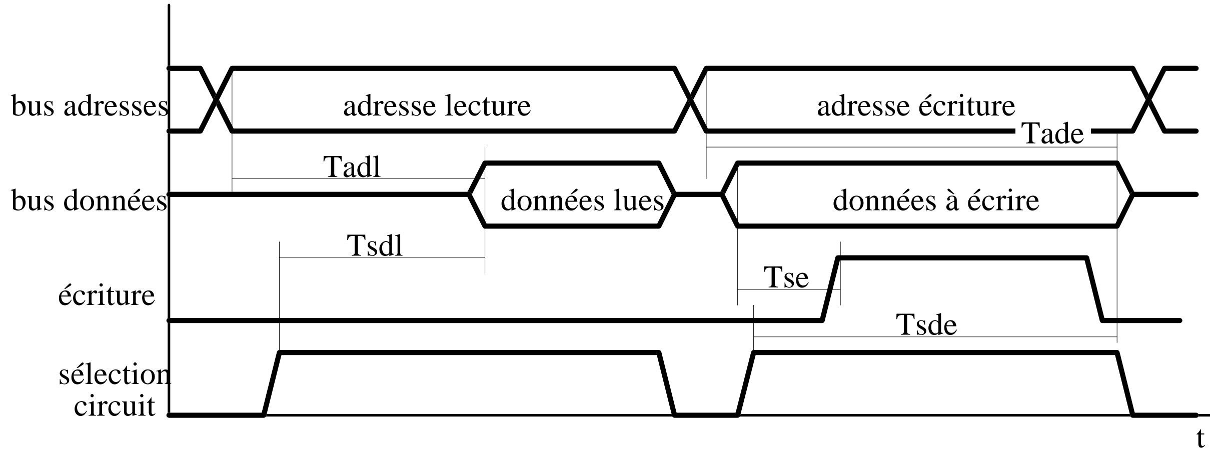
Organisation de principe d'un circuit :



Structure de principe d'une cellule mémoire :



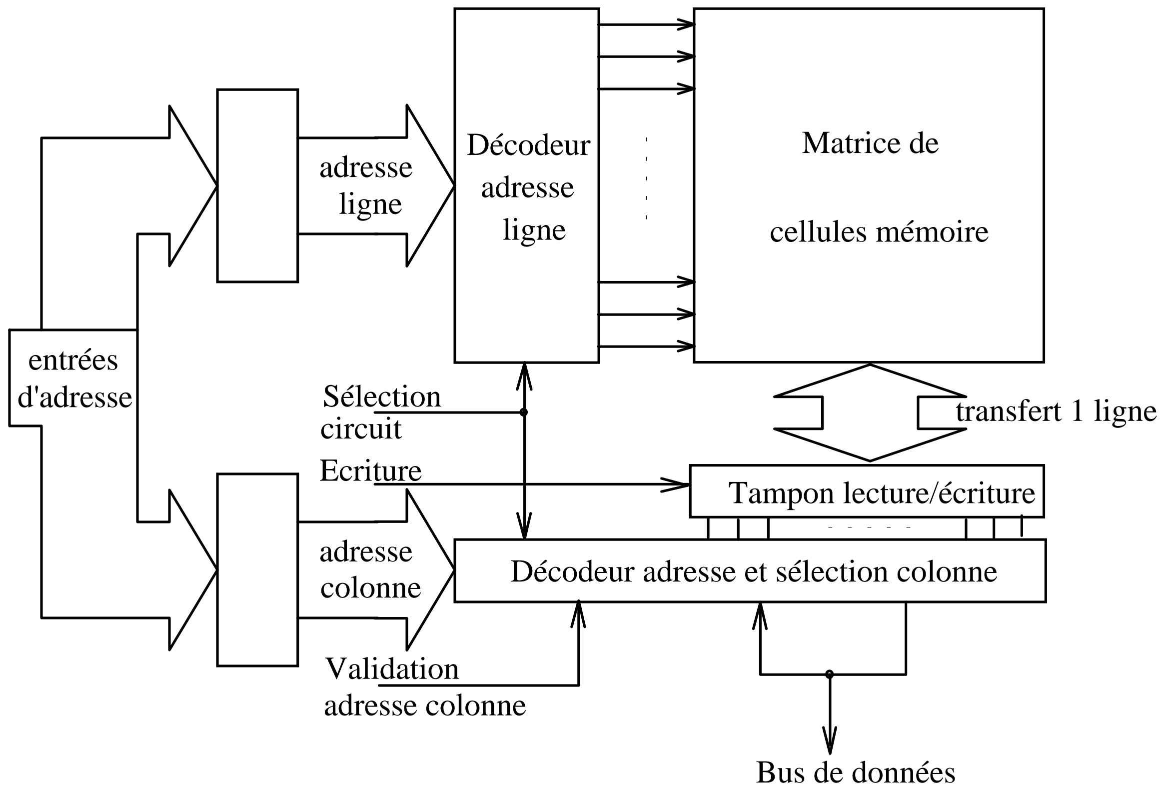
Fonctionnement :



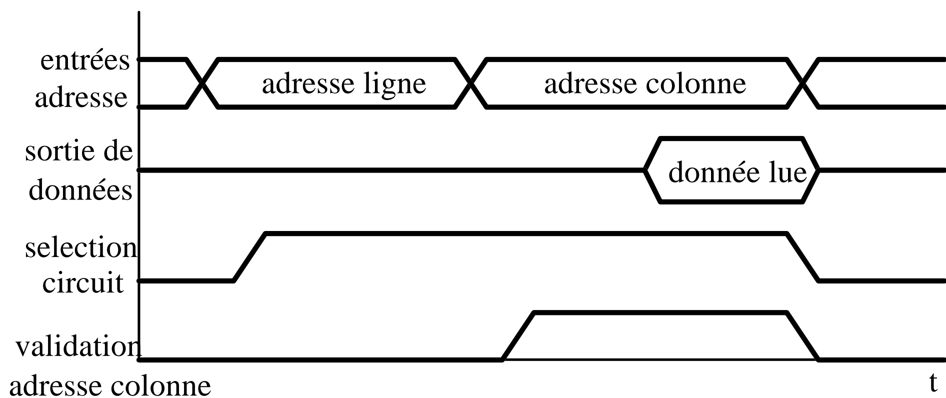
Tadl = temps d'accès en lecture par rapport à la présentation de l'adresse
Tsdl = temps d'accès en lecture par rapport au signal de sélection
Tse = temps de pré-activation des données par rapport au signal d'écriture
Tade = temps d'accès en écriture par rapport à la présentation de l'adresse
Tsdl = temps d'accès en écriture par rapport au signal de sélection

4.2.2. Mémoires RAM dynamiques (*DRAM*)

Organisation de principe d'un circuit :

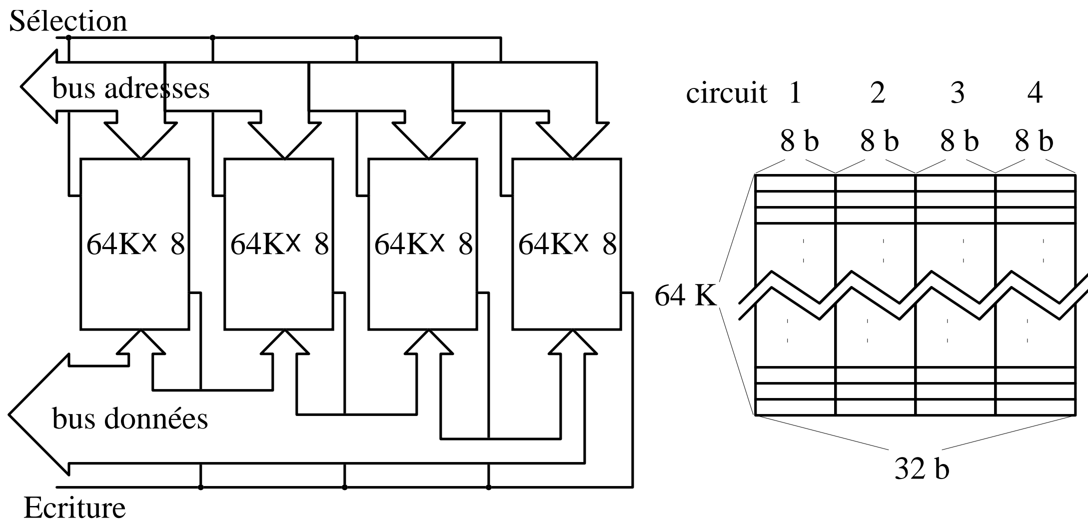


Fonctionnement (cycle de lecture simple) :

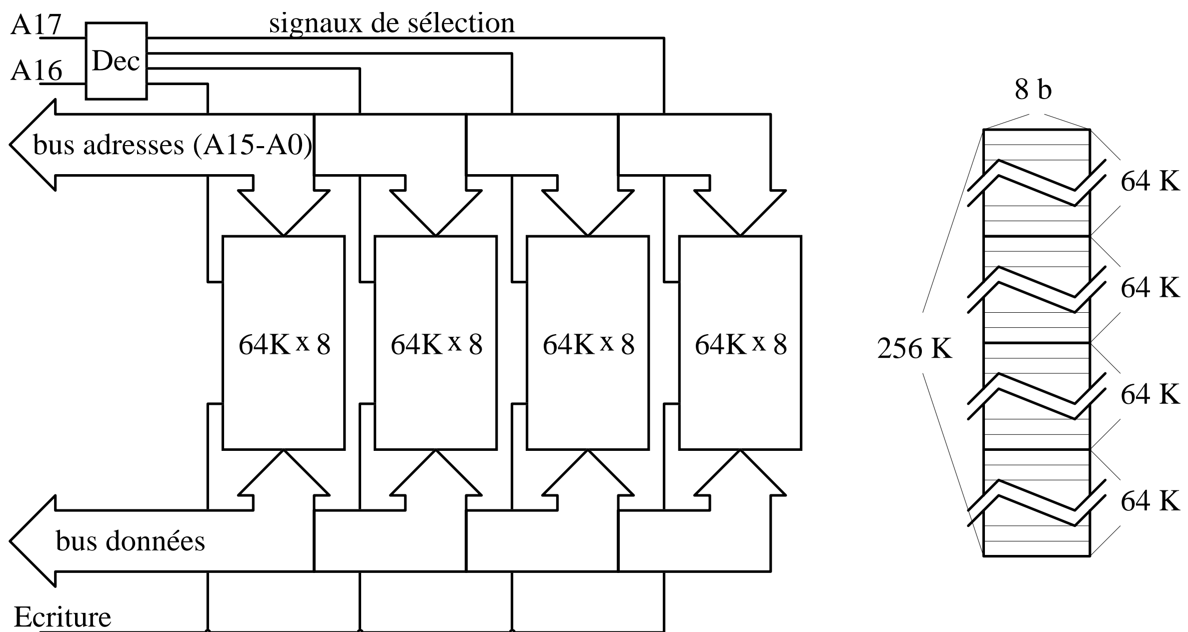


4.2.2. Augmentation de la capacité : utilisation de plusieurs circuits

Augmentation de la taille des mots : $(64K \times 8) \times 4 \rightarrow 64K \times 32$



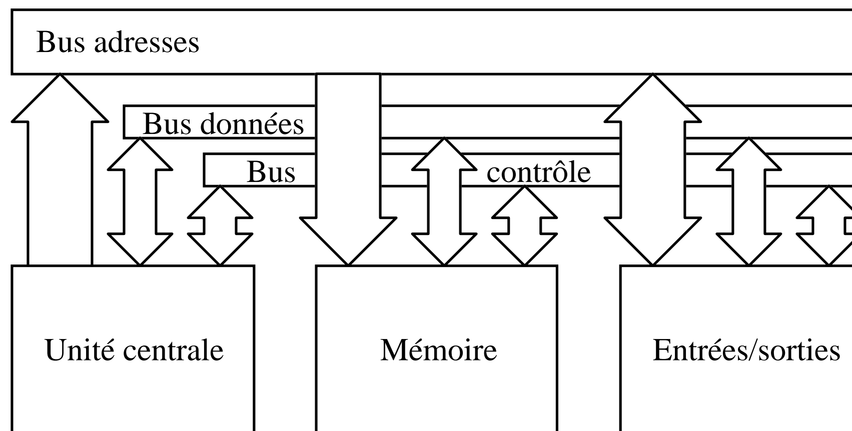
Augmentation du nombre de mots disponibles : $(64K \times 8) \times 4 \rightarrow 256K \times 8$



5. Structure et fonctionnement global d'un ordinateur

5.1. Structure et fonctionnement

5.1.1. Structure simplifiée



Rôles des composantes :

1° Unité centrale (UC) : contrôler toutes les composantes, en exécutant les instructions d'un programme ; effectuer des calculs arithmétiques et logiques ;
2° Mémoire (M) : garder le programme en cours d'exécution et les données associées ;
3° Entrées/sorties (E/S) : relier le système au monde externe à travers les unités périphériques (écran, clavier, disques, bandes magnétiques, réseaux, etc.) ;
4° Bus d'adresse (A) : véhiculer l'adresse mémoire ou d'unité E/S engendrée par l'UC (et dans certains cas par d'autres unités de contrôle) ;
5° Bus de données (D) : véhiculer l'information (instructions, données) entre l'UC, la mémoire et les unités E/S ;
6° Bus de contrôle (C) : véhiculer les signaux employés par l'UC pour le contrôle du système (Adresse mémoire valide, Adresse E/S valide, Lecture/écriture, Attente, Interruption, etc.).

5.1.2. Fonctionnement

5.1.2.1. Exécution d'une instruction

Etapes :

1° l'UC lit le code de l'instruction en mémoire : l'UC sort l'adresse de l'instruction sur le bus d'adresses et active les signaux "Adresse mémoire valide" et "Lecture" ; la mémoire sort sur le bus de données l'information se trouvant à l'adresse indiquée ; l'UC transfère cette information (code d'une instruction) du bus de données vers un registre interne (registre d'instruction) ;
2° l'UC décode le code de l'instruction, qui indique la suite d'opérations à effectuer ;
3° si la lecture d'une donnée se trouvant en mémoire est nécessaire :

l'UC calcule l'adresse de la donnée en mémoire, sort l'adresse sur le bus d'adresses et active les signaux "Adresse mémoire valide" et "Lecture" ; la mémoire sort sur le bus de données l'information se trouvant à l'adresse indiquée ; l'UC transfère ce code du bus de données vers un registre interne ;

si l'écriture d'une donnée en mémoire est nécessaire :

l'UC calcule l'adresse de la donnée en mémoire, sort l'adresse sur le bus d'adresses, transfère la donnée d'un registre interne vers le bus de données et active les signaux "Adresse mémoire valide" et "Ecriture" ; la mémoire inscrit à l'adresse indiquée l'information se trouvant sur le bus de données ;

si une opération arithmétique ou logique doit être effectuée :

l'UC récupère le(s) opérande(s) (en passant éventuellement par les étapes de lecture de données se trouvant en mémoire) et l(es) envoie à l'UAL (interne à l'UC) ; le résultat de l'opération est stocké dans un registre interne ou en mémoire (passage par les étapes d'écriture d'une donnée en mémoire) ;

4° l'UC calcule l'adresse de l'instruction suivante.

5.1.2.2. Exécution des programmes

Les instructions d'un programme sont exécutées successivement, en suivant les branchements conditionnels ou inconditionnels et les appels de procédures.
Les programmes qui composent le système d'exploitation assurent la gestion des ressources (processeur, mémoire, E/S) et font la liaison entre les programmes d'application.

5.2. Architecture et performances

5.2.1. Temps d'exécution

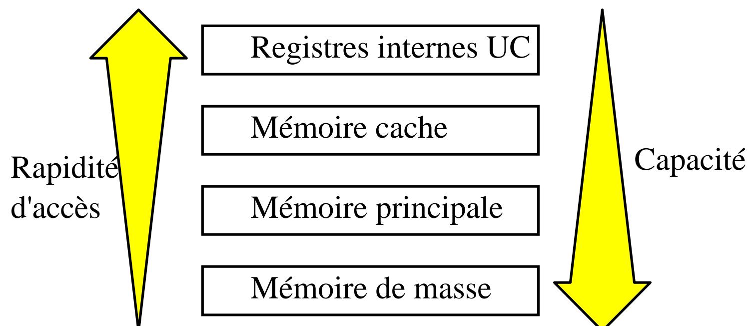
Durée d'exécution d'un programme (heure début – heure fin) = temps passé par l'UC pour exécuter effectivement le programme (temps UC utilisateur) + temps passé pour exécuter des programmes du système d'exploitation + attente des résultats d'opérations d'E/S + temps passé pour exécuter d'autres programmes (fonctionnement "temps partagé", *time-sharing*).
Périodes d'horloge par instruction (*clock Cycles Per Instruction, CPI*) :

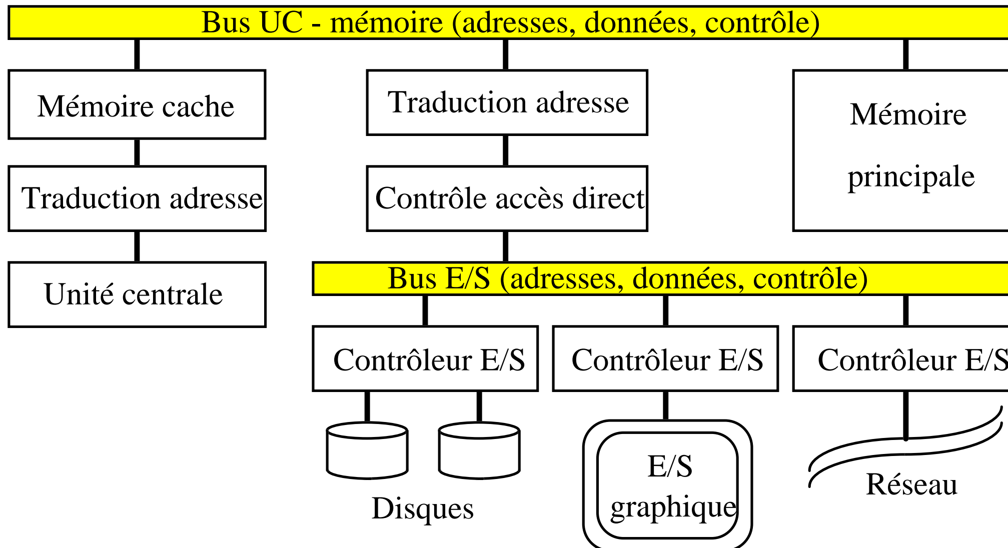
$$CPI = \frac{\text{temps UC utilisateur (en périodes d'horloge)}}{\text{nombre d'instructions du programme}}.$$

Temps UC utilisateur = (période horloge) × (CPI) × (nombre instructions du programme).
Réduction du temps UC utilisateur :
1° augmentation de la fréquence de l'horloge (amélioration de la technologie, simplification des circuits) ;
2° réduction du *CPI* (structure UC mieux adaptée aux instructions, accès mémoire plus rapides) ;
3° réduction du nombre d'instructions (instructions plus puissantes, compilateurs optimisants).

5.2.2. Amélioration des accès mémoire

Hypothèses de proximité :
1° *Proximité temporelle* : un objet déjà référencé le sera à nouveau bientôt.
2° *Proximité spatiale* : les projets proches d'un objet référencé seront bientôt référencés.





5.2.3. Modèles d'exécution et réduction du temps d'exécution

Considérons l'addition A+B=C, avec A, B, C en mémoire. Nous avons plusieurs possibilités, parmi lesquelles :

1° **A + B → C** (l'instruction d'addition agit directement sur des opérandes en mémoire et retourne le résultat en mémoire)

2° A → (R) (une première instruction ramène A dans un registre UC)
(R) + B → C (l'instruction d'addition agit sur un opérande en registre et un autre en mémoire et retourne le résultat en mémoire)

3° **A + B → (R)** (l'instruction d'addition agit directement sur des opérandes en mémoire et retourne le résultat dans un registre UC)
(R) → C (une deuxième instruction envoie le résultat en mémoire)

4° A → (R1) (une première instruction ramène A dans un registre UC)
(R1) + B → (R2) (l'instruction d'addition agit sur un opérande en registre et un autre en mémoire et retourne le résultat dans un autre registre UC)
(R2) → C (une troisième instruction envoie le résultat en mémoire)

5° A → (R1) (une première instruction ramène A dans un registre UC)
(R1) + B → (R1) (l'instruction d'addition agit sur un opérande en registre UC et un autre en mémoire et retourne le résultat dans le même registre UC)
(R1) → C (une troisième instruction envoie le résultat en mémoire)

6° A → (R1) (une première instruction ramène A dans un registre UC)
B → (R2) (une deuxième instruction ramène B dans un autre registre UC)
(R1) + (R2) → (R3) (l'instruction d'addition agit sur des opérandes en registres UC et retourne le résultat dans un registre UC)
(R3) → C (une quatrième instruction envoie le résultat en mémoire)

Notons chaque type d'**instruction d'addition** par (m,n) : au maximum m objets en mémoire, n objets au total. La solution 1° correspond alors à (3,3), 2° et 3° à (2,3), 4° à (1,3), 5° à (1,2) et 6° à (0,3). Chaque UC est conçue pour qu'elle puisse employer certaines de ces possibilités. Voici la classification correspondante de quelques UC existantes :

(0, 2)	IBM RT-PC
(0, 3)	SPARC, MIPS, HP Precision Architecture
(÷1, 2)	PDP 10, Motorola 68000, IBM 360
(÷1, ÷3)	IBM 360 (instructions RS)
(÷2, 2)	PDP 11, NS 32x32, IBM 360 (instructions SS)
toutes	VAX

Avantages et désavantages de ces possibilités :

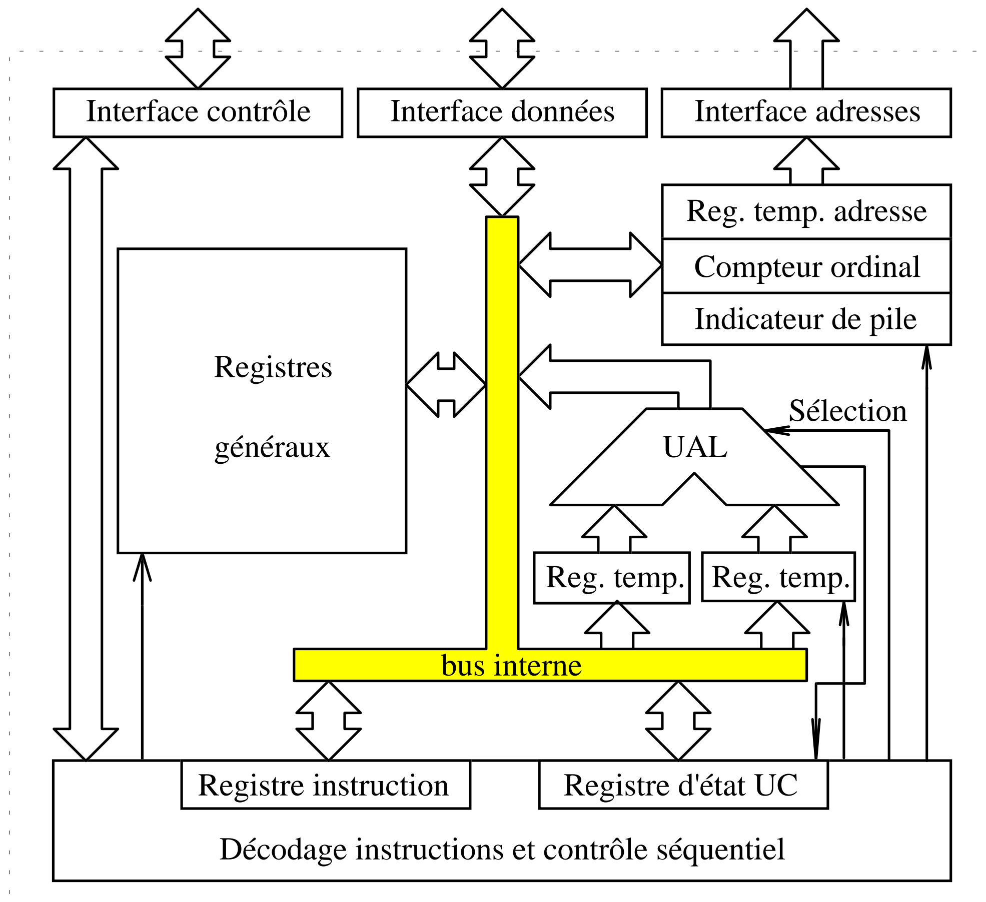
Type	Avantages	Désavantages
(0, 3)	Codes simples et de taille fixe pour les instructions. Décodage facilité et rapide. Durées identiques pour les instructions. Compilation facilitée.	Nombre plus élevé d'instructions dans les programmes. Nombreux registres UC nécessaires pour fonctionnement efficace.
(1, 2)	Codes relativement simples pour les instructions. Programmes plus compacts.	Les opérandes ne sont pas équivalents (l'un est détruit). Durée variable des instructions.
(3, 3)	Programmes très compacts. L'UC peut avoir peu de registres internes.	Variations très importantes des durées d'exécution des instructions. Décodage complexe des instructions. Compilation difficile (trop de choix différents, difficiles à évaluer). La réutilisation des mêmes données ne fait pas baisser le nombre d'accès mémoire.

CISC = *Complex Instruction Set Computer* (correspond à la philosophie (3, 3)).
RISC = *Reduced Instruction Set Computer* (correspond à la philosophie (0, 3)).
Coûts à prendre en compte : coût de développement de l'UC, coût de production des circuits, coût de développement des compilateurs.
Rapport à maximiser : performances/coût (évidemment...).
Quelque arguments en faveur des RISC :

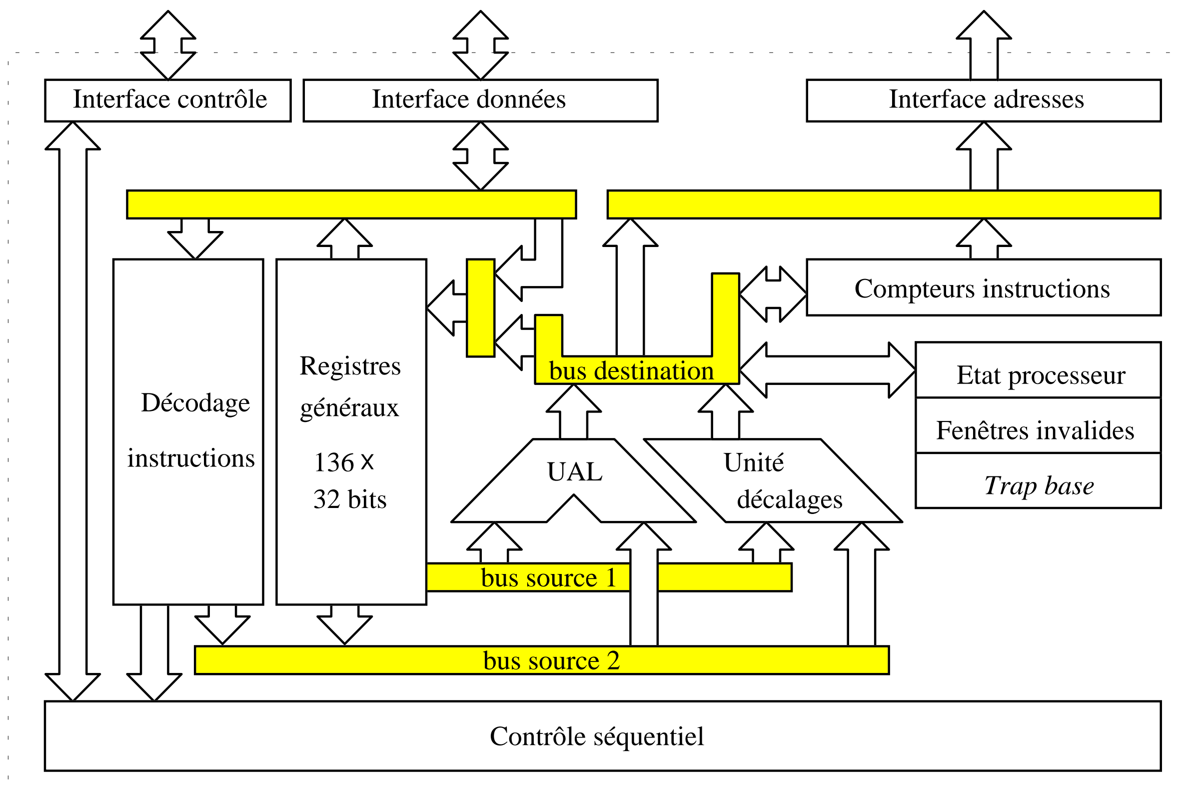
1° Les statistiques de fonctionnement montrent que les hypothèses de proximité sont souvent satisfaites, donc le rapport {(durée calculs internes)/(durée accès mémoire)} reste suffisamment élevé (une valeur basse pénaliserait les RISC par rapport aux CISC).
2° Le coût (place occupée sur circuit, temps de développement) des registres UC est nettement moins important que celui des circuits de contrôle.
3° Simplification significative du contrôle de la pipeline, donc possibilité d'augmenter le nombre d'étages.
4° Décodage simplifié des instructions, partie de contrôle simple ⇒ réduction des délais de propagation, donc réduction possible de la période de l'horloge.
5° Conception de compilateurs-optimiseurs nettement simplifiée (⇒ pour un même coût de développement, le compilateur obtenu est nettement meilleur).

6. Structure et fonctionnement de l'unité centrale

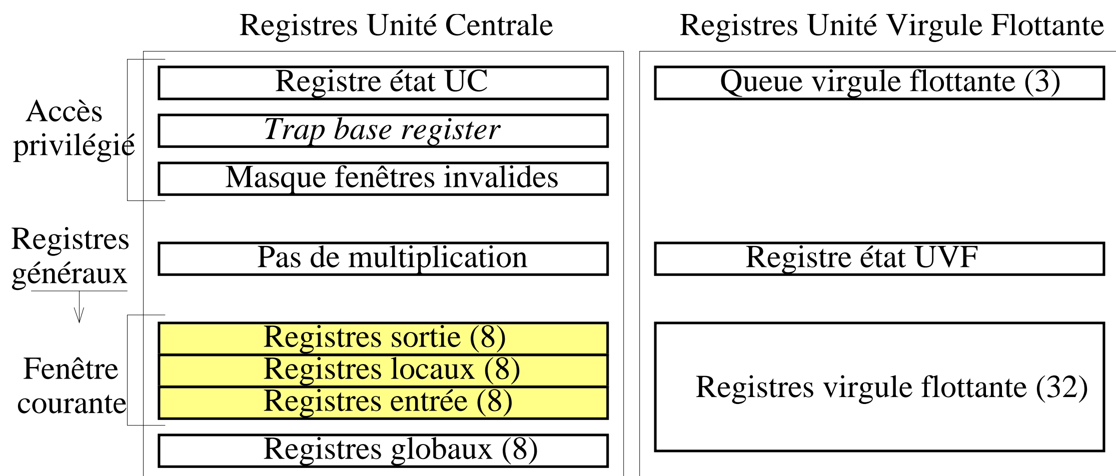
6.1. Structure et fonctionnement d'une unité centrale simple



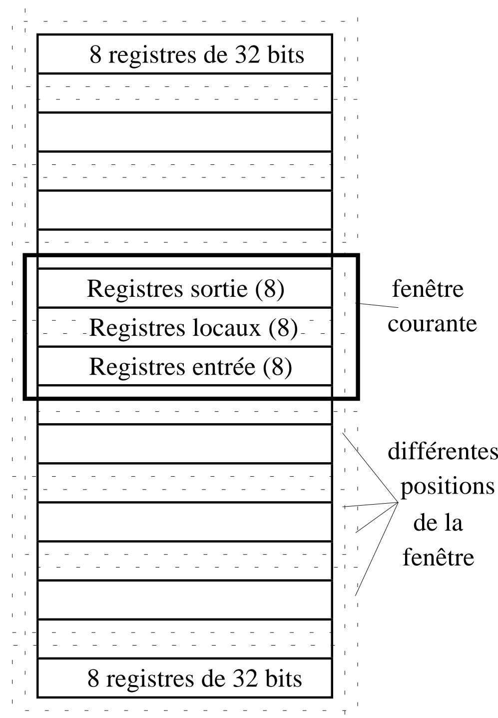
6.2. Structure et fonctionnement de l'unité centrale SPARC



6.2.1. Registres internes



Les registres généraux (registres globaux mis à part) sont organisés en pile :



A chaque instant, seuls les registres globaux et les registres "visibles à travers la fenêtre" sont accessibles. Ce mécanisme permet un changement facile et rapide du contexte (par exemple, pour l'appel de procédures).

6.2.2. Bus, *pipeline*, contrôle

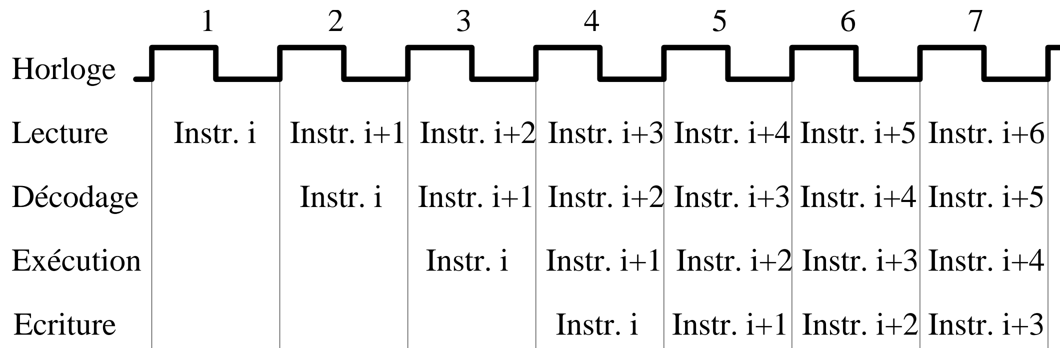
Le nombre de bus internes à l'UC est augmenté, ce qui permet d'effectuer plusieurs transferts (donc opérations élémentaires) en parallèle à l'intérieur de l'UC.
Durant une période du signal d'horloge, sur le bus de données peut se trouver :

1° le code d'une instruction lue (cycle *fetch*) ;
2° une donnée lue ;
3° une donnée à écrire.

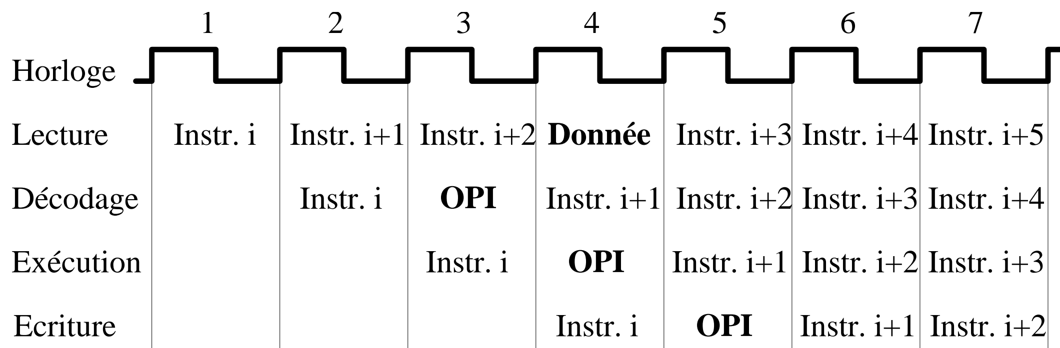
Pour le SPARC, un cycle-instruction élémentaire est décomposé en 4 étapes :

1° Lecture de l'instruction : lecture de la mémoire en utilisant comme adresse le contenu du compteur d'instructions, chargement de l'instruction dans le registre instruction ;
2° Décodage de l'instruction et adressage (à partir des adresses contenues dans le code de l'instruction) des registres internes afin de préparer les opérandes ;
3° Exécution de l'instruction (pour les opérations UAL) ou calcul d'une adresse mémoire (pour les instructions de transfert d'une donnée UC ↔ mémoire) ;
3bis Lecture/écriture mémoire : seulement pour les instructions de transfert d'une donnée entre UC ↔ mémoire ;
4° Ecriture du résultat (de l'opération UAL ou de la lecture de la mémoire) en un registre.

Chaque étape exige (normalement) une seule période d'horloge et nécessite des ressources matérielles différentes. Par conséquence, ces différentes étapes peuvent se dérouler en parallèle pour des instructions successives, comme indiqué dans la figure :



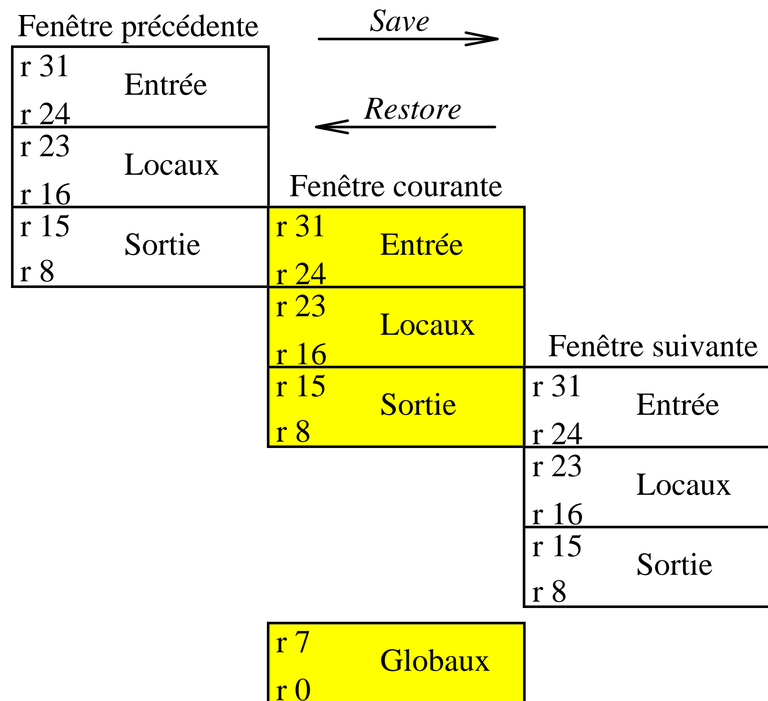
Une instruction demande toujours 4 périodes d'horloge afin d'être exécutée, mais le parallélisme permet de terminer une nouvelle instruction pour chaque période de l'horloge.
Quant une instruction exige une étape 3bis, un code d'opération interne (**OPI**) est engendré à l'intérieur de la pipeline, après le décodage de l'instruction, et l'accès (lecture ou écriture de **donnée**) au bus de données est intercalé dans la séquence normale de lecture d'instructions :



7. Les instructions du SPARC

7.1. Registres et types de données

Déplacement de la fenêtre à l'appel d'une procédure (*Save*) et au retour d'une procédure (*Restore*) :



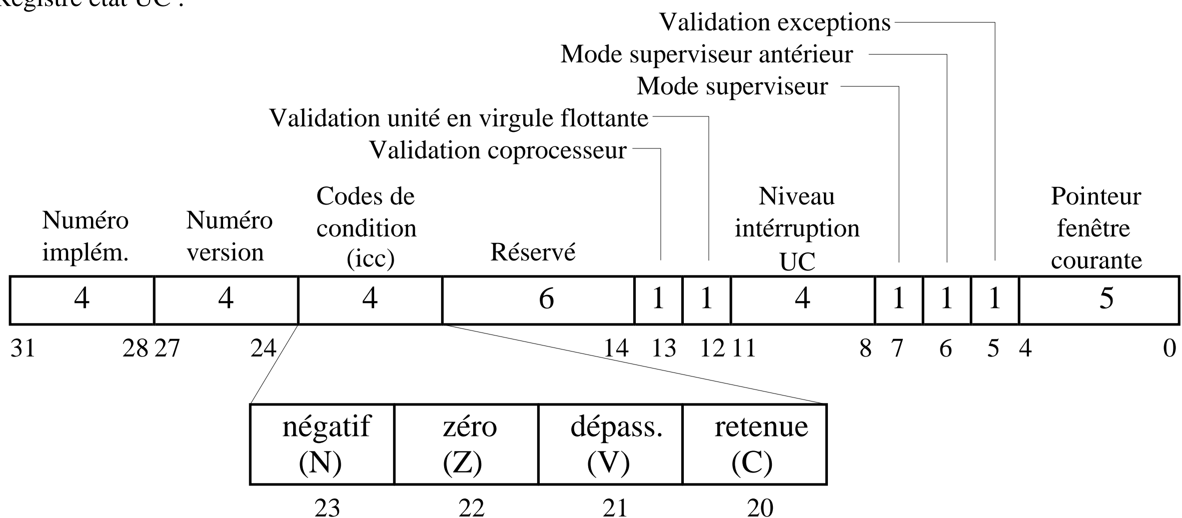
Registres à utilisation particulière :

r0 (registre global) : toute lecture de r0 produit 0
toute écriture dans r0 est sans effet

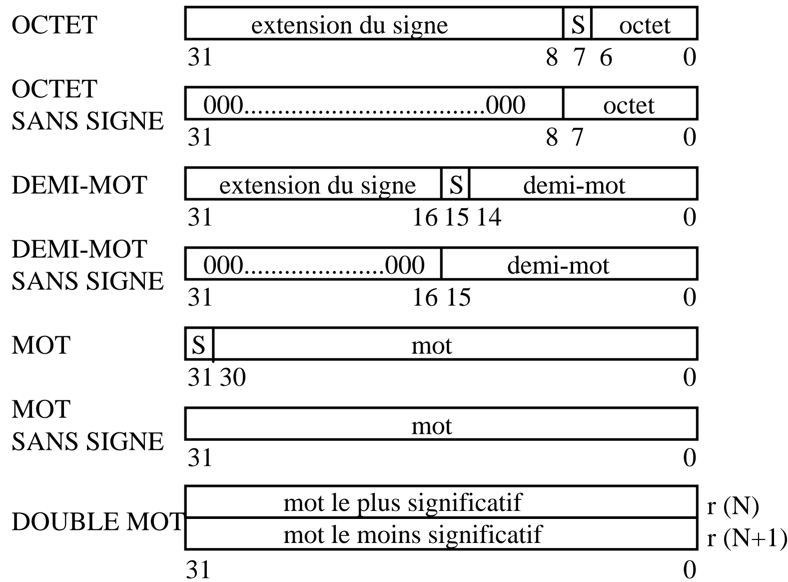
r14 (registre de sortie) : pointeur de pile (*Stack Pointer*), utilisation optionnelle

r31 (registre d'entrée) : adresse de retour (utilisation optionnelle)

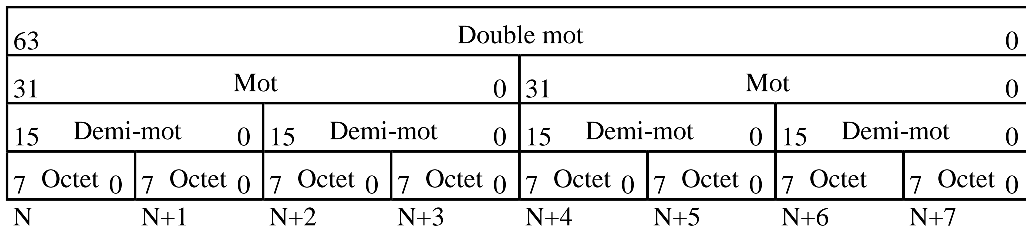
Registre état UC :



Organisation des données dans les registres :



Organisation des données en mémoire :



Directives assembleur pour la définition des données :

(programme assembleur = traduction langage assembleur → langage machine)

`.align n` : indique que ce qui suit débute à une adresse multiple de `n` (`n` = 4 ⇒ adresse de mot de 32 bits)

`.byte, .word` : permettent de définir la valeur de un ou plusieurs octets, mots, etc. qui occupent la mémoire à partir d'une adresse.

Exemple :

```
            .seg        "data"
            .align      4
      a :   .word 2
      b :   .byte 8
            .align      4
      c :   .word 15, 0x4a, 128
            .byte 0x4a, 2
```

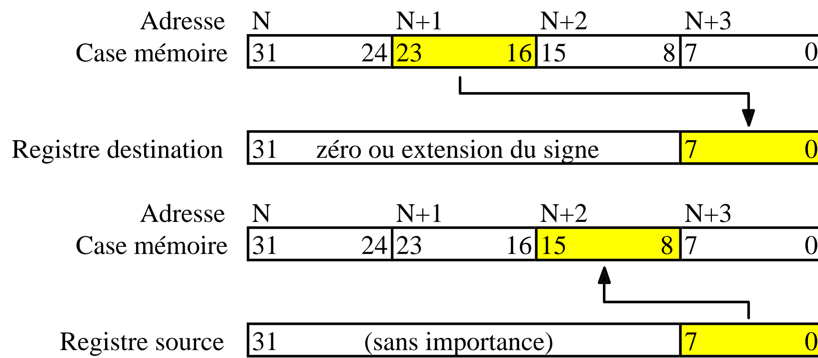
`a, b, c, d` = noms symboliques (étiquettes) donnés à des adresses en mémoire ; sont remplacés par les valeurs numériques correspondantes par le programme assembleur pendant la traduction ;

`0xHH` = notation usuelle pour le nombre `HH` (`H` étant un chiffre hexadécimal).

Effet en mémoire :

Adresse	+0	+1	+2	+3
a	00000000	00000000	00000000	00000010
b	00001000	????????	????????	????????
c	00000000	00000000	00000000	00001111
c+4	00000000	00000000	00000000	01001010
c+8	00000000	00000000	00000000	10000000
d	01001010	00000010	????????	????????

Transfert d'un octet entre la mémoire et un registre (exemples) :

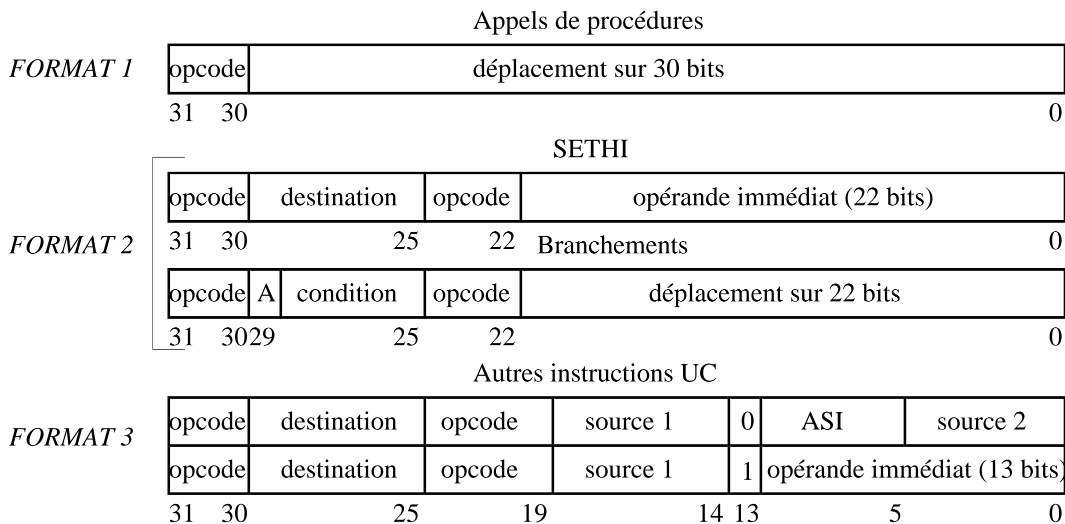


7.2. Types et formats d'instructions. Modes d'adressage

Types d'instructions (UC seulement) :

transferts registres ↔ mémoire ;
opérations arithmétiques et logiques ;
transfert de contrôle (branchements, appels de procédures, exceptions) ;
autres instructions (accès aux registres de contrôle).

Formats d'instructions :



Modes d'adressage :

1° Donnée immédiate : un des opérandes est obtenu à partir de l'opérande immédiat sur 13 bits (instruction format 3) ; employé seulement par des instructions arithmétiques et logiques.

2° Adressage directe d'un registre UC : un opérande (ou le résultat) se trouve (doit être entreposé) dans le registre UC indiqué par un champ source (respectivement. destination) de l'instruction (format 3) ; employé par des instructions arithmétiques et logiques et par les transferts registre UC ↔ mémoire.

3° Adressage indirecte en utilisant deux registres : la donnée se trouve en mémoire, et son adresse est obtenue en additionnant les contenus des registres UC indiqués par les deux champs source de l'instruction ; employé seulement par les transferts registre UC ↔ mémoire.

4° Adressage indirecte en utilisant un registre et une valeur immédiate : la donnée se trouve en mémoire, et son adresse est obtenue en additionnant la donnée immédiate présente dans l'instruction (13 bits) au contenu du registre UC indiqué par le champ source de l'instruction ; employé seulement par les transferts registre UC ↔ mémoire.

5° Adressage relatif au compteur ordinal (*Program Counter relative*) : l'adresse à laquelle le contrôle est transféré est obtenue en additionnant le contenu du compteur ordinal et le déplacement (30 ou 22 bits) présent dans l'instruction ; utilisé seulement par les instructions de transfert de contrôle (branchements, appels de procédures).

7.3. Instructions de transfert registres UC ↔ mémoire

Ces instructions servent à déplacer des octets, des demi-mots (16 bits), des mots (32 bits) et des doubles mots (64 bits) entre la mémoire et un registre général de l'UC. Ce sont les seules instructions qui peuvent accéder aux données stockées en mémoire.
Transferts mémoire → registre(s) (*Load*) :

Nom	Opérande	Syntaxe et effet	
LDSB	octet (signé)	ldsb [s1+s2], d	octet mém. → octet inf. de d, ext. signe
LDUB	octet (sans signe)	ldub [s1+s2], d	octet mém. → octet inf. d, ext. avec 0
LDSH	demi-mot (signé)	ldsh [s1+s2], d	demi-mot mém. → demi-mot inf. de d, ensuite ext. signe
LDUH	demi-mot (sans s.)	lduh [s1+s2], d	demi-mot mém. → demi-mot inf. de d, ensuite ext. avec 0
LD	mot (32 bits)	ld [s1+s2], d	mot mémoire → d
LDD	double mot (64 bits)	ldd [s1+s2], d	double mot mém. → registres d et d+1

Transferts registre(s) → mémoire (*Store*) :

Nom	Taille opérande	Syntaxe et effet	
STB	octet	stb d, [s1+s2]	octet inf. de d → mémoire
STH	demi-mot (16 bits)	sth d, [s1+s2]	demi-mot inf. de d → mémoire
ST	mot (32 bits)	st d, [s1+s2]	mot du registre d → mémoire
STD	double mot (64 bits)	std d, [s1+s2]	double mot de d et d+1 → mémoire

s1, **d** = numéros de registres généraux de l'UC (notation : %r, avec $0 \leq r \leq 31$) ; **s1** correspond au champ source 1 du code de l'instruction, **d** au champ destination ;
s2 = numéro de registre (correspond au champ source 2 du code de l'instruction) **ou** valeur immédiate présente dans le code de l'instruction (13 bits, avec signe).
Exemples :

Hypothèses :
1° le registre UC l0 contient la valeur 1024 (400 hexa) et l1 la valeur 8
2° les données suivantes se trouvent en mémoire aux adresses indiquées :

adresse :	donnée (en hexadécimal) :
1024	12345678
1028	abcdef00
…	…
1032	0000aaaa

Considérons que la suite d'instructions suivante est exécutée :

```
ldub        [%l0+3], %l2
ldsb        [%l0+4], %l3
ldsh        [%l0+0], %l4
ld          [%l0+%l1], %l5
ldd         [%l0+0], %l6
stb         %l3, [%l0+1]
sth         %l4, [%l0+2]
st          %l7, [%l0+%l1]
```

Registres internes concernés, après exécution :

registre :	donnée (en hexadécimal) :
l0	00000400
l1	0000000a
l2	00000078
l3	ffffffab
l4	00001234
l5	0000aaaa
l6	12345678
l7	abcdef00

Nouvelle configuration de la mémoire (après exécution) :

adresse :	donnée (en hexadécimal) :
1024	12ab1234
1028	abcdef00
…	…
1032	abcdef00

Questions : Comment on charge une valeur numérique spécifique, représentée sur 32 bits, dans un registre UC ? Comment on copie un registre UC dans un autre ? (voir plus loin …)

7.4. Instructions arithmétiques, logiques et de translation (*shift*)

Chaque instruction a deux formes : la forme présentée dans le tableau, qui n'affecte pas les indicateurs (signe, retenue, zéro, dépassement) du registre d'état UC, et une forme qui modifie ces indicateurs en fonction du résultat de l'opération. Pour spécifier cette deuxième forme, il faut ajouter "cc" à la fin du nom (par exemple add → addcc).

Nom	Syntaxe et effet		
ADD	add s1, s2, d	s1+s2→d	(addition simple)
ADDX	addx s1, s2, d	s1+s2+C→d	(addition avec retenue)
SUB	sub s1, s2, d	s1−s2→d	(soustraction simple)
SUBX	subx s1, s2, d	s1−s2−C→d	(soustraction avec retenue)
MULScc	mulscc s1, s2, d	un pas de multiplication, utilise aussi le reg. "Pas de mult."	
AND	and s1, s2, d	s1 ET s2→d	(ET logique bit par bit)
ANDN	andn s1, s2, d	s1 ET $\overline{s2}$→d	
OR	or s1, s2, d	s1 OU s2→d	(OU logique bit par bit)
ORN	orn s1, s2, d	s1 OU $\overline{s2}$→d	
XOR	xor s1, s2, d	s1 XOR s2→d	(OU EXCLUSIF bit par bit)
XNOR	xnor s1, s2, d	s1 XOR $\overline{s2}$→d	
SLL	sll s1, s2, d	d←s1 déplacé à gauche par *shcnt*	
SRL	srl s1, s2, d	d←s1 déplacé à droite par *shcnt*, rempliss. à gauche avec 0	
SRA	sra s1, s2, d	d←s1 déplacé à droite par *shcnt*, extension signe à gauche	
SETHI	sethi const22, d	d (22 bits à partir du MSB) ←const22	

s1, **d** = numéros de registres généraux de l'UC (notation : %r, avec $0 \le r \le 31$) ; **s1** correspond au champ source 1 du code de l'instruction, **d** au champ destination ;
s2 = numéro de registre (correspond au champ source 2 du code de l'instruction) **ou** valeur immédiate (13 bits, avec signe) présente dans le code de l'instruction ;
shcnt = valeur contenue dans les 5 derniers bits (5 bits à partir du LSB) du registre indiqué par le champ source 2 du code de l'instruction **ou** de la valeur immédiate (13 bits) présente dans le code de l'instruction ;
const22 = valeur immédiate (22 bits) présente dans le code de l'instruction.
Exemples d'utilisation :
1° Transfert entre registres UC (pas d'instruction dédiée) :

```
        or    %g0, %l2, %o1
```

effet : contenu registre g0 OU (bit par bit) contenu registre l2 → registre o1 ; comme toute lecture de g0 donne 00...........00 (32 bits), le résultat est en fait la copie du contenu du registre l2 dans le registre o1.

2° Chargement d'une valeur numérique spécifique sur 32 bits dans un registre UC :

valeur à charger : 46A3B2F1 hexadécimal, notée 0x46a3b2f1 (convention usuelle)

séquence d'instructions (exemple) :

```
sethi %hi 0x46a3b2f1, %g3
or        %g3, %lo 0x46a3b2f1, %g3
```

effet : le registre g3 est chargé avec la valeur 46A3B2F1 ;

explication : `%hi` et `%lo` sont des opérateurs interprétés par l'assembleur (logiciel qui traduit le programme en langage assembleur en programme machine) ; `%hi` indique que les 22 bits les plus significatifs de la constante numérique qui suit sont utilisés ; `%lo` indique que les 10 bits les moins significatifs de la constante numérique qui suit sont utilisés ; `sethi` permet de charger les 22 bits les plus significatifs du registre g3 (et de mettre à 0 les 10 bits les moins significatifs), `or` permet charger les 10 bits les moins significatifs de g3 (toute lecture de g0 donne 00...........00 (32 bits)).

3° Etude de la séquence d'instructions suivante :

```
sethi %hi (nbr), %l0
ldsb      [%l0+%lo(nbr)], %l2
ld        [%l0+%lo(nbr)], %l5
ldd       [%l0+%lo(nbr)], %l6
or        %l0, %lo(nbr), %l0
stb       %l5, [%l0+7]
sth       %l2, [%l0+4]
st        %l7, [%l0+%g0]
```

avec :

```
      .seg      "data"
nbr:  .word 0xabcd0000, 0x12345678
```

Après exécution :

```
nbr:
      .word 0x12345678, 0xffab5600
```

7.5. Instructions de transfert de contrôle

Nom	Syntaxe et effet
SAVE	save s1, s2, d déplacement de fenêtre après l'appel d'une procédure aussi : s1+s2 (ancienne fenêtre) →d (nouvelle fenêtre)
RESTORE	restore s1, s2, d déplacement de fenêtre avant le retour d'une procédure aussi : s1+s2 (ancienne fenêtre) →d (nouvelle fenêtre)
Bicc	b(icc){,a} const22 branchement à CO + const22×4 si condition icc vraie
CALL	call const30 appel de procédure à CO + const30×4 (voir plus bas)
JMPL	jmpl s1+s2, d branchement inconditionnel avec liaison (voir plus bas)
RETT	rett s1, s2 retour d'une exception
Ticc	t(icc) s1, s2 exception si condition icc vraie

s1, **d** = numéros de registres généraux de l'UC (notation : %r, avec $0 \leq r \leq 31$) ; **s1** correspond au champ source 1 du code de l'instruction, **d** au champ destination ;

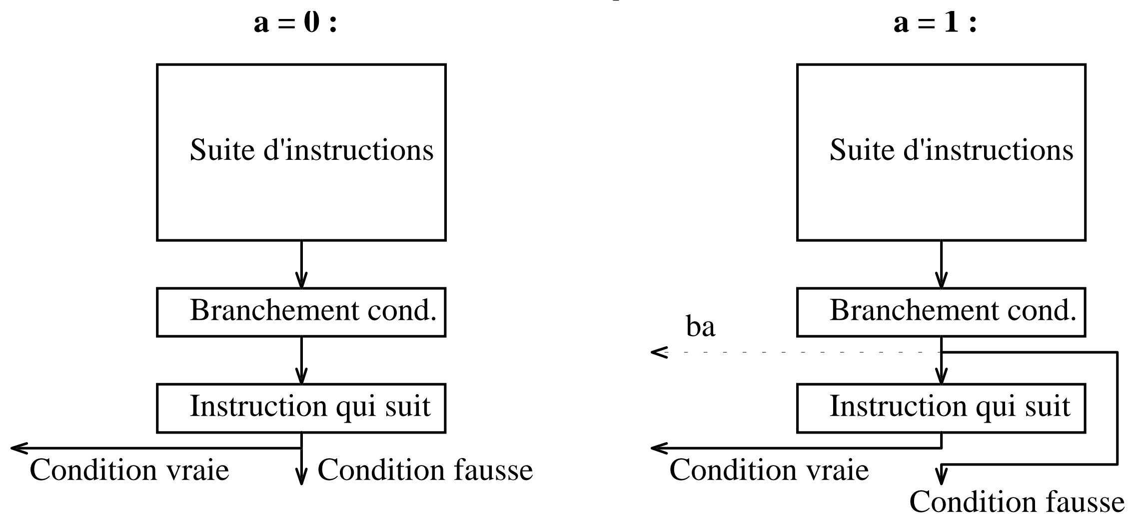
s2 = numéro de registre (correspond au champ source 2 du code de l'instruction) **ou** valeur immédiate (13 bits, avec signe) présente dans le code de l'instruction ;

{,a} : si présent, le bit annulation est mis à 1 (voir plus bas) ;

icc = conditions pour les branchements (Bicc) et les exceptions (*traps*, Ticc) :

Valeur **icc**	Signification	Valeur **icc**	Signification
a	toujours (**a***lways*)	gu	supérieur à, sans signe
n	jamais (**n***ever*)	leu	inférieur ou égal, sans signe
ne (nz)	Z = 0 (**n***ot* **e***qual*)	cc (geu)	pas de retenue (C=0, **c***arry* **c***lear*)
e (z)	Z = 1 (**e***qual*)	cs (lu)	retenue (C=1, **c***arry* **s***et*)
l	inférieur à (**l***ess*)	pos	positif (N=0, **p***ositive*)
le	inférieur ou égal (**l***ess or* **e***qual*)	neg	négatif (N=1, **n***egative*)
g	supérieur à (**g***reater*)	vc	pas de dépassement (V=0)
ge	sup. ou égal (**g***reater or* **e***qual*)	vs	dépassement (V=1, *o***V***erflow*)

Dans l'UC SPARC nous trouvons à chaque instant plusieurs instructions successives, en différentes étapes d'exécution. Un transfert du contrôle (suite à un branchement conditionnel, un appel ou un retour de procédure) nécessiterait ainsi l'élimination d'une instruction du *pipeline* (pour un branchement conditionnel, la condition de branchement est évaluée et l'adresse à laquelle le contrôle doit être transféré est calculée pendant l'étape 2 — décodage de l'instruction) et donc introduirait un retard d'au moins une période d'horloge. Le SPARC peut éviter cela en permettant l'exécution de l'instruction qui suit l'instruction de transfert de contrôle, avant d'effectuer ce transfert de contrôle. Bien sûr, le programmeur (ou le compilateur) doit écrire après l'instruction de branchement une instruction dont l'exécution avant le transfert du contrôle soit conforme à l'algorithme à implémenter. Si une telle instruction n'est pas trouvée, une instruction inopérante doit être écrite à la place (instruction inopérante recommandée par le constructeur : `sethi %hi 0, %g0`).
Influence du bit d'annulation sur l'exécution de l'instruction qui suit un branchement conditionnel :



Le branchement inconditionnel (instruction `ba`, *branch always*) est un cas particulier : quand le bit d'annulation est à 1, ce branchement s'exécute comme un branchement inconditionnel standard (transfert du contrôle sans exécution d'une instruction intermédiaire).

Appel d'une procédure :

L'appel d'une procédure est composé d'une instruction (optionnelle) de déplacement de la fenêtre de registres (SAVE) et d'une instruction de transfert du contrôle (CALL).
Exécution de SAVE :

1° le contenu du registre **s1** est additionné au contenu de **s2** (registre ou donnée immédiate sur 13 bits) ;
2° la fenêtre de registres est déplacée (le contenu du champ "Pointeur fenêtre courante" du registre état UC est **réduit** de 1 — modulo 8) ;
3° le résultat de l'étape 1° est écrit dans le registre **d** de la **nouvelle** fenêtre (ou registre global, accessible quelque soit la fenêtre) ; le rôle des opérations 1° et 3° est d'inscrire une nouvelle valeur de pointeur de pile dans le registre **d** de la nouvelle fenêtre, à partir de la valeur du pointeur de pile du programme appelant (registre **s1**) et d'un déplacement (**s2**) ; si aucun pointeur de pile n'est employé, cette opération peut être rendue ineffective en utilisant par exemple :

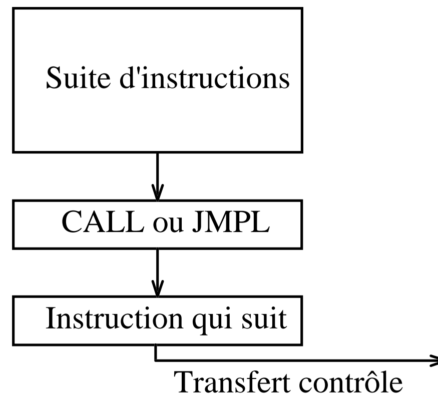
```
save        %g0, 0, %g0
```

Exécution de l'appel :

1° étape de l'exécution de CALL : le contenu du Compteur ordinal (compteur d'instructions) courant est écrit dans le registre o7 de la fenêtre courante afin de pouvoir récupérer l'adresse de retour dans le programme appelant ;

2° étape de l'exécution de CALL : le déplacement immédiat (sur 30 bits) présent dans le code de l'instruction est multiplié par 4 (afin d'obtenir un alignement sur une adresse de mot) et la valeur obtenue est additionnée au contenu du Compteur ordinal ;

3° l'instruction écrite après le CALL est exécutée (il n'y a pas de bit d'annulation) ; il peut s'agir d'une instruction SAVE ;



4° la première instruction du programme appelant est exécutée.

Retour d'une procédure :

Le retour d'une procédure est composé d'une instruction (optionnelle) de déplacement de la fenêtre de registres (RESTORE) et d'une instruction de transfert du contrôle (JMPL).

Exécution de RESTORE:

1° le contenu du registre **s1** est additionné au contenu de **s2** (registre ou donnée immédiate sur 13 bits) ;

2° la fenêtre de registres est déplacée (le contenu du champ "Pointeur fenêtre courante" du registre état UC est **augmenté** de 1 — modulo 8) ;

3° le résultat de l'étape 1° est écrit dans le registre **d** de la **nouvelle** fenêtre (ou registre global, accessible quelque soit la fenêtre) ; le rôle des opérations 1° et 3° est de refaire, dans le registre **d** de la nouvelle fenêtre, la valeur du pointeur de pile du programme appelant (opération inverse à celle de SAVE) ; si aucun pointeur de pile n'est employé, cette opération peut être rendue ineffective en utilisant par exemple :

```
restore          %g0, 0, %g0
```

Exécution du retour :

1° étape de l'exécution de JMPL : le contenu du Compteur ordinal (compteur d'instructions) courant est écrit dans le registre **d**, global ou de la fenêtre courante (cette étape est inutile pour le retour d'une procédure, mais JMPL peut avoir aussi d'autres utilisations) ; pour un retour, **d** peut être %g0 (écriture inopérante) ;

2° étape de l'exécution de JMPL : la somme entre le contenu du registre **s1** et le contenu de **s2** (registre ou donnée immédiate) remplace le contenu du Compteur ordinal ; pour que cela corresponde à un retour de procédure, **s1** doit être %i7 et **s2** doit être la valeur immédiate **8** ;

3° l'instruction écrite après le JMPL est exécutée (il n'y a pas de bit d'annulation) ; il peut s'agir d'une instruction RESTORE ;

4° le retour est effectif, l'instruction suivante du programme appelant est exécutée.

Séquences de retour correctes :

1°
```
jmpl      %i7 + 8, %g0
restore   …
```

(l'apparition dans un programme de la **pseudo-instruction** `ret` est remplacée par le programme de traduction langage assembleur → langage machine avec cette forme de l'instruction JMPL) ;

2°
```
restore   …
jmpl      %o7 + 8, %g0
```

```
          sethi %hi 0, %g0   ← instruction inopérante
```

L'instruction JMPL peut aussi remplacer un CALL :

```
          jmpl        s1+s2, %o7
```

Exemples d'utilisation :

1° Placer dans le registre l0 le maximum (valeur avec signe) des contenus de l1 et l2 :

```
          …
          subcc %l1, %l2, %g0
          bl          et1
          or          %g0, %l1, %l0
          ba, a et2
    et1:  or          %g0, %l2, %l0
    et2:  …
```

(observation : l'instruction `or %g0, %l1, %l0` est toujours exécutée).

2° Multiplication de nombres entiers (facteurs sur 32 bits, produit sur 64 bits) positifs :

```
          .seg        "data"
          .align      4
    fpr:  .word 0x10000000, 0x10000000
          .seg        "prog"
          sethi %hi (fpr), %l0
          or          %l0, %lo (fpr), %l0
          ldd         [%l0+0], %l1
          wr          %l1, %g0, %y
          or          %g0, %g0, %l1
          or          %g0, 32, %l3
    mlt:  subcc %l3, 1, %l3
          bnz         mlt
          mulscc      %l1, %l2, %l1
          rd          %y, %l2
          std         [%l0+0], %l1
```

3° Chercher la première occurrence de l'octet inférieur de l1 dans une suite de 10 octets :

```
          .seg        "data"
    chr:  .byte 0x4d, 0x41, 0x39, 0x46, 0x4e
          .byte 0x42, 0x43, 0x4a, 0x49, 0x4c
          .seg        "prog"
          sethi %hi (chr), %l0
          or          %l0, %lo (chr), %l0
          or          %g0, 0, %l3
    cmp:  ldub        [%l0+%l3], %l2
          subcc %l1, %l2, %g0
          bz          nxt
          subcc %l3, 9, %g0
          bnz         cmp
          add         %l3, 1, %l3
    nxt:  …
```

(si à la fin le registre l3 contient la valeur 10, l'octet n'a pas été trouvé dans la suite).

4° Traduire en langage assembleur SPARC l'algorithme suivant :

```
    Programme MAX
    variables : chr[0…9] : tableau d'entiers ;
                i, mx :          entiers ;
    début
          i:=0 ;
          mx:=-256 ;
```

```
        tant_que i ≤ 9 faire
              si chr[i] > mx alors
                    mx:=chr[i] ;
              fin si
              i:=i+1 ;
        fin tant_que
  fin MAX
```

sachant que le tableau chr est donné par :

```
        .seg        "data"
  chr:  .byte 0x4d, 0x41, 0x39, 0x46, 0x4e
        .byte 0x42, 0x43, 0x4a, 0x49, 0x4c
```

Programme (**mx** gardée dans le registre l2 et **i** dans le registre l1) :

i) traduction directe, en écrivant des instructions inopérantes après les branchements :

```
        .seg        "prog"
        sethi %hi (chr), %l0
        or          %l0, %lo (chr), %l0
        or          %g0, 0, %l1
        sub         %g0, 256, %l2
  lp:   ldsb        [%l0+%l1], %l3
        subcc %l2, %l3, %g0
        bge         lp1
        sethi %hi 0, %g0   ← instruction inopérante
        or          %l3, %g0, %l2
  lp1:  add         %l1, 1, %l1
        subcc %l1, 10, %g0
        bnz         lp
        sethi %hi 0, %g0   ← instruction inopérante
        .end        "prog"
```

(le résultat **mx** reste dans le registre l2).

ii) optimisation, avec le bit d'annulation à 0 et respectivement à 1 pour les 2 branchements :

```
        .seg        "prog"
        sethi %hi (chr), %l0
        or          %l0, %lo (chr), %l0
        or          %g0, 0, %l1
        sub         %g0, 256, %l2
        ldsb        [%l0+%l1], %l3
  lp:   subcc %l2, %l3, %g0
        bge         lp1
        add         %l1, 1, %l1
        or          %l3, %g0, %l2
  lp1:  subcc %l1, 10, %g0
        bnz, a      lp
        ldsb        [%l0+%l1], %l3
        .end        "prog"
```

(le résultat **mx** reste dans le registre l2).

5° Calcul récursif de la somme S des N premiers entiers positifs (on considère $2^{15} \geq N \geq 0$) :

L'algorithme (procédure récursive) :

```
  Procédure SOMREC(n)
  paramètres :      n :   entier ;
  variables : s :   entier ;
  début
        si n ≤ 1 alors
              s:=1 ;
        sinon
              s:=n + SOMREC(n-1) ;
        fin si
        retourner s ;
```

```
        fin SOMREC
```

Le programme en langage assembleur SPARC :

```
                .seg          "proc"
        somrec:
                save          %g0, 0, %g0
                subcc %i0, 1, %g0
                bleu,a        rt
                or            %g0, 1, %i0
                call          somrec
                sub           %i0, 1, %o0
                add           %i0, %o0, %i0
        rt:     ret                         ← équivalente à  jmpl %i7 + 8, %g0
                restore       %g0, 0, %g0
```

un appel de la procédure à partir d'un programme principal aura la forme suivante :

```
                …
                <préparation de la valeur de N dans le registre o0>
                call          somrec
                sethi %hi 0, %g0   ← instruction inopérante
                <récupération de la valeur de S du registre o0>
                …
```

7.7. Autres instructions

Parmi les autres instructions de l'UC SPARC nous pouvons mentionner :

Nom	Syntaxe et effet	
RDY	rd %y, d	registre pas de multiplication (%y) → registre d
RDPSR*	rd %psr, d	registre état UC (%psr) → registre d
RDWIM*	rd %wim, d	registre fenêtres invalides (%wim) → registre d
RDTBR*	rd %tbr, d	*Trap Base Register* (%tbr) → registre d
WRY	wr s1, s2, %y	s1 XOR s2 → registre pas de multiplication (%y)
WRPSR*	wr s1, s2, %psr	s1 XOR s2 → registre état UC (%psr)
WRWIM*	wr s1, s2, %wim	s1 XOR s2 → registre fenêtres invalides (%wim)
WRTBR*	wr s1, s2, %tbr	s1 XOR s2 → *Trap Base Register* (%tbr)
LDSTUB¨	ldstub [s1+s2], d	octet mém.→d, ensuite 1111.1111→octet mém.
SWAP¨	swap [s1+s2], d	case mémoire ↔ registre d

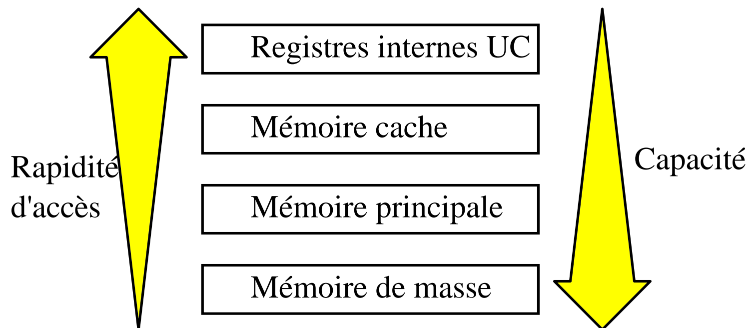
Les instructions marquées avec * ne peuvent être exécutées que si l'UC est en mode superviseur (bit S du registre d'état UC est 1). Les instructions marquées avec ¨ sont indivisibles et ne peuvent pas être interrompues ; ces instructions sont employées dans des systèmes multiprocesseur.

s1, **d** = numéros de registres généraux de l'UC (notation : %r, avec $0 \leq r \leq 31$) ; **s1** correspond au champ source 1 du code de l'instruction, **d** au champ destination ;

s2 = numéro de registre (correspond au champ source 2 du code de l'instruction) **ou** valeur immédiate (13 bits, avec signe) présente dans le code de l'instruction.

8. Organisation et gestion de la mémoire

8.1. Hiérarchie de mémoires



Hypothèses de localité :

1° *Localité temporelle* : un objet déjà référencé le sera à nouveau bientôt.

2° *Localité spatiale* : les objets proches d'un objet référencé seront référencés bientôt.

Ces hypothèses de localité sont confirmées par des statistiques de fonctionnement ; cela assure l'utilisation efficace de chaque niveau intermédiaire de la hiérarchie comme un tampon de taille limitée entre l'unité centrale et le niveau suivant, plus lent, de la hiérarchie.

Unité minimale d'information qui peut être soit présente soit absente (dans la mémoire cache ou dans la mémoire principale) = **bloc**. Toute adresse a donc deux composantes :

numéro du bloc	position dans le bloc

Questions à prendre en compte (pour la mémoire cache et pour la mémoire principale) :

1° Où peut-on placer un bloc ?

2° Comment retrouve-t-on un bloc ?

3° Quel bloc doit être remplacé en cas d'absence du bloc référencé (*miss*) ?

4° Comment fonctionne l'écriture ?

Caractéristiques* :

Composante	Registres UC	Cache	Mémoire principale	Disques
Taille typique	< 1 Koctet	< 512 Koctets	< 512 Moctets	> 1 Goctet
Temps d'accès	10 ns	20 ns	100 ns	20 000 000 ns
Débit typique	800 Moctets/s	200 Moctets/s	133 Moctets/s	4 Moctets/s
Gérée par	Compilateurs	*Hardware*	Syst. exploitation	Syst. exploit.

*année de référence 1990

8.2.1. La mémoire cache et sa gestion

La mémoire cache permet d'adapter la vitesse de la mémoire principale à la vitesse de l'UC.

Les questions et les réponses :

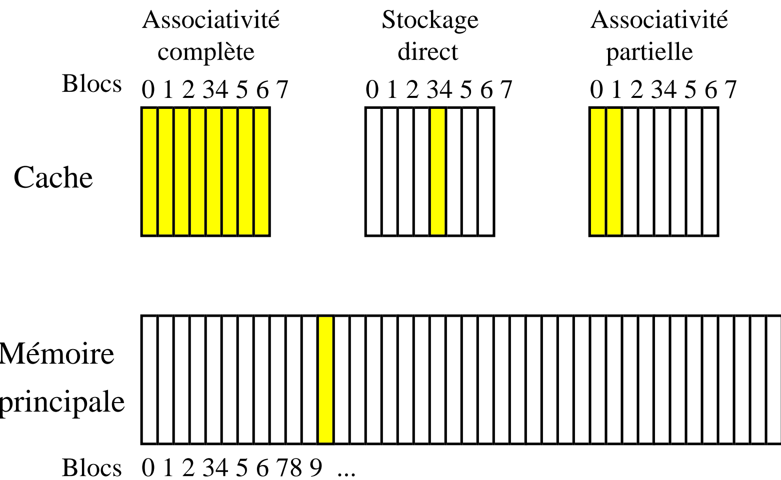
Q1 : Où peut-on placer un bloc ?

R : Possibilités :

1° stockage complètement associatif : un bloc quelconque de la mémoire principale peut se retrouver dans n'importe quel bloc de la mémoire cache ;

2° stockage direct : le bloc n de la mémoire principale peut se retrouver seulement dans le bloc $m = (n \text{ modulo } sb)$ de la mémoire cache, sb étant la taille en nombre de blocs de la mémoire cache ;

3° associativité partielle (*set associative*) : séparation de la mémoire cache en groupes de blocs et associativité complète dans un groupe, c.à.d. le bloc n de la mémoire principale peut se retrouver dans n'importe quel bloc du groupe $g = (n \text{ modulo } sg)$ de la mémoire cache, sg étant le nombre total de groupes de blocs dans la mémoire cache.



Q2 : Comment retrouve-t-on un bloc ?
R : Chaque bloc de la cache a un descripteur associé, qui contient son numéro de bloc en mémoire principale.

1° Si le stockage est direct, l'adresse d'un bloc dans la cache est obtenue par la troncation de son numéro de bloc en mémoire principale. Le descripteur associé est ensuite comparé avec le numéro du bloc en mémoire principale ; si le résultat est négatif, le bloc recherché par l'UC est ramené de la mémoire principale (l'UC est mise en attente) ; si le résultat est positif, le transfert UC ↔ cache a lieu normalement.

2° Si le stockage est complètement associatif, la composante "numéro du bloc en mémoire principale" de l'adresse sortie par l'UC est comparée simultanément avec les descripteurs de tous les blocs de la cache ; si aucune coïncidence n'est obtenue, le bloc recherché est absent de la cache et doit donc être ramené de la mémoire principale.

3° Si le stockage est partiellement associatif, la combinaison correspondante des deux méthodes antérieures est employée.

Q3 : Quel bloc doit être remplacé dans le cache en cas d'absence du bloc référencé (*miss*) ?
R : Si le stockage est direct, il n'y a pas d'alternatives. Si un degré d'associativité est présent, deux stratégies de base (résultats des statistiques meilleurs pour la deuxième stratégie, mais sensiblement proches …) :

1° Choix aléatoire du bloc à remplacer.

2° Remplacement du bloc le moins récemment utilisé (principes de localité).

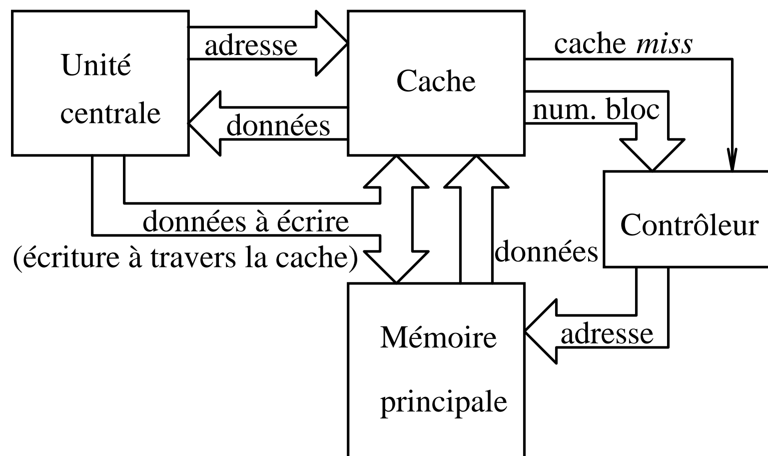
Q4 : Comment fonctionne l'écriture ?
R : Premier constat (statistique) : pour les UC de type RISC, les écritures en mémoire représentent environ 10% des accès mémoire …
Possibilités :

1° Ecriture à travers la cache : toute écriture est effectuée simultanément en mémoire cache et en mémoire principale ; le remplacement d'un bloc dans la cache ne produit pas d'écriture en mémoire principale.

2° Ecriture au remplacement du bloc : l'UC n'écrit que dans la cache, mais tout bloc se trouvant dans la cache et ayant été modifié par l'UC est réécrit en mémoire principale au moment où il doit être remplacé dans la cache par un autre bloc.

Problème principal : les dispositifs d'entrée/sortie effectuent des transferts directement avec la mémoire principale, donc des incohérences peuvent apparaître entre les informations gardées en mémoire cache et en mémoire principale. Les solutions combinent en général des composantes matérielles et logicielles.

Introduction de la mémoire cache dans un système :



Comment assurer une débit élevé pour les transferts cache ↔ mémoire centrale (condition nécessaire pour l'efficacité de la cache) :

1° Utilisation d'un bus de données de taille supérieure entre la cache et la mémoire principale (64 bits, 128 bits).

2° Séparation de la mémoire principale en **bancs** et multiplexage sur le bus de données des informations destinées aux bancs différents.

3° Utilisation de modes spécifiques d'accès pour les circuits de RAM dynamiques qui composent la mémoire principale (par ex., accès page : l'adresse de ligne est présentée une seule fois et ensuite plusieurs colonnes différentes sont lues successivement).

8.2.2. Mémoire principale et mémoire virtuelle

La mémoire principale permet d'adapter la vitesse de la mémoire de masse (disques magnétiques) à la vitesse de l'UC. Les blocs sont dans ce cas appelés couramment "pages".
L'adresse mémoire produite par l'unité centrale est appelée adresse virtuelle, et l'espace d'adressage correspondant "mémoire virtuelle". Une adresse reçue par la mémoire principale est appelée adresse physique, et l'espace d'adressage correspondant "espace physique". La mémoire principale (physique) a en général une taille nettement inférieure à la mémoire virtuelle. L'association (partie de) mémoire virtuelle → espace physique est assurée par des mécanismes de gestion de la mémoire (appelés "gestion de la mémoire virtuelle"). La gestion de la mémoire virtuelle n'est pas assurée principalement par le *hardware*, comme pour la mémoire cache, mais principalement par des composantes du système d'exploitation. Principe : l'espace virtuel et l'espace physique sont composés de "pages" (de même taille), à chaque instant un certain nombre de pages virtuelles — correspondant au nombre de pages dans l'espace physique — se trouvent en mémoire principale, les autres pages virtuelles étant seulement dans la mémoire de masse ; quand une nouvelle page virtuelle est nécessaire, elle est chargée de la mémoire de masse en mémoire principale ou elle remplace une autre page virtuelle.
Des mécanismes de protection entre programmes (et entre programmes et zones de données) sont rajoutés à la gestion de la mémoire virtuelle : chaque entrée dans la table de correspondances contient aussi une description des accès permis à cette page.
Les questions et les réponses :

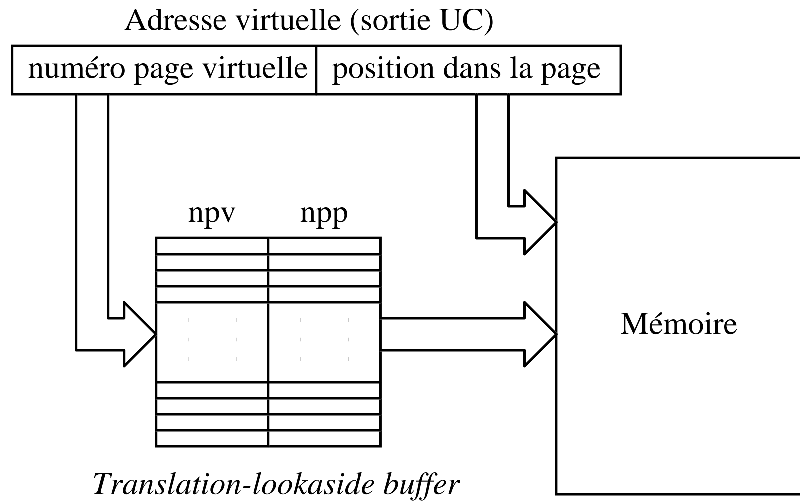
Q1 : Où peut-on placer une page virtuelle en mémoire principale ?

R : Une page virtuelle peut être placée dans n'importe quelle page physique (stockage complètement associatif). Ce choix assure le nombre le plus bas de transferts de pages virtuelles entre la mémoire principale et la mémoire de masse (ces transferts étant extrêmement coûteux …).

Q2 : Comment retrouve-t-on une page virtuelle ?

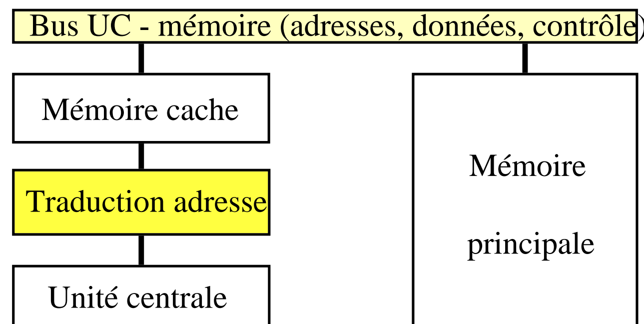
R : Une table de correspondances est employée : le numéro de la page virtuelle (une partie de l'adresse produite par l'UC) est l'adresse dans la table, et à cette adresse on trouve le numéro de la page physique correspondante (ou l'indication qu'elle est absente de l'espace physique), ainsi que des information concernant la protection de la page. Si aucune page physique ne correspond à a la page virtuelle recherchée, cette page est chargée de la mémoire de masse. La table de correspondances a souvent une taille importante et seulement une partie (en général la plus récemment utilisée) se trouve, à un instant

donné, dans un circuit de traduction rapide spécifique (*translation-lookaside buffer, TLB*), le reste étant gardé en mémoire.



Q3 : Quelle page doit être remplacée en mémoire en cas d'absence de la page référencée (*page fault*) ?
R : En général c'est la page la moins récemment référencée (consistant avec les principes de localité), mais d'autres choix sont possibles et la stratégie choisie par le système d'exploitation peut dépendre du contexte.
Q4 : Comment fonctionne l'écriture ?
R : En raison du coût excessif d'une écriture en mémoire de masse, tous les systèmes existants utilisent l'écriture au remplacement de la page : l'UC n'écrit que dans la mémoire principale, mais toute page modifiée par l'UC est réécrite en mémoire de masse au moment où elle doit être remplacée en mémoire principale par une autre page.

La traduction de l'adresse virtuelle en adresse physique a lieu ***avant*** l'entrée de l'adresse dans la mémoire cache !



9. Les entrées/sorties (E/S, *I/O*) et leur gestion

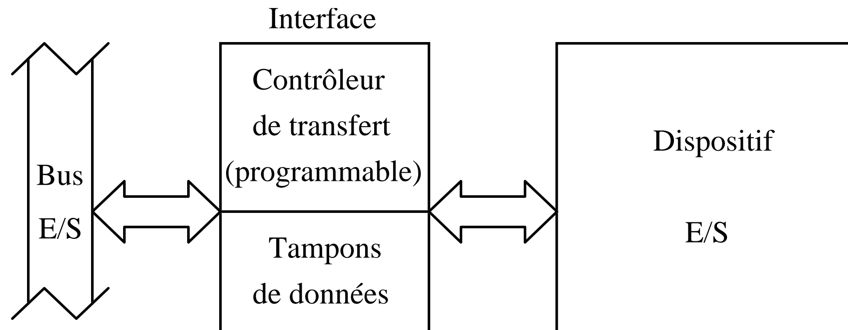
9.1. Types de dispositifs d'E/S

Quelques critères :
1° Direction de transfert :
  Entrée seulement : ex. clavier, souris ;
  Sortie seulement : ex. écran cathodique, imprimante ;
  Entrée et sortie : ex. disques et bandes magnétiques, connexions aux réseaux.
2° Partenaire du transfert :
  Humain (ex. clavier, souris, écran cathodique) ;
  Machine (ex. disques, imprimantes, réseaux).
3° Taux de transfert (E/S ↔ mémoire principale) :
  Lents (< 1 Koct/s, ex. clavier, imprimante matricielle) ;
  Vitesse moyenne (entre 1 Koct/s et 1000 Koct/s, ex. scanner, réseaux, imprim. laser) ;

Rapides (> 1000 Koct/s, ex. disques magnétiques, affichage graphique).

9.2. L'interface avec l'UC

En général, les dispositifs d'E/S (appelés aussi dispositifs périphériques) sont connectés à un bus spécifique (E/S) de l'ordinateur, et non pas directement au bus qui relie l'UC à la mémoire principale. La connexion se fait à travers une interface spécifique, programmable :



Activités possibles de l'UC, pour un dispositif périphérique donné :

1° l'UC écrit des codes de contrôle dans des registres internes de l'interface ou même du périphérique ;
2° l'UC lit le contenu des registres d'état de l'interface, voir même du périphérique ;
3° l'UC écrit des données destinées au périphérique dans le tampon ;
4° l'UC lit du tampon des données venant du périphérique.

Il est donc nécessaire que l'UC puisse accéder aux registres et tampons mentionnés, pour tous les périphériques. La solution la plus utilisée est de traiter ces registres et tampons comme une zone de mémoire (*memory-mapped I/O*) et d'utiliser les instructions de transfert UC ↔ mémoire principale (solution adoptée par ex. pour le SPARC). Une solution alternative est d'avoir des instructions de l'UC dédiées aux transferts UC ↔ périphériques, ainsi que des signaux spécifiques sur le bus de contrôle (solution adoptée par ex. pour les INTEL 80x86).

9.3. Dialogue avec les périphériques, interruptions

Deux techniques peuvent être utilisées pour contrôler les transferts avec les périphériques :

1° La boucle programmée (*polling*) : l'UC lit périodiquement les registres d'état des interfaces et effectue les transferts demandés par les périphériques. Cela occupe en général inutilement l'UC, et en plus le temps de réponse à une demande spécifique peut devenir imprévisible (en raison de la présence de plusieurs périphériques, tous étant traités dans la même boucle et donc ayant la même priorité).

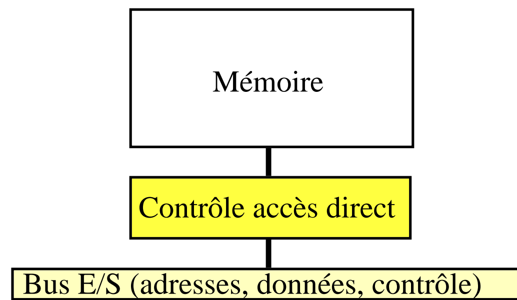
2° Les interruptions matérielles : quand un périphérique demande un transfert, il le signale (par des lignes spécifiques du bus de contrôle) à l'UC. Si le niveau de priorité de l'interruption (codé sur ces lignes du bus de contrôle) est supérieur au niveau de priorité du processus exécuté par l'UC, celle-ci interrompt le processus courant, sauvegarde son contexte, lit (éventuellement) le vecteur d'interruption du bus de données et détourne le contrôle vers un programme de traitement de l'interruption (identifié en général par le vecteur d'interruption). Ce programme effectue le transfert demandé, ensuite le contexte antérieur de l'UC est restauré et l'exécution du processus interrompu est reprise.
Si le niveau de priorité de l'interruption est inférieur au niveau de priorité du processus exécuté par l'UC, l'interruption n'est pas prise en considération (reste en attente). Différents périphériques peuvent avoir différents niveaux de priorité.

Des solutions mixtes sont parfois employées.

9.4. Accès direct à la mémoire (*Direct Memory Access, DMA*)

Le transfert de quantités importantes de données entre la mémoire principale et un périphérique (par exemple le contenu d'une page virtuelle entre le disque et une page physique) demande à l'UC un nombre important de cycles. Pour cette raison, des contrôleurs spécialisés dans les transferts mémoire principale ↔ dispositifs d'E/S ont été développés (accès direct à la mémoire, *DMA*). L'UC programme (composante du système d'exploitation) le contrôleur *DMA* pour effectuer un ou plusieurs transferts. Ce contrôleur décharge ensuite l'UC des responsabilités associées au(x) transfert(s) à effectuer. Une partie des interruptions matérielles en provenance des

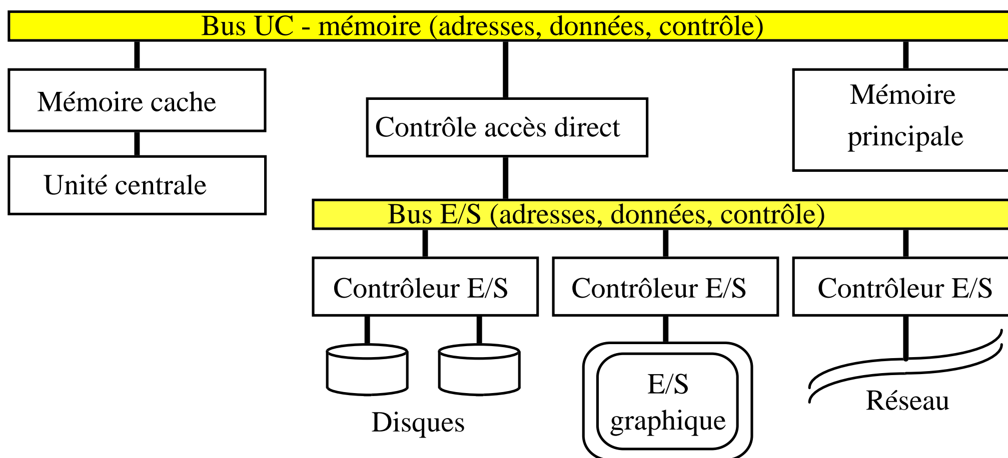
périphériques peuvent être adressées directement au contrôleur *DMA*. Les versions évoluées de contrôleurs *DMA* peuvent exécuter leur propre programme et s'appellent processeurs d'E/S (*Input-Output Processors, IOP*).



9.5. L'interface avec la mémoire

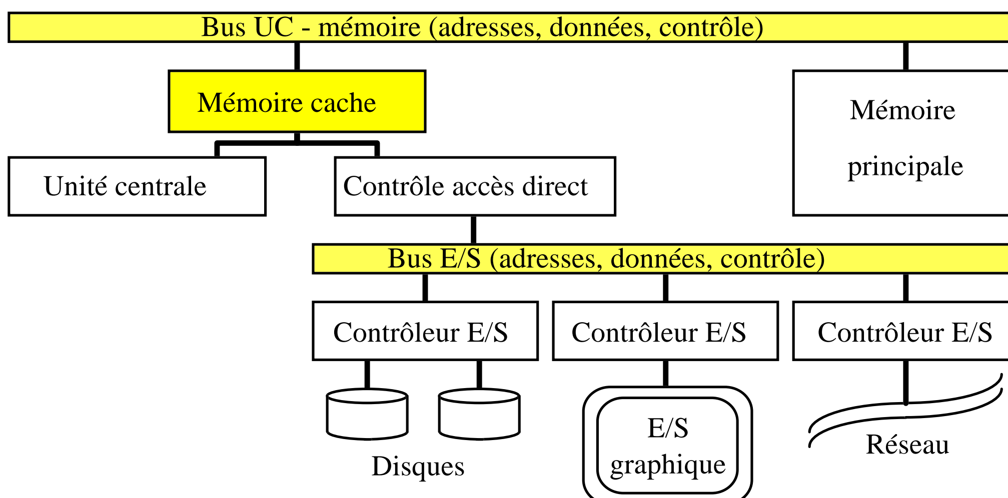
Solutions de connexion entre le bus d'E/S et le bus UC — mémoire principale quant une mémoire cache est présente :

1° Connexion directe avec le bus UC — mémoire principale :



Cette solution est très performante (peu d'interférences entre les transferts E/S et le travail de l'UC), mais des incohérences peuvent apparaître entre les informations gardées dans la cache et celles se transférées entre la mémoire principale et les périphériques. Les solutions à ce problème combinent des composantes matérielles et logicielles.

2° Connexion avec le bus local de l'UC (bus UC ↔ cache) :

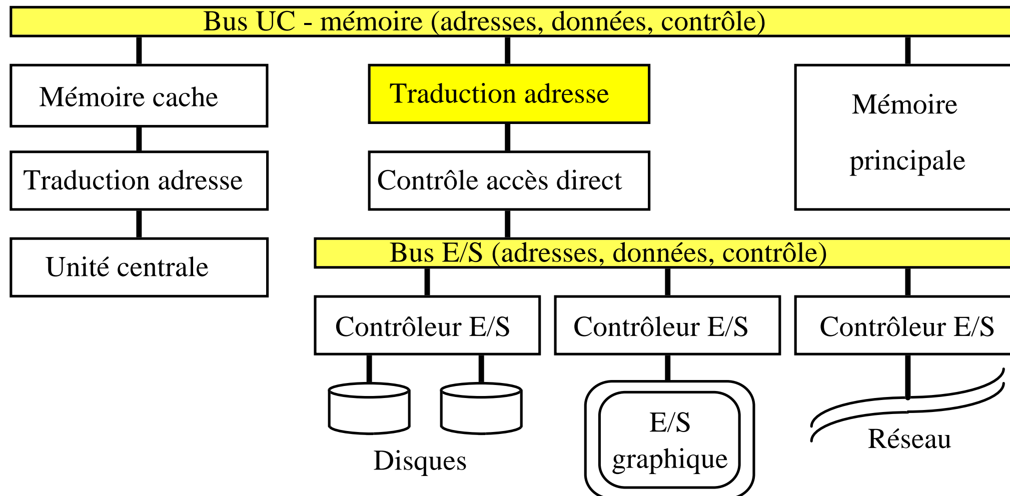


La solution est moins performante, mais aucune incohérence ne peut apparaître. Toutefois, la première solution est en général préférée.

Solutions possibles de connexion entre le bus d'E/S et le bus UC — mémoire principale quant la mémoire virtuelle est employée :

1° Utilisation de l'adresse physique par le contrôleur *DMA* ou par l'*IOP*. A première vue c'est la solution la plus simple, mais des difficultés apparaissent en cas notamment de transfert direct de plusieurs pages ou de transfert périodique de certaines pages (par ex. pour le système d'affichage graphique). Pour cette raison, la deuxième solution (solution suivante) est employée.

2° Utilisation de l'adresse virtuelle par le contrôleur *DMA* ou par l'*IOP* :



Le système d'exploitation gère les deux modules de traduction d'adresse, qui font appel à une même table de correspondances.