

Unité Centrale de Traitement : Processeur

VII.1 Introduction

L'unité centrale de traitement (CPU : Central Processing Unit), encore dénommée processeur ou microprocesseur, est l'élément de l'ordinateur qui interprète et exécute les instructions d'un programme. C'est le cerveau de l'ordinateur. Mais on trouve aussi des processeurs, dits spécialisés, qui peuvent décharger l'unité centrale et assurer des tâches en parallèle. Ceci est très fréquent pour la gestion des entrées/sorties.

Un processeur est aujourd'hui un circuit électronique à très haute densité d'intégration (ULSI : Ultra Large Scale Integration), qui peut compter quelques dizaines de millions de transistors. Le premier circuit de ce type a été créé par Intel en 1971 : le 4004 conçu pour équiper des calculatrices. Il comptait alors 2300 transistors pour 46 instructions. La loi de Moore, formulée en 1965 par un des fondateurs de la compagnie Intel, qui prédit un doublement des capacités des processeurs tous les 18-24 mois, a jusqu'à présent été relativement bien suivie. Il ne s'agit pas seulement de l'augmentation de la fréquence de fonctionnement ou du nombre de transistors. Les concepteurs cherchent aussi à augmenter la quantité de traitement par cycle d'horloge.

Une unité centrale se compose d'au moins deux unités fonctionnelles : l'unité de commande et l'unité de calcul. A l'origine celle-ci s'identifiait à l'unité arithmétique et logique, chargée de l'exécution des opérations booléennes et des opérations arithmétiques (addition, soustraction, multiplication, division, comparaison, etc.) pour des entiers. En parallèle à cette unité, on peut trouver une unité de calcul sur les réels ainsi qu'une unité de traitement dédiée aux opérations multimédia (traitement des images et du son).

A côté de ces deux unités fonctionnelles on trouve une interface de gestion des communications sur le bus externe, ainsi qu'une mémoire cache. Celle-ci est baptisée de premier niveau car située à proximité immédiate du cœur du processeur. Mais depuis quelques années, les concepteurs ont été amenés à embarquer également la mémoire de second niveau. Ces caches peuvent être scindés pour séparer les instructions et les données (architecture Harvard).

L'unité de commande contient une unité chargée du décodage des instructions, une unité pour le calcul des adresses des données à traiter. On y trouve également le séquenceur qui contrôle le fonctionnement des circuits de l'unité de calcul nécessaires à l'exécution de chaque instruction.

L'unité centrale comprend un certain nombre de registres pour stocker des données à traiter, des résultats intermédiaires ou des informations de commande. Parmi ces registres certains servent pour les opérations arithmétiques ou logiques, d'autres ont des fonctions particulières comme le registre instruction (RI) qui contient l'instruction à exécuter, le compteur ordinal (CO) qui pointe sur la prochaine instruction ou un registre d'état (PSW : Processor Status Word) contenant des informations sur l'état du système (retenue, dépassement, etc.).

La figure 1 présente un schéma général très simplifié de l'organisation de l'unité centrale, sans tenir compte de la présence d'une mémoire cache et de l'interface avec le bus externe.

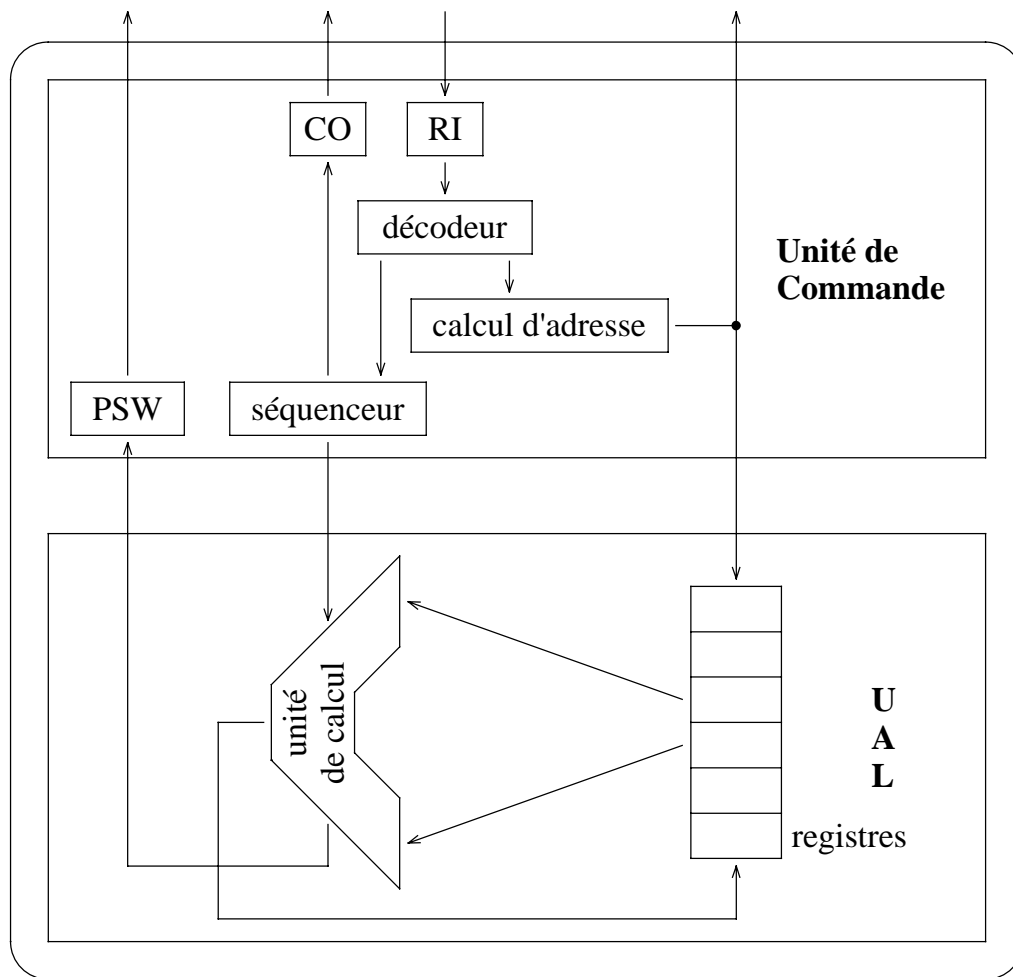


Figure 1

Avant d'étudier le fonctionnement de l'unité de commande nous allons passer rapidement au travers de certains éléments de l'ALU, c'est-à-dire le traitement numérique et logique des entiers. Nous avons, en fait, déjà étudié dans les chapitres précédents les principales fonctions intervenant dans l'ALU pour des entiers non signés : comparaison, addition, soustraction,

décalage, etc. Nous allons étendre cette étude aux entiers signés et pour cela nous commençons par étudier la représentation des nombres en machine.

VII.2 Représentation des nombres

VII.2.a Entiers non signés

Les entiers positifs ou nuls peuvent être codés en binaire pur. A l'aide de n bits nous pouvons représenter 2^n nombres compris entre 0 et $2^n - 1$:

$$A = \sum_{k=0}^{n-1} a_k 2^k$$

Sur le papier on utilise souvent les notations, plus compactes, octale ou hexadécimale : un chiffre (de 0 à 7) en octal représente 3 bits et un caractère (de 0 à F) hexadécimal permet de représenter 4 bits (table 1). Ainsi un octet s'écrit comme deux caractères hexadécimaux.

binaire	octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

binaire	hexadécimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Table 1

Si on n'utilise qu'une seule représentation binaire des entiers non signés, on rencontre plusieurs manières de coder les entiers signés. Nous allons en étudier quelques-unes. La table 2

présente une illustration sur 9 bits de ces différents codes : le bit de signe, le plus à gauche, y est mis en évidence.

Valeur décim.	signe et val. absolue	complément. vrai		complément. restreint		binaire décalé	
+127	0 11111111	0	11111111	0	11111111	1	11111111
...
+1	0 00000001	0	00000001	0	00000001	1	00000001
+0	0 00000000	0	00000000	0	00000000	1	00000000
-0	1 00000000			1	11111111		
-1	1 00000001	1	11111111	1	11111110	0	11111111
...
-127	1 11111111	1	00000001	1	00000000	0	00000001
-128		1	00000000			0	00000000

Table 2

VII.2.b Entiers signés sous la forme signe et valeur absolue

L'idée la plus naturelle pour coder des entiers signés consiste à ajouter le signe sous la forme d'un bit supplémentaire : 0 pour les entiers positifs et 1 pour les entiers négatifs. Soit pour n+1 bits :

$$A = \text{signe}(a_n) \sum_{k=0}^{n-1} a_k 2^k \quad \text{avec} \quad \begin{cases} \text{signe}(0) = 1 \\ \text{signe}(1) = -1 \end{cases}$$

On code alors 2^{n+1} entiers compris entre $-(2^n - 1)$ et $2^n - 1$. Cependant le zéro possède deux représentations : +0 et -0, ce qui peut être un inconvénient, en particulier dans certains tests.

VII.2.c Entiers signés en complément vrai

Nous avons vu que la représentation en complément à deux facilite les opérations arithmétiques d'addition et de soustraction. Notons A^+ le complément vrai, ou complément à deux, d'un entier positif, défini par :

$$A^+ = \bar{A} + 1$$

Il est facile de vérifier que :

$$A + \bar{A} = 2^n - 1 \quad (\text{n bits à 1})$$

donc :

$$A + A^+ \equiv 0 \quad (n \text{ bits})$$

Par exemple :

$$\begin{array}{r} A \quad \quad 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \\ + A^+ \quad \quad \underline{1 \ 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 1} \\ \hline 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \end{array}$$

Le complément vrai d'un entier peut donc être assimilé à son opposé. Considérons le codage sur $n+1$ bits. Par convention nous choisissons d'écrire tout entier positif avec le bit de plus haut poids a_n à 0 et sa valeur algébrique sur n bits. Nous pouvons ainsi coder 2^n entiers positifs de 0 à $2^n - 1$. Prenons le complément vrai de tous ces mots de $n+1$ bits. Commençons par le zéro. Son complément est constitué de $(n+1)$ bits à 1. En additionnant 1, pour passer au complément vrai, la retenue se propage jusqu'au dernier bit et nous obtenons $(n+1)$ bits à 0 :

$$\begin{array}{r} A \quad \quad 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ \bar{A} \quad \quad 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \\ A^+ \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \end{array}$$

Dans cette représentation, il n'y a donc qu'un seul zéro. Tout autre entier positif possède au moins un bit à 1. Dans le complément tous les bits à 1 deviennent des 0. Lors de l'addition de 1, pour passer au complément vrai, le premier bit à 0 arrête la propagation de la retenue et les bits de plus haut poids restent inchangés :

$$\begin{array}{r} A \quad \quad 0 \ X \ X \ X \ 1 \ 0 \ 0 \ 0 \quad X = \text{indifférent} \\ \bar{A} \quad \quad 1 \ \bar{X} \ \bar{X} \ \bar{X} \ 0 \ 1 \ 1 \ 1 \\ A^+ \quad \quad 1 \ \bar{X} \ \bar{X} \ \bar{X} \ 1 \ 0 \ 0 \ 0 \end{array}$$

On remarque en particulier que le bit de plus haut poids de A^+ est toujours égal à 1. Ce bit a_n peut donc être interprété comme un bit de signe : 0 pour les entiers positifs et 1 pour les entiers négatifs.

Considérons un entier positif A et son opposé $B = -A$. Nous pouvons écrire leur représentation binaire respective :

$$A \rightarrow (0 \ a_{n-1} \ a_{n-2} \ \dots \ a_1 \ a_0) \quad \text{et} \quad B \rightarrow (1 \ b_{n-1} \ b_{n-2} \ \dots \ b_1 \ b_0)$$

Compte tenu de notre convention nous obtenons la valeur de l'entier positif A en utilisant la somme classique sur les n bits de bas poids :

$$A = \sum_{k=0}^{n-1} a_k 2^k$$

Comme la représentation binaire de B est le complément vrai de celle de A, nous pouvons écrire :

$$\begin{array}{r}
 \\
 \\
 + \\
 \hline
 1
 \end{array}$$

Soit encore :

$$\begin{array}{r}
 \\
 \\
 + \\
 \hline
 1
 \end{array}$$

C'est-à-dire :

$$\sum_{k=0}^{n-1} a_k 2^k + \sum_{k=0}^{n-1} b_k 2^k = 2^n$$

ou

$$A + \sum_{k=0}^{n-1} b_k 2^k = 2^n$$

Donc :

$$B = -A = \sum_{k=0}^{n-1} b_k 2^k - 2^n$$

En prenant en compte le bit de signe, nous pouvons rassembler les deux expressions précédentes donnant la valeur numérique associée à une représentation binaire en complément à deux sur n+1 bits sous une forme générale :

$$A = \sum_{k=0}^{n-1} a_k 2^k - a_n 2^n$$

Ce codage nous permet de représenter 2^{n+1} nombres entiers compris entre -2^n et $2^n - 1$.

VII.2.d Entiers en représentation biaisée

En binaire pur, avec $n+1$ bits nous pouvons représenter 2^{n+1} entiers compris entre 0 et $2^{n+1}-1$. Si nous soustrayons à chacun des nombres ainsi représentés la valeur médiane de cet intervalle, soit 2^n , nous obtenons des entiers signés compris entre -2^n et $2^n - 1$. Cela correspond à la représentation biaisée ou décalée. Pour $n+1$ bits le biais à soustraire est 2^n :

$$A = \sum_{k=0}^n a_k 2^k - 2^n$$

Nous pouvons remarquer que cette représentation est identique au complément vrai à une inversion de la convention du bit de signe près : 0 pour les valeurs négatives et 1 pour les valeurs positives. En effet l'expression précédente peut encore s'écrire :

$$A = \sum_{k=0}^{n-1} a_k 2^k - (1 - a_n) 2^n$$

En particulier, le bit de signe du zéro est également 1.

VII.2.e Nombres fractionnaires en virgule fixe

Pour le codage des entiers on attribue un poids 1 au bit de poids le plus faible. Si nous lui attribuons un poids 2^{-m} avec n bits nous pouvons écrire :

$$F = \sum_{k=0}^{n-1} a_k 2^{k-m} = \frac{\sum_{k=0}^{n-1} a_k 2^k}{2^m} = \frac{A}{2^m}$$

Les m bits de poids faibles représentent la partie fractionnaire du nombre et les $n-m$ bits de poids forts sa partie entière. La position de la virgule est fixée par le choix de m . Cette représentation est applicable aux représentations signées.

En valeur absolue la plus petite valeur représentée correspond à 2^{-m} , et la grande à 2^{n-m} (ou presque), soit une gamme dynamique (rapport entre ces deux valeurs extrêmes) de 2^n .

VII.2.f Nombres en virgule flottante

En calcul scientifique on a souvent besoin de manipuler des nombres très grands ou très petits. Pour cela on utilise la notation exponentielle, sous la forme :

$$A = \pm M 2^E \quad \text{avec} \quad 2^{-1} \leq M < 1$$

où M est la mantisse et E l'exposant. La précision de la représentation dépend du nombre de bits réservés à la mantisse. Les formats ont été, et sont encore, très variés selon les constructeurs et les processeurs. Il existe cependant des standards, comme la norme IEEE 754. Dans celle-ci la représentation en simple précision est codée sur 32 bits.

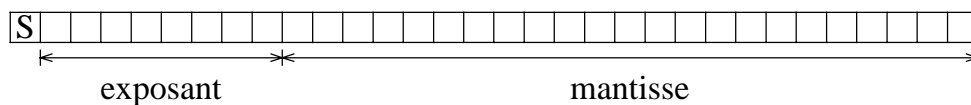


Figure 2

Ce format comporte le bit de signe de la mantisse (bit 31), suivi de $n = 8$ bits pour l'exposant en représentation biaisée (bits 23 à 30), puis de $m = 23$ bits pour la mantisse codée en virgule fixe (bits 0 à 22).

Sauf pour le zéro (codé par un mot nul : 000...0) nous pouvons toujours choisir l'exposant de telle façon que la mantisse soit comprise entre 0.5 et 1. Cela signifie que le bit de plus haut poids de la mantisse est toujours égal à 1 : la mantisse est dite normalisée. Ce bit peut être oublié (en particulier dans la norme IEEE 754), on gagne ainsi un bit pour la résolution. On dit alors que la mantisse est à bit caché. Le choix de la représentation biaisée pour l'exposant permet, dans ce cas, d'éviter toute confusion entre les représentations du zéro et du nombre 1/2. En effet $1/2 = 0,5 \cdot 2^0$. Le bit de signe de $E = 0$ en représentation biaisée étant 1, il y a un bit à 1 dans la représentation binaire avec mantisse à bit caché de 1/2.

VII.3 Opérations arithmétiques et logiques

Le traitement des opérations logiques (ET, OU inclusif, NON, OU exclusif, etc...) est direct en électronique numérique. Outre ces fonctions logiques l'unité arithmétique et logique est chargée de l'exécution d'opérations arithmétiques comme addition, soustraction, décalage et multiplication. Nous avons déjà étudié le principe de l'additionneur-soustracteur. Etudions rapidement le principe de la multiplication.

La multiplication ou la division d'un entier non signé par une puissance de deux (2^m) revient à effectuer un décalage de m cases respectivement vers la gauche ou la droite et à remplacer les cases "vides" par des 0. Considérons un entier non signé codé sur n bits :

$$A = \sum_{k=0}^{n-1} a_k 2^k$$

Multiplions ce nombre par 2^m :

$$B = 2^m A = \sum_{k=0}^{n-1} a_k 2^{k+m} = \sum_{k=m}^{n+m-1} a_{k-m} 2^k$$

Il faut donc $n+m$ bits pour coder le résultat et nous pouvons écrire pour B :

$$B = \sum_{k=0}^{n+m-1} b_k 2^k \quad \text{avec} \quad \begin{cases} b_k = 0 & \forall k \in [0, m-1] \\ b_k = a_{k-m} & \forall k \in [m, n+m-1] \end{cases}$$

Ainsi lorsqu'on multiplie deux nombres de n bits il peut falloir jusqu'à $2n$ bits pour contenir le résultat. Dans le cas des nombres signés en complément vrai il faut apporter une attention particulière au bit de signe. Avant la multiplication on procède à une extension de signe pour passer à $2n$ bits. C'est-à-dire qu'on recopie le bit de signe dans les n bits supplémentaires de haut poids (fig. 3).

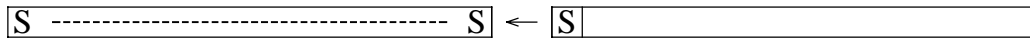


Figure 3

Considérons en effet un entier signé A codé en complément vrai sur n bits :

$$A = \sum_{k=0}^{n-2} a_k 2^k - a_{n-1} 2^{n-1}$$

et un nombre B obtenu à partir de A par extension de signe sur m bits ($m > n$) :

$$\begin{cases} b_k = a_k & \forall k \in [0, n-1] \\ b_k = a_s = a_{n-1} & \forall k \in [n-1, m-1] \end{cases}$$

Vérifions que A et B représentent la même quantité algébrique.

$$B = \sum_{k=0}^{m-2} b_k 2^k - b_{m-1} 2^{m-1}$$

Développons :

$$\begin{aligned}
 B &= \sum_{k=0}^{n-2} a_k 2^k + \sum_{k=n-1}^{m-2} a_{n-1} 2^k - a_{n-1} 2^{m-1} \\
 &= \sum_{k=0}^{n-2} a_k 2^k + a_{n-1} \left(\sum_{k=n-1}^{m-2} 2^k - 2^{m-1} \right) \\
 &= \sum_{k=0}^{n-2} a_k 2^k + a_{n-1} 2^{n-1} \left(\sum_{k=0}^{m-n-1} 2^k - 2^{m-n} \right) \\
 &= \sum_{k=0}^{n-2} a_k 2^k + a_{n-1} 2^{n-1} \left[(2^{m-n} - 1) - 2^{m-n} \right] \\
 &= \sum_{k=0}^{n-2} a_k 2^k - a_{n-1} 2^{n-1}
 \end{aligned}$$

Ce que nous voulions montrer.

Considérons maintenant la multiplication par 2 d'un entier signé A codé sur n bits :

$$\begin{aligned}
 A &= \sum_{k=0}^{n-2} a_k 2^k - a_{n-1} 2^{n-1} \\
 B = 2A &= \sum_{k=0}^{n-2} a_k 2^{k+1} - a_{n-1} 2^n \\
 &= \sum_{k=1}^{n-1} a_{k-1} 2^k - a_{n-1} 2^n \\
 &= \sum_{k=0}^{n-1} b_k 2^k - b_n 2^n
 \end{aligned}$$

Le résultat nécessite donc n+1 bits et nous avons pour le codage de B :

$$\begin{cases} b_0 = 0 \\ b_k = a_{k-1} \quad \forall k \in [1, n] \end{cases}$$

Donc après extension du bit de signe une multiplication par une puissance de deux (2^m) est réalisée en décalant de m cases vers la gauche le mot binaire et en injectant des 0 dans les m bits de bas poids. Cette technique conserve automatiquement le signe du résultat.

Réalisons la division entière (sans partie fractionnaire) par 2 d'un entier signé A :

$$\begin{aligned}
 B = \frac{A}{2} &= \sum_{k=1}^{n-2} a_k 2^{k-1} - a_{n-1} 2^{n-2} \\
 &= \sum_{k=0}^{n-3} a_{k+1} 2^k + (1-2) a_{n-1} 2^{n-2} \\
 &= \sum_{k=0}^{n-3} a_{k+1} 2^k + a_{n-1} 2^{n-2} - a_{n-1} 2^{n-1} \\
 &= \sum_{k=0}^{n-2} a_{k+1} 2^k - a_{n-1} 2^{n-1} \\
 &= \sum_{k=0}^{n-2} b_k 2^k - b_{n-1} 2^{n-1}
 \end{aligned}$$

Nous avons donc pour le codage de B sur n bits :

$$\begin{cases} b_k = a_{k+1} & \forall k \in [0, n-2] \\ b_{n-1} = a_{n-1} = a_s & \text{conservation du bit de signe} \end{cases}$$

Pour une division il faut donc conserver et propager le bit de signe dans le décalage à droite :

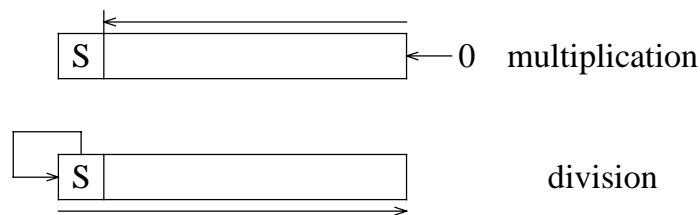


Figure 4

La multiplication de deux entiers signés de n bits en complément vrai se ramène à une combinaison de décalages et d'additions et éventuellement une soustraction. En effet :

$$A = \sum_{k=0}^{n-2} a_k 2^k - a_{n-1} 2^{n-1} \Rightarrow A \times B = \sum_{k=0}^{n-2} a_k 2^k B - a_{n-1} 2^{n-1} B$$

L'algorithme de Booth permet de réduire le nombre d'additions en repérant les séries de 0 et de 1. Dans le cas des zéros il n'y a que des décalages à effectuer. Dans le cas d'une série de 1 il suffit de se rappeler que :

$$\sum_{k=p}^m 2^k = 2^p \sum_{k=0}^{m-p} 2^k = 2^p (2^{m-p+1} - 1) = 2^{m+1} - 2^p$$

Donc :

$$\sum_{k=p}^m 2^k B = (2^{m+1} - 2^p) B$$

Les m-p+1 décalages et additions sont remplacés par deux décalages, une addition et une soustraction.

La figure 5 donne une représentation symbolique de l'ALU, comprenant deux opérandes en entrée A et B, une destination D = F(A, B), n lignes en entrée permettant de sélectionner la fonction F à exécuter, d'apporter un éventuel report de retenue et de synchroniser le fonctionnement de l'unité, ainsi que m sorties indiquant une éventuelle retenue (carry), un dépassement de capacité (overflow), un résultat négatif, nul ou positif, etc. Les opérandes et la destination sont toujours des registres.

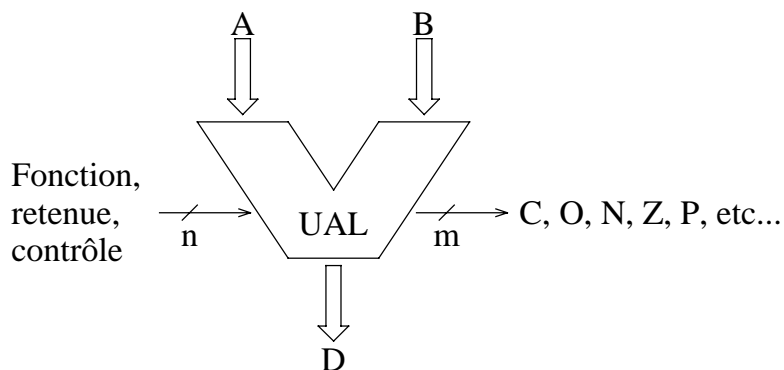


Figure 5

Les premiers ordinateurs disposaient d'opérateurs spécifiques câblés. Aujourd'hui dans les ordinateurs modernes on trouve de moins en moins de circuits de ce type. Il n'existe par exemple

plus de circuits propres à la multiplication ou à la division. L'enchaînement des actions à entreprendre pour réaliser une multiplication (décalages, additions et soustractions) est géré par l'unité de commande. Il en est de même pour le chargement des opérandes, la sélection des opérations à exécuter, etc.

VII.4 Unité de commande

L'unité de commande dirige le fonctionnement de tous les autres éléments de l'unité centrale en leur envoyant des signaux de commande. Les principaux éléments de l'unité de commande sont :

- le compteur ordinal (CO) : registre contenant l'adresse en mémoire où se trouve l'instruction à chercher;
- le registre instruction (RI) qui reçoit l'instruction qui doit être exécutée;
- le décodeur qui détermine l'opération à effectuer et les opérandes;
- le séquenceur qui génère les signaux de commande aux différents composants;
- l'horloge (interne ou externe) qui émet des impulsions permettant la synchronisation de tous les éléments de l'unité centrale.

Une horloge est un système logique, piloté par un oscillateur, qui émet périodiquement une série d'impulsions calibrées. Ces signaux périodiques constituent le cycle de base ou cycle machine. Nous avons déjà vu (dans le premier chapitre) les différentes phases de l'exécution d'une instruction. Un cycle d'instruction peut se décomposer en un cycle de recherche (instruction et opérandes) et un cycle d'exécution. On rencontre parfois le terme de cycle cpu pour indiquer le temps d'exécution de l'instruction la plus courte.

N'oublions cependant pas que les performances d'un ordinateur ne dépendent pas de la seule cadence de l'unité centrale. Elles dépendent également des mémoires et des bus, ainsi que de l'architecture, avec par exemple l'utilisation d'antémémoire (mémoire cache) pour anticiper les transferts des instructions et des données.

VII.5 Fréquence d'horloge

L'oscillateur fournissant le signal d'horloge peut être intégré sur la puce silicium du processeur. Ce n'est généralement pas le cas, pour au moins deux raisons.

Tout d'abord, la fréquence d'oscillation est sensible à la température. Or celle d'un processeur n'est pas très stable. Il serait possible d'observer une dérive fatale, la fréquence augmentant avec la température et la température augmentant avec la vitesse de fonctionnement. Cela poserait également des problèmes de synchronisation avec les composants externes au CPU.

Par ailleurs en embarquant une horloge dans la puce du processeur la fréquence de celui-ci serait figée par construction. Or les galettes de silicium ne sont jamais parfaitement homogènes. Les performances peuvent varier selon les zones. Le constructeur serait alors obligé

d'ajuster la fréquence de l'oscillateur au plus bas, en prenant un facteur de sécurité. Alors qu'actuellement, les processeurs sont triés et vendus à un prix variant selon leur vitesse de fonctionnement.

On préfère donc un oscillateur interne synchronisé sur un signal d'horloge externe, souvent fourni par un oscillateur à quartz. Nous allons étudier ce système de synchronisation, basé sur une boucle à verrouillage de phase.

VII.5.a Boucle à verrouillage de phase

Nous ne présentons ici que le principe, très schématisé, du fonctionnement d'un PLL (Phase Locked Loop). La structure minimale consiste en une boucle contenant :

- un comparateur de phase;
- un oscillateur commandé en tension.

On trouve de nombreuses utilisations de PLL dans les ordinateurs : horloge CPU, enregistrement et lecture de données sur un disque dur ou un disque optique, transmission de données, etc.

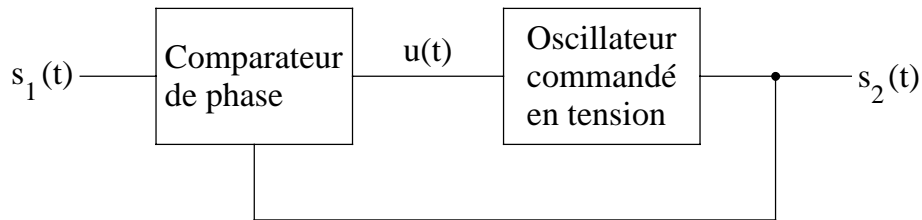


Figure 6

Le comparateur de phase est un circuit fournissant une tension dont la valeur dépend de la différence de phase entre deux signaux périodiques. Nous illustrons notre analyse en ne considérant que les signaux sinusoïdaux. Mais cela ne retire rien à la généralité de l'étude (cf. : Fourier).

$$\begin{cases} s_1(t) = S_1 \cos[\varphi_1(t)] \\ s_2(t) = S_2 \cos[\varphi_2(t)] \end{cases}$$

Donc par définition, la tension à la sortie du comparateur de phase peut se mettre sous la forme :

$$u(t) = S_1 S_2 f[\varphi_1(t) - \varphi_2(t)] = S_1 S_2 f[\Delta\varphi(t)]$$

où f est une fonction de période 2π . Nous supposons que les amplitudes S_1 et S_2 sont constantes. Pour simplifier les notations, en nous limitant aux termes importants, nous prenons $S_1 = S_2 = 1$, ce qui nous donne :

$$u(t) = f[\Delta\varphi(t)]$$

Cette tension est utilisée pour commander l'oscillateur. L'oscillateur commandé en tension (VCO : Voltage Controlled Oscillator) délivre un signal, ici sinusoïdal, dont la pulsation instantanée est une fonction linéaire de la tension de commande. C'est-à-dire :

$$\omega_2(t) = \frac{d\varphi_2(t)}{dt} = \omega_0 + k_\omega u(t)$$

ω_0 est la pulsation de repos du VCO;

k_ω est la sensibilité du VCO.

Si nous explicitons la tension de commande fournie par le comparateur de phase dans cette expression il vient :

$$\omega_2(t) = \frac{d\varphi_2(t)}{dt} = \omega_0 + k_\omega f[\Delta\varphi(t)]$$

Dérivons cette expression par rapport au temps :

$$\frac{d^2\varphi_2(t)}{dt^2} = k_\omega \frac{df}{d\Delta\varphi} \frac{d\Delta\varphi(t)}{dt}$$

Nous notons k_φ la sensibilité du comparateur de phase, définie par :

$$k_\varphi = \frac{df}{d\Delta\varphi}[\Delta\varphi(t)] = \frac{du(t)}{d\Delta\varphi(t)}$$

L'équation différentielle peut alors s'écrire :

$$\frac{d^2\varphi_2(t)}{dt^2} = k_\omega k_\varphi \left[\frac{d\varphi_1(t)}{dt} - \frac{d\varphi_2(t)}{dt} \right]$$

Soit :

$$\frac{d^2\varphi_2(t)}{dt^2} + k_\omega k_\varphi \frac{d\varphi_2(t)}{dt} = k_\omega k_\varphi \frac{d\varphi_1(t)}{dt}$$

La solution de cette équation différentielle peut se mettre sous la forme d'une somme de deux termes :

- le régime transitoire : solution générale de l'équation sans second membre;
- le régime permanent : solution particulière de l'équation complète.

Si nous avons pour le signal s_1 une pulsation constante :

$$\varphi_1(t) = \omega_1 t + \varphi_1^0$$

nous pouvons essayer une solution particulière telle que :

$$\frac{d^2 \varphi_2(t)}{dt^2} = 0$$

Il vient alors :

$$\frac{d \varphi_2(t)}{dt} = \frac{d \varphi_1(t)}{dt}$$

C'est-à-dire que les deux phases sont égales à une constante additive près :

$$\varphi_2(t) = \varphi_1(t) + \theta = \omega_1 t + \varphi_1^0 + \theta$$

La pulsation du signal en sortie du VCO est donc égale à celle du signal s_1 et le déphasage entre les deux signaux est fixe :

$$\begin{cases} \omega_2 = \omega_1 \\ \Delta\varphi^0 = \varphi_1(t) - \varphi_2(t) = \varphi_1^0 - \varphi_2^0 = \theta \end{cases}$$

Ainsi, après une phase transitoire, la boucle s'accroche ou se verrouille. Le déphasage entre les deux signaux est tel que la tension de commande permet d'obtenir une pulsation identique à celle du signal de référence.

$$\omega_2 = \omega_0 + k_\omega f(\Delta\varphi^0) = \omega_1 \quad \Rightarrow \quad f(\Delta\varphi^0) = \frac{\omega_0 - \omega_1}{k_\omega}$$

Ce qui nous donne pour le déphasage :

$$\Delta\varphi^0 = f^{-1}\left(\frac{\omega_0 - \omega_1}{k_\omega}\right)$$

Nous avons simplifié la résolution de l'équation différentielle en supposant la sensibilité du comparateur k_φ constante. Si ce n'est pas le cas, cela ne change rien au régime permanent. Par contre cela peut influencer sur le régime transitoire et en particulier sur les conditions d'accrochage de la boucle.

VII.5.b Multiplicateur de fréquence

Cependant la transmission d'un signal d'horloge de très haute fréquence pose de nombreux problèmes techniques. Pour ne pas limiter la fréquence de fonctionnement de l'unité centrale à celle de ce signal externe on utilise un multiplicateur de fréquence. Il s'agit en fait d'un diviseur de fréquence, rencontré en logique séquentielle, placé dans la contre-réaction de la boucle à verrouillage de phase (fig. 7). Si celui-ci divise la fréquence ω_2 par n , la boucle est verrouillée si :

$$\omega'_2 = \frac{\omega_2}{n} = \omega_1$$

C'est-à-dire que l'oscillateur émet un signal avec une fréquence égale à n fois la fréquence du signal de référence. Ce coefficient de multiplication n est chargé dans l'unité centrale lorsque celle-ci est initialisée.

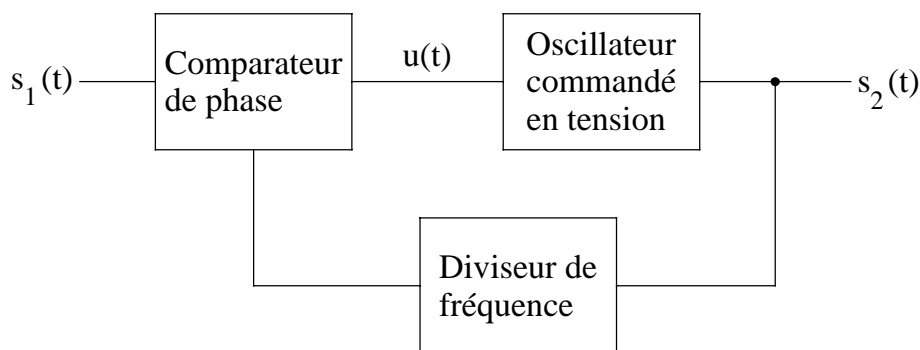


Figure 7

VII.5.c Fréquence d'horloge, gravure et puissance dissipée

L'augmentation de la fréquence d'un processeur pose quelques problèmes techniques, en particulier au niveau de la puissance thermique dissipée et des perturbations électromagnétiques. Plus un circuit fonctionne rapidement, plus les intensités des courants, et donc la puissance consommée, sont élevées. Une façon de réduire cet inconvénient consiste à utiliser une tension d'alimentation plus faible, par exemple 3.3 V au lieu de 5 V. Par ailleurs plus les signaux sont rapides plus ils induisent, par couplage capacitif, des parasites dans leur voisinage. En baissant la tension on diminue l'amplitude de ces parasites, mais on augmente la sensibilité du circuit aux parasites externes.

L'augmentation de la densité d'intégration fournit également des réponses à ces difficultés. Mais commençons par résumer succinctement le processus de fabrication d'un circuit intégré. On

part d'un "lingot" de silicium : un monocristal sans défaut ultra pur (cylindre de 6 ou 8 pouces de diamètre aujourd'hui, mais 12 et 16 pouces en développement). Celui-ci est ensuite découpé en très fines tranches ou galettes (environ 500-600 μm d'épaisseur). Ces galettes sont ensuite polies mécaniquement et chimiquement. Pour réaliser les transistors, les résistances et les connexions il faut doper certaines zones avec des impuretés de type donneur ou accepteur. Pour cela on procède par couches superposées. Différentes techniques sont utilisées pour déposer, faire croître ou diffuser ces implants. Mais dans tous les cas il faut délimiter en surface les zones à traiter. Pour cela on utilise un masque : une plaquette de quartz sur laquelle est tracée l'image en chrome de la couche concernée (échelle 5 à 10). Par ailleurs, la galette est recouverte d'une pellicule de résine photorésistive. Le masque et la galette sont exposés à un rayonnement ultraviolet, qui provoque la polymérisation de la résine. Puis un révélateur retire la résine non polymérisée. Le dopage peut alors être réalisé, puis le reste de la résine est retiré à l'aide d'un solvant. On répète ces opérations quelques dizaines de fois.

Sur une galette on réalise un grand nombre de circuits. Ceux-ci sont tous testés et marqués avant que la galette ne soit découpée en plaquettes. Chaque plaquette est ensuite placée sur un support et les connexions avec les contacts extérieurs sont réalisés par des fils d'aluminium ou d'or de 30 μm de diamètre. L'ensemble est ensuite enfermé dans un boîtier en plastique ou en céramique.

La réduction du pas de la gravure présente de nombreux avantages. Cela permet évidemment de diminuer la surface de silicium utilisée, un gain d'un facteur 2 sur la gravure réduit la surface par 4. Il est alors possible :

- soit d'augmenter les fonctionnalités pour une même surface de silicium;
- soit de diminuer la surface de silicium, donc le coût, à fonctionnalités constantes.

Par ailleurs, cela permet une diminution du temps de transit des signaux et une diminution des intensités, donc une réduction de la puissance dissipée. Ces deux derniers avantages permettent donc augmenter la vitesse de fonctionnement. Par contre, la diminution de la surface de silicium rend l'évacuation de la chaleur plus difficile. Il faut faire appel à de la graisse ou de la pâte thermique pour assurer une bonne conduction entre la puce et le ventilateur, ou encore utiliser un refroidissement par plaque à effet Peltier.

Actuellement les constructeurs atteignent une finesse de 0.18 μm . En octobre 2000, IBM a annoncé la construction d'un site de production pour un pas inférieur à 0.1 μm . Il semble cependant qu'on approche des limites technologiques de la gravure. La finesse du tracé de la gravure est principalement limitée par la longueur d'onde de la source lumineuse utilisée pour l'insolation. Il faut en particulier limiter les phénomènes de diffraction. Les rayons X ou les électrons pourraient être employés dans un avenir plus ou moins proche. Au-delà la nanoélectronique est encore du domaine du laboratoire.

VII.6 Séquenceur

Le séquenceur est un automate distribuant, selon un chronogramme précis, des signaux de commande aux diverses unités participant à l'exécution d'une instruction. Il peut être câblé ou microprogrammé.

Un séquenceur câblé est un circuit séquentiel complexe comprenant un sous-circuit pour chacune des instructions à commander. Ce sous-circuit est activé par le décodeur.

L'idée de la microprogrammation a été introduite par Maurice Wilkes en 1951. Il est en effet toujours possible de remplacer un circuit logique par un transcodeur ou une ROM. Considérons un ensemble de n fonctions logiques dépendant de m variables logiques. Les valeurs de ces fonctions pour les $N = 2^m$ combinaisons possibles peuvent être calculées (table de vérité) et mémorisées sous forme de N mots de n bits. Ensuite en utilisant les m variables sous forme d'une adresse il est possible de restituer le résultat recherché.

De même pour reproduire une séquence d'opérations élémentaires il suffit d'un mot par "tranche" de temps. Cette série de mots constitue un microprogramme. Le code opération de l'instruction à exécuter peut être utilisé pour définir le pointeur sur la première microinstruction du microprogramme. En fonction du code opération le contenu d'un compteur est initialisé, puis celui-ci s'incrémente ensuite à chaque cycle d'horloge. La période de l'horloge utilisée à ce niveau peut être plus élevée que celle qui règle la cadence des autres éléments de l'unité centrale. Ce compteur sert à adresser une mémoire morte. La figure 8 illustre ce principe.

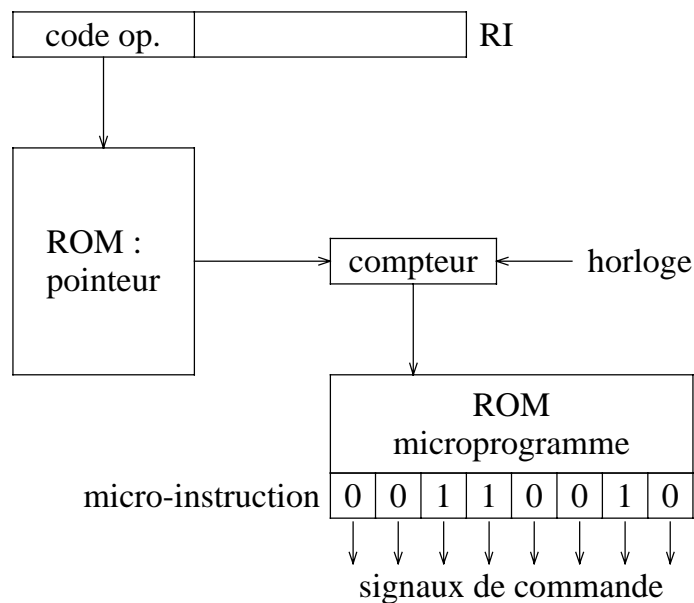


Figure 8

Le format des micro-instructions varie selon les machines. Le schéma de la figure 8 correspond à celui proposé initialement par Wilkes avec des micro-instructions longues où chaque bit correspond à une ligne de commande. On parle alors de microprogrammation horizontale. A l'extrême une autre solution consiste à utiliser des micro-instructions compactes nécessitant un décodage avant la génération des signaux de commande. La microprogrammation verticale est alors dite verticale. En microprogrammation horizontale la mémoire de commande comprend peu de micro-instructions, chacune comptant un grand nombre de bits. En microprogrammation verticale la longueur des micro-instructions est plus courte mais il y a un plus grand nombre de micro-instructions et il faut un décodage supplémentaire. Dans la pratique on rencontre des micro-instructions mixtes pour lesquelles certains bits agissent directement sur les lignes de commande associées alors que d'autres champs nécessitent un décodage. Un microprogramme peut également contenir des boucles, des tests et des ruptures de séquence. Le compteur est alors remplacé par un micro-séquenceur.

Intrinsèquement un séquenceur microprogrammé est plus lent qu'un séquenceur câblé. L'avantage et les gains en performance d'un séquenceur microprogrammé résident dans la simplicité de sa conception et la souplesse de son utilisation. Il est ainsi possible d'offrir un jeu d'instructions très complexes. Une instruction peut donc être équivalente à plusieurs instructions d'une autre machine. On gagne alors sur le temps de transfert des instructions. Par ailleurs cela permet une plus grande souplesse aux compilateurs de haut niveau pour optimiser le code objet. Il est également possible d'augmenter le nombre d'instructions sans augmenter la complexité, donc le coût, du processeur. Nous avons ici un premier exemple de l'imbrication du matériel et du logiciel dans la conception d'une architecture.

Pour programmer un ordinateur on utilise généralement des langages dits évolués ou de haut niveau : C, C++, Java, Basic, Fortran, Pascal, Ada, Assembleur, etc. Cependant l'unité centrale ne peut exploiter que les instructions machine : les codes binaires qui sont chargés dans le registre instruction.

Le terme de langage désigne un jeu d'instructions et de règles syntaxiques. A l'aide d'un langage évolué le programmeur écrit un code source. Celui-ci n'est pas directement exécutable par l'ordinateur. Il faut le traduire en code machine ou code objet. C'est le rôle des compilateurs ou assembleurs et des interpréteurs. Un interpréteur ne produit pas de code objet il traduit les instructions directement au fur et à mesure de l'exécution du programme.

La figure 9 schématise les différents niveaux de programmation. Lorsque l'utilisateur peut accéder au niveau de la microprogrammation la machine est dite microprogrammable.

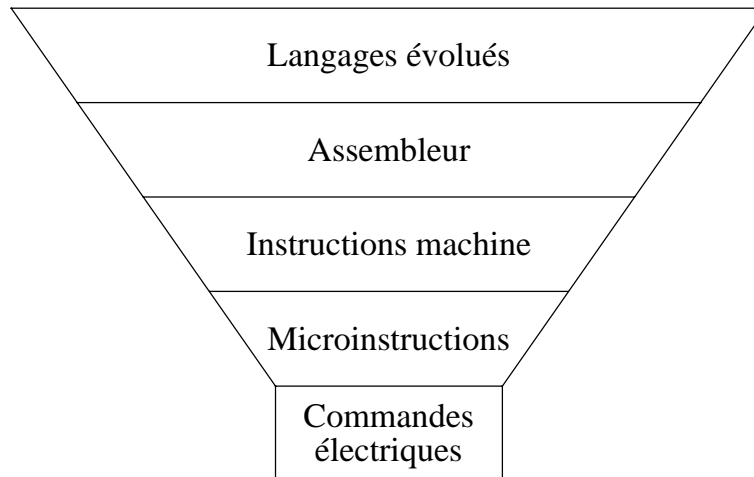


Figure 9

VII.7 Registres de l'unité centrale

Le nombre et le type des registres implantés dans une unité centrale font partie de son architecture et ont une influence importante sur la programmation et les performances de la machine. Nous voudrions ici passer en revue les registres fondamentaux, que l'on retrouve sur toutes les machines ou presque.

Compteur ordinal (CO) : Ce registre (Program Counter : PC) contient l'adresse de la prochaine instruction à exécuter. Après chaque utilisation il est automatiquement incrémenté du nombre de mots correspondant à la longueur de l'instruction traitée : le programme est exécuté en séquence. En cas de rupture de séquence (branchement conditionnel ou non, appel à une routine, etc.) il est chargé avec la nouvelle adresse. Le compteur ordinal, dont la taille dépend de l'espace adressable, n'est généralement pas accessible directement au programmeur.

Registre instruction (RI) : C'est le registre de destination dans lequel le CPU transfère l'instruction suivante à partir de la mémoire. Sa taille dépend du format des instructions machines. Le décodeur utilise le registre instruction pour identifier l'action (ou le microprogramme) à entreprendre ainsi que les adresses des opérandes, de destination ou de saut. Le programmeur n'a pas accès au registre instruction.

Accumulateur (ACC) : L'accumulateur est un registre de l'unité arithmétique et logique. Il a de nombreuses fonctions. Il peut contenir un des deux opérandes avant l'exécution et recevoir le résultat après. Cela permet d'enchaîner des opérations. Il peut servir de registre tampon pour les opérations d'entrées/sorties : dans certaines machines c'est le seul registre par lequel on peut échanger des données directement avec la mémoire. Sa taille est égale à la longueur des mots en mémoire. Il possède souvent une extension (Q), pour les multiplications, décalages, divisions, etc. Le registre ACC est accessible au programmeur et très sollicité. Certaines machines possèdent plusieurs accumulateurs.

Registres généraux ou banalisés : Ils permettent de limiter les accès à la mémoire, ce qui accélère l'exécution d'un programme. Ils peuvent conserver des informations utilisées fréquemment, des résultats intermédiaires, etc. Ils sont accessibles au programmeur.

Registres d'indice ou d'index : (XR) Ils peuvent être utilisés comme des registres généraux mais ils ont une fonction spéciale utilisée pour l'adressage indexé. Dans ce cas l'adresse effective d'un opérande est obtenue en ajoutant le contenu du registre d'index à l'adresse contenue dans l'instruction. Ce type d'adressage et de registre est très utile pour manipuler des tableaux. Le programmeur dispose alors d'instructions permettant l'incréméntation ou la décrémentation du registre d'index. En particulier les registres d'index peuvent être incrémentés ou décrémentés automatiquement après chaque utilisation. Dans certaines machines ces instructions sont applicables à tous les registres généraux, il n'y a alors pas de registre d'index spécifique.

Registre de base : A de très rares exceptions à l'intérieur d'un programme on ne fait référence qu'à des adresses relatives ou virtuelles. Par contre l'unité centrale a besoin de connaître les adresses physiques où se situent réellement instructions et données. Celles-ci dépendent de l'endroit où a été chargé le programme en mémoire, l'espace physique occupé par un programme pouvant ne pas être contigu. Le rôle des registres de base est de permettre le calcul des adresses effectives. Un registre de base contient une adresse de référence, par exemple l'adresse physique correspondant à l'adresse virtuelle 0. L'adresse physique est obtenue en ajoutant au champ adresse de l'instruction le contenu du registre de base. Le registre de base est encore utilisé quand le nombre de bits du champ adresse ne permet pas d'accéder à toute la mémoire.

Registre d'état (Program Status Word : PSW) : Une partie des bits de ce registre, aussi appelé registre condition, constitue des drapeaux (flags) qui indiquent certains états particuliers. Par exemple à la fin de chaque opération on peut y trouver le signe du résultat (Négatif, Zéro ou Positif), ainsi qu'une éventuelle retenue (Carry) ou un dépassement de capacité (Overflow). Ces bits indicateurs peuvent être testés pour déterminer la suite du déroulement du programme : branchements conditionnels. On trouve également le mode de fonctionnement de l'unité centrale. Deux modes sont possibles le mode utilisateur et le mode système ou superviseur. Dans le mode utilisateur certaines instructions sont interdites : elles provoquent un déroutement vers le système d'exploitation. Un bit peut également indiquer un déroulement pas à pas : demande de trace (T). Le registre peut aussi contenir le niveau de l'interruption en cours de traitement ou un masque des niveaux d'interruptions autorisés.

Registre pointeur de pile (PP) : Une pile est une zone mémoire dans laquelle les informations sont rangées de façon contiguë. Le pointeur de pile (Stack Pointer : SP) indique le sommet de la pile : la position de la dernière information enregistrée. Dans certaines machines le pointeur de pile indique la position où sera mémorisée la prochaine donnée. Le fonctionnement d'une pile est du type Dernier Entré Premier Sorti (LIFO : Last In First Out). Les deux principales opérations liées à la pile concernent l'ajout d'un élément dans la pile ou le retrait, souvent nommées respectivement PUSH et PULL. Lorsqu'une donnée est enregistrée dans la pile elle est placée à l'adresse qui suit celle du dernier mot stocké. Après l'opération le pointeur de pile est incrémenté. Lorsque un mot est retiré de la pile il correspond à la dernière information qui y a été entrée. Après l'opération le pointeur est décrémenté. Une pile est réservée à l'usage de l'unité centrale, en particulier pour sauvegarder les registres et l'adresse de retour en cas d'interruption ou lors de

l'appel d'une procédure. Le pointeur de pile est accessible au programmeur, ce qui est souvent source d'erreur. Certaines machines sont dotées de plusieurs pointeurs de piles.

Pour améliorer les performances d'un processeur il faut disposer du plus grand nombre de registres possible. On réduit ainsi les accès à la mémoire. De plus, il est préférable d'éviter de les spécialiser. On évite ainsi des transferts entre registres, par exemple pour calculer la valeur d'un indice et utiliser ensuite cet indice pour modifier une case d'un tableau.

VII.8 Structures des instructions au niveau machine

VII.8.a Format des instructions

Les ordinateurs sont capables d'effectuer un certain nombre d'opérations élémentaires. Une instruction au niveau machine doit fournir à l'unité centrale toutes les informations nécessaires pour déclencher une telle opération élémentaire : type d'action, où trouver le ou les opérandes, où ranger le résultat, etc. C'est pourquoi une instruction comporte en général plusieurs champs ou groupes de bits. Le premier champ contient le code opération. Les autres champs peuvent comporter des données ou l'identification des opérandes. La figure 10 donne quelques exemples d'instructions à n adresses, pour n = 0, 1 et 2. Sur certaines machines les instructions sont toutes de même longueur, sur d'autres cette longueur peut varier avec le code opération ou le mode d'adressage.

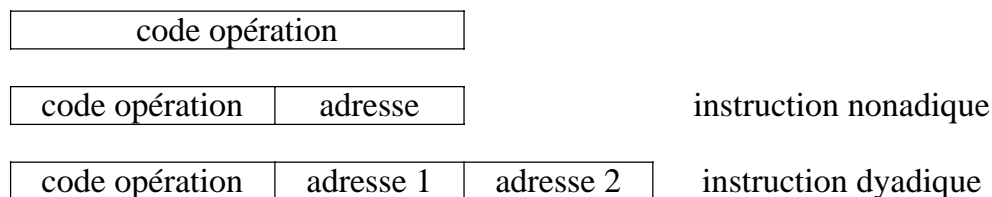


Figure 10

On distingue six groupes d'instructions :

- transferts de données : de mémoire à registre, de registre à registre, de registre à mémoire;
- opérations arithmétiques : addition, soustraction, multiplication et division;
- opérations logiques : ET, OU inclusif, NON, OU exclusif, etc.;
- contrôle de séquence : branchements conditionnels ou non, appel de procédure, etc.;
- entrées/sorties;
- manipulations diverses : décalage, conversion de format, permutation circulaire des bits, échange d'octets, incrémentation, etc.

Le choix de la longueur et du format des instructions est une étape très importante dans la conception d'une nouvelle architecture. C'est encore une affaire de compromis. La longueur des instructions se répercute sur la dimension de la mémoire centrale donc sur le coût : il faut deux fois plus de bits pour stocker un programme de n instructions de 32 bits qu'un programme de n

instructions de 16 bits. La longueur des instructions par rapport à celle du mot mémoire influence également le temps de traitement : il faut tenir compte du temps de transfert des mots qui constituent une instruction. Ce choix dépend des vitesses relatives d'accès mémoire et de traitement effectif par l'unité centrale. Le temps de recherche doit être minimisé pour les processeurs très rapides.

La largeur en bits de chacun des différents champs est également importante, en particulier pour le code opération. Le nombre de bits est déterminé par le nombre d'opérations distinctes envisagées : n bits autorisent 2^n instructions. Cependant toutes les opérations ne nécessitent pas forcément le même nombre de champs ou des champs de même longueur. Ainsi sur une machine même pour une longueur d'instruction donnée le format des instructions peut ne pas être fixe. Il peut dépendre du type d'opération. Pour illustrer le concept du code opération expansif imaginons des instructions de 16 bits découpées en quatre champs de 4 bits (fig. 11). Avec ce format nous pouvons définir 16 instructions à 3 adresses. C'est peu et toutes les instructions ne nécessitent pas trois adresses. Nous pouvons également définir 15 instructions à 3 adresses (code opération 0000 à 1110 dans le premier champ), 14 instructions à 2 adresses (code opération sur 8 bits 1111 0000 à 1111 1101 identifié par les quatre premiers bits à 1), 31 instructions à 1 adresse (code opération sur 12 bits 1111 1110 0000 à 1111 1111 1110 identifié par les sept premiers bits à 1) et 16 instructions sans champ adresse (identifiée par les douze premiers bits à 1). Soit un total de 76 instructions. Dans d'autres variantes la longueur et le format des instructions sont définis par les premiers bits du code.

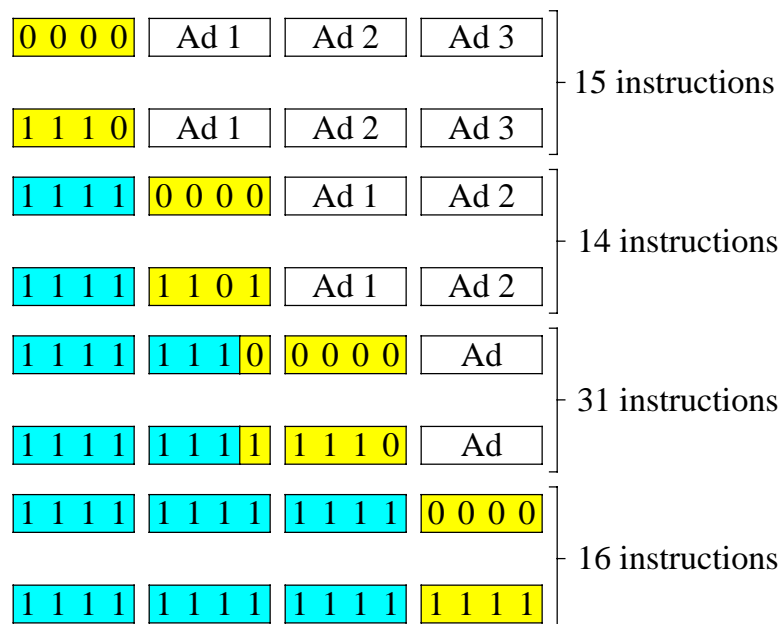


Figure 11

D'autre part comme la capacité mémoire ne cesse de croître les champs d'adresse demandent de plus en plus de bits. C'est pourquoi, pour éviter une inflation de certains registres comme le registre instruction, aujourd'hui on préfère les instructions à une adresse. On peut par

Après cette instruction le registre R2 contient le mot qui se situe à l'adresse 100 en mémoire, c'est-à-dire 250.

Adressage indirect : Le champ adresse (ou le mot suivant) contient l'adresse d'un pointeur : mot en mémoire qui contient l'adresse effective de l'opérande.

MOV (R1), R4

Après cette instruction R4 contient la valeur du mot dont l'adresse est contenue dans R1. Comme R1 vaut 100 on trouve 250 dans R4.

Adressage indexé : Ce mode d'adressage est très utile lorsqu'on travaille, par exemple, sur des tableaux. Considérons un bloc de n mots consécutifs débutant à l'adresse A . Le $k^{\text{ième}}$ mot se trouve à l'adresse $A + (k - 1)$. Pour référencer ce mot il est possible d'utiliser un registre d'index. L'adresse effective est calculée en additionnant le contenu de ce registre d'index à l'adresse qui se trouve dans le champ adresse de l'instruction. Sur certaines machines tous les registres généraux peuvent être utilisés comme registres d'index. La présence d'un registre d'index s'accompagne généralement de la possibilité d'incrémenter et décrémentation automatiques.

MOV R4, 100(R3)+
CLR 100(R3)

Avant la première opération R3 est nul, donc le contenu de R4 est transféré à l'adresse 100. Après le registre R3 est incrémenté. L'instruction suivante permet de mettre à zéro le contenu du mot à l'adresse suivante.

Adressage basé : L'adressage basé est comparable à l'adressage indexé mais cette fois l'adresse effective est obtenue en additionnant le contenu du registre de base au contenu du champ adresse de l'instruction. Ce mode d'adressage est utilisé par exemple en cas d'allocation dynamique de la mémoire : la position du programme en mémoire peut changer en fonction de la charge du système et il n'occupe pas toujours un espace contigu. Cette technique permet également de réduire le nombre de bits dans le champ adresse : le registre de base contient la première adresse d'un bloc de 2^k mots et l'adresse (sur k bits) contenue dans l'instruction représente le déplacement à l'intérieur du bloc.

Adressage relatif : L'adresse effective est obtenue en additionnant le contenu du compteur ordinal au contenu du champ adresse de l'instruction. Ce type d'adressage est utilisé par exemple dans des instructions de branchement.

N'oublions pas que le calcul de l'adresse effective peut nécessiter quelques opérations (addition par exemple). L'utilisation de certains modes d'adressage sophistiqués (le 68020 de Motorola dispose par exemple d'une cinquantaine de modes d'adressage) peut donc augmenter le temps de traitement d'une instruction.

VII.9 Traitement anticipé des instructions : pipeline

Nous avons vu que le traitement d'une instruction peut se découper en plusieurs phases. A chacune d'elles il est possible d'associer une unité fonctionnelle. Considérons, pour illustrer notre propos, un microprocesseur disposant de trois unités :

- unité de recherche (R);
- unité de décodage (D);
- unité d'exécution (E).

dont on suppose qu'elles remplissent leurs fonctions respectives en un cycle d'horloge.

La figure 13 schématise de déroulement d'une instruction. La phase "recherche" mobilise l'unité de recherche pendant un cycle d'horloge. La phase "décodage" occupe l'unité de décodage pendant un deuxième cycle et la phase "exécution" utilise l'unité d'exécution pendant un troisième cycle. Il faut donc trois cycles d'horloge pour traiter une instruction. La prise en compte de l'instruction suivante se fait à partir du 4^{ème} cycle, et ainsi de suite. L'exécution de N instructions requiert donc 3 N cycles d'horloge (fig. 14).

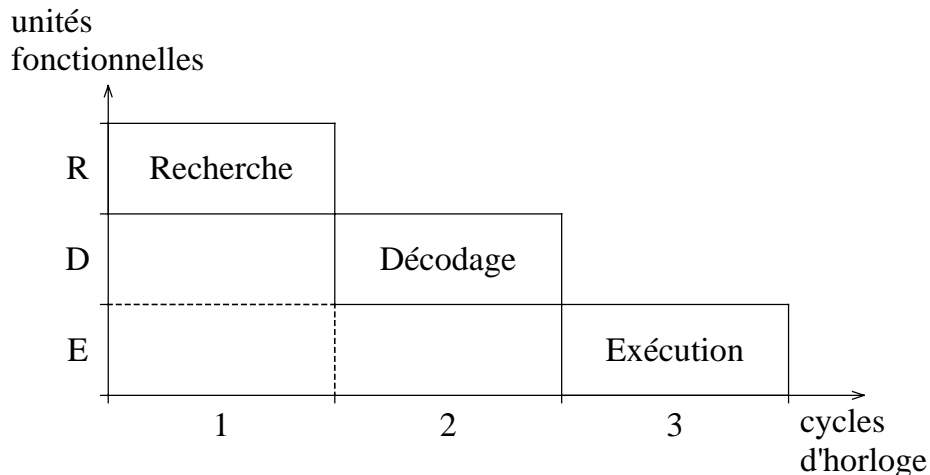


Figure 13

Cependant nous constatons alors qu'à chaque cycle d'horloge deux des trois unités fonctionnelles sont inemployées. Il y a moyen d'accélérer les calculs en demandant à l'unité de recherche d'aller chercher l'instruction suivante dès le deuxième cycle d'horloge. Cette instruction est ensuite décodée au cycle suivant, alors que l'unité de recherche continue également. Nous obtenons alors le diagramme schématisé sur la figure 15. Un processeur dans lequel ce traitement anticipé est possible est dit avec pipeline. Le compteur ordinal est incrémenté à chaque cycle d'horloge.

Dans un processeur avec pipeline il faut toujours 3 cycles pour exécuter chaque instruction, mais dès qu'une instruction est terminée il suffit d'attendre un cycle pour que la

suivante soit terminée, et non trois comme dans le cas précédent. Dans ces conditions il ne faut que $N+2$ cycles d'horloge pour exécuter N instructions. On calcule presque 3 fois plus vite.

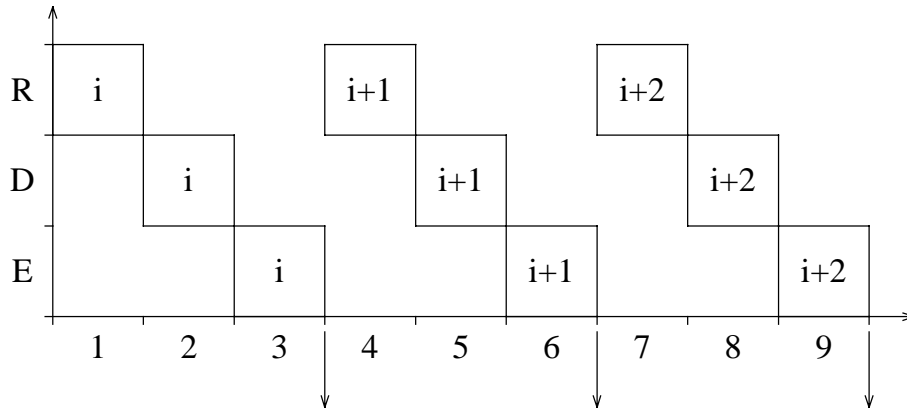


Figure 14

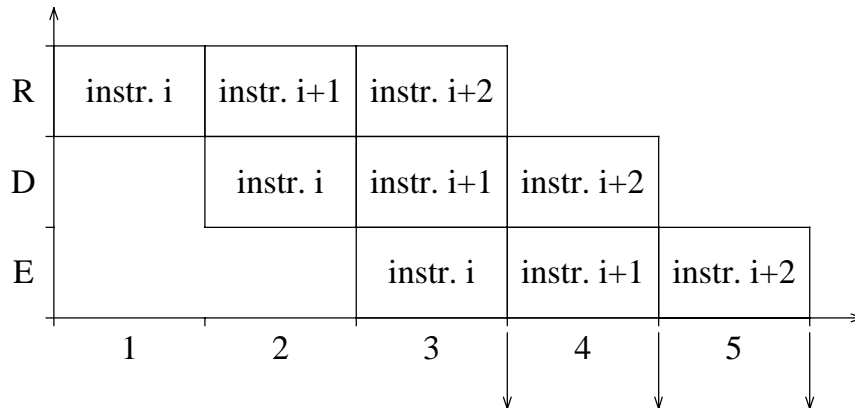


Figure 15

Il n'est cependant pas simple de bénéficier de ce gain de performance en totalité. Par exemple lors d'une rupture de séquence (branchement, fin de boucle, appel de sous-programme, etc.), il faut gérer les instructions dont le traitement a été anticipé. Il existe généralement deux types de branchements : simple ou retardé. Pour un branchement simple le traitement anticipé est interrompu et la chaîne est vidée avant d'exécuter le branchement. Il faut ensuite attendre à nouveau trois cycles pour obtenir un résultat. Pour un branchement retardé on termine le traitement commencé avant d'exécuter le branchement. Une bonne utilisation de ces branchements permet d'optimiser le fonctionnement du processeur. On peut trouver dans certains processeurs une unité qui cherche à anticiper la destination d'un branchement, de manière statistique par exemple. Ainsi en fin de boucle on revient systématiquement au début, sauf la dernière fois.

Le traitement des interruptions, que nous étudierons dans le prochain chapitre, pose également quelques problèmes. En général, pour simplifier la gestion de la reprise du programme

interrompu, les instructions engagées dans le pipeline sont traitées avant la prise en compte effective de l'interruption. Selon la longueur du pipeline cela peut générer un retard significatif.

Selon leur complexité, la durée de traitement de toutes les instructions n'est pas nécessairement identique. C'est particulièrement vrai en cas d'accès à la mémoire. Pour que les difficultés soient plus visibles considérons un processeur avec une quatrième unité fonctionnelle chargée de l'éventuelle lecture en mémoire des opérandes :

- unité de recherche (R);
- unité de décodage (D);
- unité de lecture (L);
- unité d'exécution (E).

Comme l'unité de lecture n'est pas toujours sollicitée le traitement d'une instruction prend 3 ou 4 cycles. Considérons le traitement de deux instructions consécutives I1 et I2, la première écrivant en mémoire, la seconde lisant en mémoire. La figure suivante illustre un conflit dans le traitement de ces deux accès. L'unité de lecture pour le besoin de l'instruction I2 aurait dû fonctionner au 4^{ème} cycle d'horloge. Mais simultanément l'unité d'exécution aurait été en train d'écrire en mémoire. Le traitement de l'instruction I2 est donc pénalisé d'un cycle.

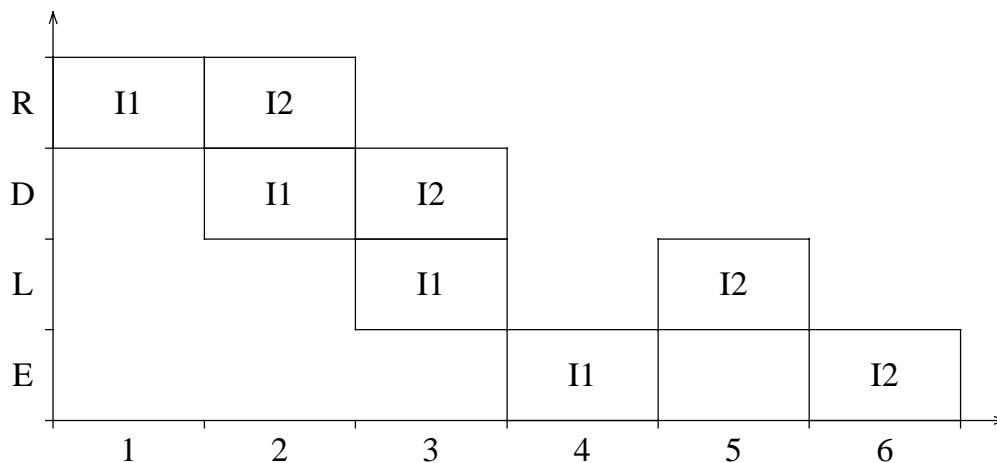


Figure 16

Nous avons simplifié. Le temps d'accès à la mémoire peut correspondre à plusieurs cycles d'horloge. D'autre part ce temps d'accès peut être variable. Ainsi deux instructions consécutives peuvent faire appel au même bloc mémoire ou à deux blocs différents. En fonction de l'architecture de la mémoire nous savons que les temps d'accès pour les deux instructions peuvent être identiques dans un cas et différents dans l'autre.

Par ailleurs, il peut y avoir dépendance entre deux instructions, la seconde ayant besoin d'une adresse calculée par la première par exemple.

La gestion de ces problèmes est généralement résolue par les compilateurs qui peuvent insérer des instructions NOP ou permuter des instructions du programme en tenant compte des spécificités du processeur.

VII.10 Unités d'exécution parallèles

Dans le précédent paragraphe nous avons étudié le découpage principalement de l'unité de commande en unités fonctionnelles pouvant travailler en parallèle. On peut également trouver plusieurs unités d'exécution ou de calcul.

L'unité de commande choisit l'unité à utiliser en fonction du type de l'instruction : ALU, FPU (Floating Point Unit) ou unité multimédia. Ce mode de fonctionnement ressemble à l'utilisation d'un coprocesseur. Un coprocesseur n'est pas utilisé pour exécuter du traitement parallèle mais pour augmenter ou améliorer les fonctionnalités du CPU. Celui-ci charge des registres pour fournir des données au coprocesseur, démarre ce dernier et se suspend jusqu'à ce que le traitement demandé soit effectué. En fin d'exécution le coprocesseur place le résultat dans un registre, réveille le processeur principal et se suspend jusqu'à une nouvelle requête. Un coprocesseur peut permettre d'augmenter l'ensemble des instructions ou réaliser certains calculs plus rapidement que l'unité centrale ("coprocesseur flottant").

Si le processeur est capable d'exécuter en parallèle plusieurs instructions différentes sur plusieurs unités d'exécution il est dit superscalaire. Il peut en particulier disposer de plusieurs unités arithmétiques et logiques parallèles. Les performances d'une machine superscalaire dépendent beaucoup du compilateur qui doit être capable d'identifier les instructions pouvant être exécutées simultanément, c'est-à-dire sans dépendance. L'approche Très Long Mot Instruction (VLIW : Very Long Instruction Word) est assez similaire, mais le parallélisme est totalement géré par le compilateur. Les instructions pouvant être exécutées simultanément sont placées dans un mot de grande longueur (128 bits pour i860 d'Intel).

On peut également imaginer un processeur avec une unité de contrôle mais disposant de n unités d'exécution identiques et de n ensembles de registres. Cet ordinateur peut donc exécuter simultanément le même calcul sur n jeux de données. Cette configuration est appelée SIMD (Single Instruction on Multiple Data) ou processeur vectoriel.

VII.11 Processus RISC

En première approximation la puissance d'un système peut être identifiée au temps d'exécution d'une tâche. Celui-ci peut s'écrire sous la forme suivante :

$$\text{temps/tâche} = \text{Nb.instructions/tâche} \times \text{Nb.cycles/instruction} \times \text{temps/cycle}$$

Le nombre de cycles par instruction est une moyenne, mais il doit tenir compte des transferts entre la mémoire et le processeur, y compris celui de l'instruction elle-même. Pour améliorer les performances, on peut donc agir sur chacun de ces trois termes, en tenant compte de leurs corrélations.

Le dernier terme correspond évidemment à la fréquence de l'horloge. L'évolution, dans un premier temps, a consisté à chercher la diminution du nombre d'instructions dans un programme, en utilisant des instructions et des modes d'adressage de plus en plus sophistiqués. Puis est apparue une philosophie différente qui cherche à réduire le nombre de cycles nécessaires à la réalisation de chaque instruction. Cette approche s'est appuyée sur les progrès réalisés en matière de matériel (mémoire virtuelle, cache, unité de gestion de mémoire, pipeline, etc.) et logiciel (compilateurs).

Le concept RISC (Reduced Instruction Set Computer) est apparu en 1975 chez IBM (IBM801 de John Coke), d'après des idées de Seymour Cray, avant d'être approfondi dans les années 80 par les universités de Stanford et Berkeley. Il repose sur la constatation que même les systèmes ou les applications les plus sophistiqués n'utilisent qu'une petite fraction du jeu d'instructions à leur disposition.

Des études statistiques, portant sur un grand nombre de systèmes d'exploitation et d'applications réels, ont montré que :

- Dans 80 % d'un programme on n'utilise que 20 % du jeu d'instructions.
- Les opérations les plus usitées sont :
 - les opérations d'échange entre l'unité centrale et la mémoire;
 - les appels à des sous-programmes.
- L'instruction d'appel d'une procédure est la plus gourmande en temps : sauvegarde et restitution du contexte et passage des paramètres.
- 80 % des variables locales sont des scalaires.
- 90 % des structures de données complexes sont des variables globales.
- La profondeur maximale d'appels imbriqués est en moyenne de huit. Une profondeur plus importante ne se rencontre que dans 1 % des cas.

Les processeurs classiques sont désormais appelés CISC (Complex Instruction Set Computer) par opposition au terme RISC.

L'apparition de mémoires très rapides à un coût très faible a constitué un des principaux arguments technologiques ayant favorisé les processeurs RISC. A l'origine les temps de décodage d'une instruction complexe et d'exécution du microcode correspondant étaient masqués par les temps d'accès à la mémoire. En réduisant celui-ci, le goulot d'étranglement s'est déplacé vers les unités de décodage et d'exécution. Il a donc fallu envisager un jeu d'instructions élémentaires câblées simplifiant l'architecture du processeur et réduisant les temps d'exécution. D'autre part, la capacité mémoire n'étant pratiquement plus limitée il n'y avait pas d'obstacle à augmenter la taille des programmes : la fonctionnalité de chaque instruction étant limitée il faut générer un plus grand nombre d'instructions (en pratique pas plus de 30 %).

Par contre la génération de code est plus compliquée. Les instructions complexes facilitent la traduction des programmes écrits en langage évolué. C'est également au compilateur de se charger de l'optimisation du code en fonction des caractéristiques des processeurs. Les recherches ayant débouché sur les processeurs RISC ont également porté sur les compilateurs. L'évolution des outils logiciels a vu l'émergence de compilateurs évolués permettant une optimisation

efficace des codes générés avec des gains de l'ordre de 20 % en temps d'exécution. Le développement des compilateurs, pris en charge par les concepteurs de systèmes, et le développement des architectures sont très interdépendants.

Passons en revue les idées de base qui font la spécificité des architectures RISC. L'objectif recherché est d'exécuter une instruction à chaque cycle d'horloge. Ceci est rarement réalisé sur la totalité du jeu d'instructions. L'obtention de cette performance repose sur la mise en œuvre de plusieurs principes :

- simplification des instructions;
- simplification du format : longueur fixe des instructions et des champs;
- restriction des modes d'adressage;
- séquenceur câblé pour une exécution rapide;
- utilisation intensive de registres très nombreux;
- limitation des accès mémoire à deux instructions : LOAD et STORE;
- utilisation de caches;
- traitement anticipé des instructions (pipeline) : ce qui permet d'obtenir un résultat à chaque cycle.

Le rôle du compilateur est essentiel pour l'optimisation du code engendré. Un effort particulier est fait sur les points suivants :

- allocation optimale des registres;
- élimination des redondances;
- optimisation des boucles, en ne conservant à l'intérieur que ce qui est modifié;
- optimisation du pipeline;
- optimisation du choix des instructions.

Le compilateur doit être capable d'exploiter au maximum les caractéristiques de l'architecture.

La figure 17 illustre l'influence du temps d'accès de la mémoire sur les performances de deux processeurs (RISC et CISC) cadencés à 20 MHz. La performance maximale de l'architecture RISC sert de référence (performance relative = 1). On observe une nette différence de comportement entre ces deux courbes. Pour un temps d'accès mémoire inférieur ou égal à un cycle d'horloge (50 ns) il n'y a pas amélioration des performances pour l'architecture CISC qui sature. Par contre dans les mêmes conditions on constate un gain important (ici un facteur 2) en performance pour le processeur RISC. Par contre celui-ci n'est compétitif que lorsqu'il est associé à une mémoire rapide. Cette figure explique le développement presque exclusif des architectures CISC jusqu'à l'apparition d'une nouvelle génération de mémoires rapides.

La simplicité des processeurs RISC fournit au moins deux autres avantages : le coût du développement et la surface de silicium sont notablement réduits. Plus simple un processeur RISC nécessite moins de temps et moins de main d'œuvre pour sa conception et sa mise au point. Par exemple le prototype RISC-II de l'université de Berkeley a été développé en deux ans par deux étudiants. De même le processeur ARM a été élaboré en dix-huit mois par une équipe de quatre personnes (6 hommes-ans). Par contre le 80386 d'Intel a demandé 100 hommes-ans. Le risque d'erreur de conception pouvant avoir un impact sur la clientèle avec retour de composants

par exemple est aussi beaucoup plus faible. Le rapport coût/performance est donc largement en faveur des processeurs RISC. Par contre le développement des compilateurs est beaucoup plus long. Dans un processeur CISC l'unité de contrôle occupe plus de 50 % de la surface de la plaquette silicium. Dans un processeur RISC elle en occupe moins de 20 %, il reste donc plus de place pour intégrer registres, caches, unités d'exécution parallèles et unités de gestion.

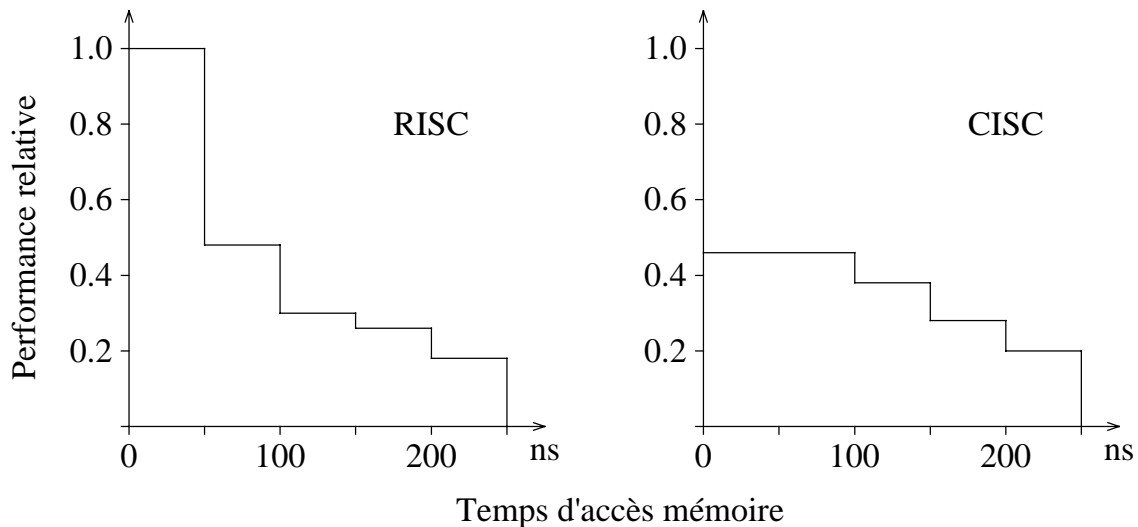


Figure 17

Il nous reste à décrire une organisation particulière des registres internes que l'on rencontre dans certaines architectures RISC. Elle est identifiée sous la dénomination de technique des fenêtres de registres, destinée à faciliter le passage de paramètres lors d'appel de procédures. Le nombre des registres est toujours important dans un processeur RISC. Il y avait par exemple 138 registres dans le prototype RISC-I de Berkeley avec lequel la technique a été développée. Celle-ci repose sur les statistiques déjà mentionnées sur le partage des variables et la profondeur d'imbrication des appels de procédures. Pour le RISC-I les dix premiers registres, accessibles à tous les niveaux, sont destinés à recevoir les variables globales. Les 128 registres suivants sont divisés en huit groupes ou fenêtres. Chaque groupe correspond à une profondeur d'appel. La profondeur maximale est donc limitée à huit. Les six premiers registres d'un bloc servent à l'échange des informations avec l'étage supérieur. Les dix autres sont disponibles pour les variables locales. Outre les registres globaux, chaque niveau intermédiaire dispose donc de vingt-deux registres : les seize qui lui sont propres et les six premiers registres de l'étage suivant (fig. 18). Cette technique simplifie les échanges d'informations entre les procédures appelantes et appelées. Elle demande cependant beaucoup de temps lors de la sauvegarde des registres en cas d'interruption.

Les architectures CISC et RISC, utilisant les mêmes technologies, ont de plus en plus tendance à se rapprocher. Certains processeurs, par exemple, traduisent les instructions CISC en suites d'instructions RISC avant l'exécution. Cela permet d'assurer la compatibilité des nouveaux

processeurs avec tous les logiciels développés antérieurement, ce qui est indispensable si on veut pouvoir vendre le nouveau processeur.

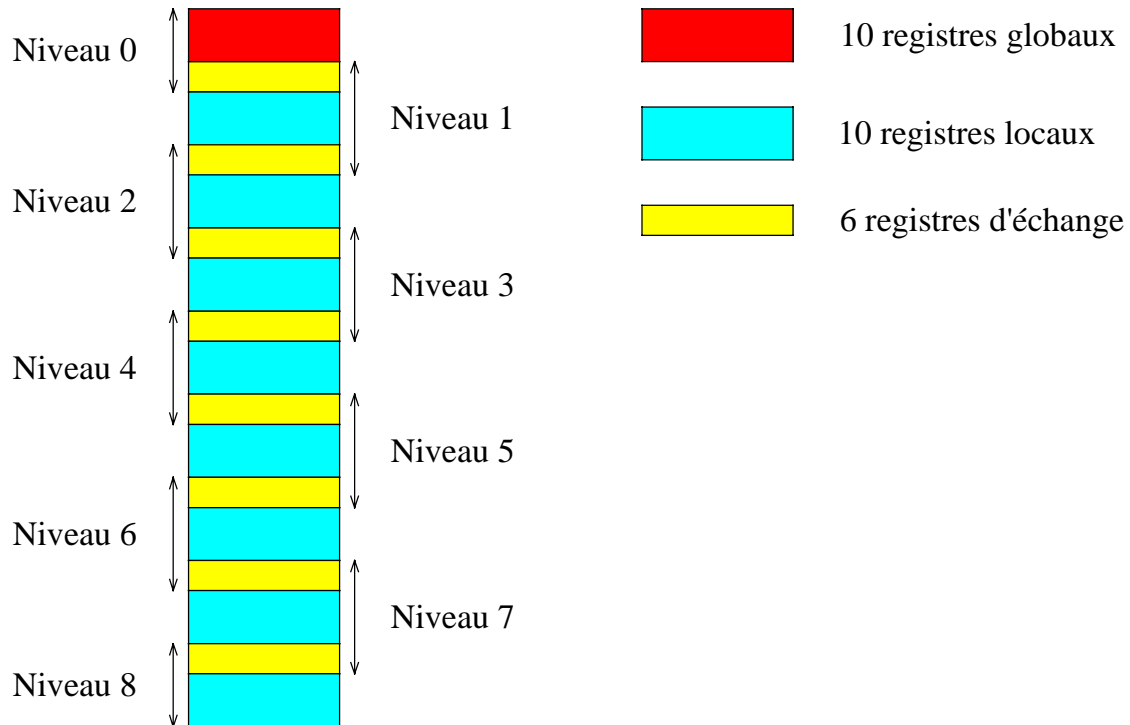


Figure 18

VII.12 Evaluation des performances

Il est très important, pour le concepteur comme pour le client, de pouvoir évaluer les performances d'une machine. Pour cela l'idéal serait de pouvoir faire appel à un programme de test (benchmark) synthétisant l'ensemble des performances d'une architecture. Un tel programme universel n'existe pas. Par contre il existe un grand nombre de programmes de tests. Chacun de ces tests permet d'évaluer certains types de caractéristiques. Les tests peuvent être classés en trois familles :

- Les tests élaborés par les concepteurs de systèmes.
- Les tests adaptés à des applications spécifiques : par exemple le CERN compare les performances des machines en utilisant des programmes représentatifs des applications habituelles en physique des particules : analyses de données et simulations. C'est très certainement la meilleure solution.
- Les tests publics.

Parmi ces derniers nous pouvons citer : Dhrystone, Linpack, Whestone, Doduc, etc. Par exemple Dhrystone est destiné à mesurer l'efficacité des processeurs et compilateurs pour des programmes écrits en langage C. Pour cette raison il fait souvent référence dans le monde UNIX. Il est basé sur une étude statistique des instructions relevées dans un millier de programmes réels. Il ne fait pas intervenir les opérations en virgule flottante, les opérations d'entrées/sorties et les appels aux fonctions systèmes. Le test consiste à exécuter une boucle de programme un nombre fixe de fois. La performance est exprimée en Dhrystone/seconde : le nombre moyen de boucles exécutées par seconde. Le test de Linpack, écrit en FORTRAN, est surtout destiné au domaine scientifique et en particulier aux calculateurs vectoriels. Il est basé sur le temps nécessaire à la résolution d'un système linéaire de 100 (300 ou même 1000) équations à 100 (300 ou 1000) inconnues. Les résultats sont exprimés en Mflops : millions d'opérations flottantes par seconde. On parle maintenant en Gflops et Tflops. Whestone est destiné à l'évaluation des performances arithmétiques et en particulier celles des co-processeurs.

L'unité la plus fréquemment rencontrée dans la littérature publicitaire est le MIPS : million d'instructions par seconde. Il faut cependant être très prudent. Par exemple les fonctionnalités des instructions sont loin d'être comparables d'une machine à l'autre : plusieurs instructions pour un processeur peuvent être nécessaires pour obtenir le même résultat qu'une instruction d'un jeu plus complexe. D'autre part, lorsque le temps d'exécution n'est pas le même pour toutes les instructions il est indispensable de savoir quels types d'instructions ont été utilisés pour effectuer l'évaluation.

D'autre part pour que les comparaisons puissent être significatives il faut assurer une standardisation des programmes d'évaluation. C'est le but par exemple d'une commission regroupant quatorze constructeurs créée en 1988 : la SPEC (System Performance Evaluation Cooperative). La SPEC a initialement retenu 10 programmes principalement écrits en C et en FORTRAN. Le temps d'exécution de chacun de ces programmes est comparé à un temps de référence : le temps d'exécution sur Vax 11/780. Le rapport entre cette référence et le temps d'exécution sur le système testé constitue un SPECratio. La moyenne pour les dix programmes fournit la mesure finale exprimée en SPECmark. Depuis 1992 on distingue performances pour les calculs en entiers ou en flottants : SPECInt92 et SEPECfp92.

