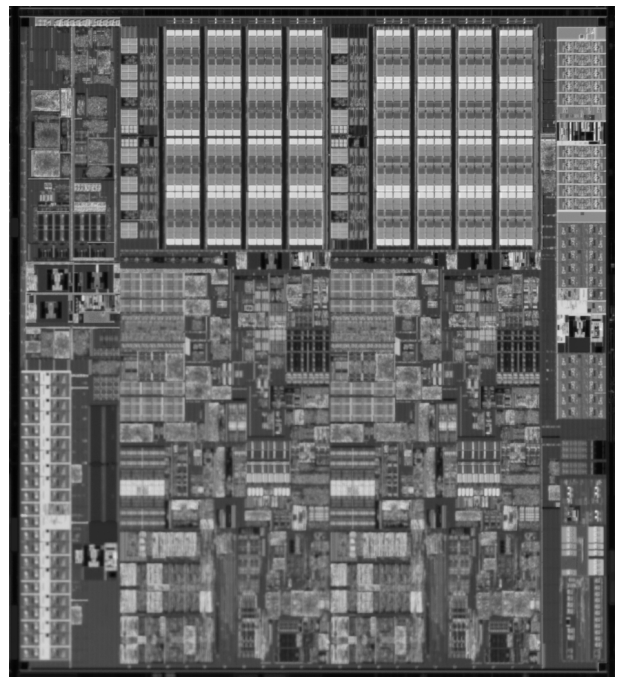


Architecture de l'ordinateur



Computers are my forte!
BRAZIL (Terry Gilliam, 1985)

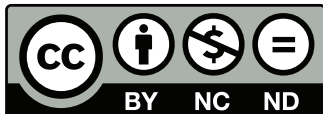
Ce document a initialement été publié sous forme de livre :

Emmanuel Lazard
Architecture de l'ordinateur
Collection Synthex
Pearson Education France, 2006
ISBN : 2-7440-7176-5

Ce livre étant épuisé et l'éditeur n'envisageant pas de réédition, ce dernier a donné son accord pour que le texte du livre soit en libre diffusion. Ce document lui est quasiment identique à quelques corrections orthographiques et mises à jour près.

Les polices de caractères utilisées dans ce document sont : Gentium, Inconsolata et Gill Sans.

Ce polycopié est diffusé sous Licence Creative Commons « Paternité - Pas d'Utilisation Commerciale - Pas de Modification 2.0 ».



Vous êtes libres de reproduire, distribuer et communiquer cette création au public selon les conditions suivantes.

Paternité — Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre).

Pas d'Utilisation Commerciale — Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.

Pas de Modification — Vous n'avez pas le droit de modifier, de transformer ou d'adapter cette création.

Les programmes figurant dans ce livre ont pour but d'illustrer les sujets traités. Il n'est donné aucune garantie quant à leur fonctionnement une fois compilés, assemblés ou interprétés dans le cadre d'une utilisation professionnelle ou commerciale.

La première image de couverture est une machine Z3 construite en 1941 par Konrad Zuse (Allemagne). Programmable, travaillant en binaire (et même en arithmétique flottante), elle est à base de relais électriques et est considérée comme ce qui s'approcherait le plus du « premier ordinateur ». Elle a été détruite lors d'un raid allié en 1943 mais une réplique fonctionnelle en a été faite dans les années 60 et se trouve au *Deutsches Museum* de Munich (Image courtesy of Computer History Museum).

La seconde photo représente l'architecture *Westmere* des derniers processeurs Intel avec environ un milliards de transistors sur une puce (Image courtesy of Intel Corporation).

Sommaire

Introduction	ii		
1. Représentation des nombres	1	6. Mémoire cache	135
1. Calcul binaire et entiers positifs	2	1. Principe	136
2. Nombres négatifs	6	2. Caractéristiques	138
3. Nombres réels	9	3. Amélioration des caches	145
4. Codage des caractères	11	Problèmes et exercices	149
5. Types et programmation	13	7. Mémoire virtuelle	161
Problèmes et exercices	15	1. Principe	162
2. Circuits logiques	21	2. Implémentation	164
1. Fonctions booléennes	22	3. Segmentation	172
2. Circuits combinatoires	30	Problèmes et exercices	175
3. Circuits logiques séquentiels	34	8. Entrées/sorties	185
Problèmes et exercices	40	1. Bus	186
3. Ordinateur et processeur	51	2. Interruptions	189
1. Architecture de von Neumann	52	3. Gestion des entrées/sorties	192
2. Les instructions	59	4. Technologies de stockage	196
3. UAL et registres	65	Problèmes et exercices	199
4. Séquenceur	68	Bibliographie	201
5. Architectures évoluées	70	Index	202
Problèmes et exercices	78		
4. Exemple de langage assembleur	89		
1. Description d'un processeur	90		
2. Instructions	91		
3. Programme d'assemblage	99		
4. Extensions possibles	102		
Problèmes et exercices	104		
5. Mémoire	115		
1. Caractéristiques	116		
2. Mémoire à semi-conducteurs	119		
3. Programmation et stockage en mémoire	123		
Problèmes et exercices	127		

Introduction

L'architecture de l'ordinateur est le domaine qui s'intéresse aux différents composants internes des machines, en explicitant leur construction et leurs interactions. Un ordinateur est un outil complexe qui peut effectuer des tâches variées, et dont les performances globales dépendent des spécifications de tous ses éléments. Comprendre son architecture permet de savoir dans quelle mesure les caractéristiques propres à chaque composant influencent la réactivité de la machine en fonction de son usage : pourquoi ajouter de la mémoire accélère-t-il l'ordinateur ? Pourquoi le temps d'accès d'un disque dur n'est-il qu'un des paramètres permettant de mesurer son efficacité ? Comment les processeurs font-ils pour aller toujours plus vite ?

L'utilité de ce domaine de connaissances est encore plus évidente pour les programmeurs qui développent des applications, de l'étudiant s'amusant sur son matériel personnel au professionnel écrivant des lignes de code au sein d'une équipe. La programmation dans des langages évolués est théoriquement indépendante des contraintes architecturales, mais dans les faits, celles-ci ont un impact sur l'écriture, la correction et les performances des programmes, *via*, par exemple, l'utilisation des pointeurs, le choix de la taille des variables ou de l'ordre des boucles. Négliger ou ignorer la structure de l'ordinateur amènera tôt ou tard à des déconvenues et à des erreurs.

Cet ouvrage se veut une présentation générale de l'architecture de l'ordinateur et de ses éléments, associant les descriptions techniques au logiciel, qu'il soit système d'exploitation ou application personnelle. Après avoir exposé les briques de base que sont la représentation des nombres et les circuits logiques, il entreprend de décortiquer la pièce maîtresse de l'ordinateur, à savoir le processeur, ainsi que son langage de programmation spécifique, en incluant une illustration des architectures avancées des processeurs actuels.

Mais un processeur isolé serait inutilisable sans l'ensemble des composants qui le soutiennent pour assurer le bon déroulement des programmes. Cet ouvrage décrit le système de stockage de l'information, depuis la mémoire cache jusqu'aux disques durs, en passant par la mémoire principale et la mémoire virtuelle. Pour finir, il présente le mécanisme des entrées/sorties, lié à la communication de l'ordinateur avec son environnement, aussi bien du point de vue matériel (contrôleur) que logiciel (pilote).

Cet ouvrage est inspiré d'un enseignement délivré depuis de nombreuses années à des étudiants en informatique dans des formations professionnelles. Il a pour ambition de faire comprendre les mécanismes internes de l'ordinateur et leurs implications sur le développement des logiciels aux élèves de licence scientifique, et de façon plus générale de faire découvrir l'architecture des machines à tous ceux qui s'intéressent à ce sujet. À la fin de chaque chapitre, des exercices corrigés permettent d'appliquer directement les notions présentées, à travers l'étude ou l'écriture de programmes, des applications numériques et des constructions de circuits.

L'auteur remercie chaleureusement les relecteurs, Daniel Chillet et Stéphane Nicolet, qui, par leurs conseils, remarques et documentation, ont largement amélioré les essais préliminaires. Merci à Karen, Kathy et Maude qui ont contribué, de mille et une manières, à l'écriture de ce livre.

LE PLAN

L'architecture des ordinateurs, les descriptions techniques et les interactions avec le logiciel sont exposées dans les huit chapitres de la façon suivante :

Chapitre 1 : Représentation des nombres.

Ce chapitre est centré sur la représentation informatique des nombres usuels et des caractères. Il montre comment ceux-ci sont stockés en mémoire (et l'impact sur le logiciel) et les limites de leur représentation (valeurs admises, valeurs maximales).

Chapitre 2 : Circuits logiques.

Ce chapitre présente les bases de la logique booléenne et son implémentation sous forme de circuits logiques, des premières portes logiques aux circuits élémentaires présents dans tous les circuits intégrés électroniques.

Chapitre 3 : L'ordinateur et le processeur.

Après une vue d'ensemble de l'ordinateur, ce chapitre propose la construction d'un processeur simple. Seront présentées ses différentes unités fonctionnelles et son langage de programmation. Il sera également question des avancées architecturales des processeurs récents.

Chapitre 4 : Un exemple de langage assembleur.

Ce chapitre est entièrement dévolu à la programmation en langage assembleur d'un processeur fictif. Les différentes instructions présentées montrent les possibilités du langage de commande du processeur. Elles seront mises en œuvre dans de nombreux exercices de programmation.

Chapitre 5 : La mémoire.

Ce chapitre débute par une présentation des différentes zones de stockage dans un ordinateur, en les distinguant suivant leurs caractéristiques (technologies, performances). Il s'intéresse ensuite à la mémoire principale à semi-conducteurs et à son utilisation en programmation, en explicitant la façon dont un logiciel gère son espace de mémorisation des variables.

Chapitre 6 : La mémoire cache.

La description des principes fondateurs de la mémoire cache est suivie par un inventaire exhaustif des caractéristiques de celle-ci (taille, organisation, remplacement, réécriture) et une illustration des techniques améliorant l'efficacité des caches (optimisation du code, caches multiniveaux...).

Chapitre 7 : La mémoire virtuelle.

La mémoire virtuelle est à mi-chemin entre le matériel et le système d'exploitation. Ce chapitre en présente les principes théoriques (pagination, segmentation) ainsi que les techniques d'implémentation en insistant sur les optimisations effectuées.

Chapitre 8 : Les entrées/sorties.

Le dernier chapitre se préoccupe des liens entre le processeur et son environnement, d'abord par l'étude succincte des bus de communication le reliant aux autres composants, puis par l'explication du fonctionnement des cartes d'entrées/sorties permettant de brancher des périphériques. Le côté logiciel comprend une description du mécanisme des interruptions, qui offre au processeur un moyen de réagir à un événement extérieur.

Chapitre 1

Représentation des nombres

Pour comprendre comment fonctionne un ordinateur, il faut d'abord comprendre comment il traite les nombres. La machine travaille uniquement avec deux valeurs (0 ou 1), appelées « bits », mémorisées dans des cases de taille limitée.

Comment représenter et stocker des nombres positifs, négatifs, décimaux ou même des caractères ? Voilà ce que vous allez découvrir dans ce chapitre.

Loin d'être enfouies dans les profondeurs du matériel, ces notions resurgissent au moment de la programmation dans le choix des types des variables.

Représentation des nombres

- 1. Calcul binaire et entiers positifs..... 2
- 2. Nombres négatifs 6
- 3. Nombres réels 9
- 4. Codage des caractères 11
- 5. Types et programmation 13

Problèmes et exercices

- 1. Écrire dans différentes bases 15
- 2. Représenter des entiers..... 15
- 3. Calculer en binaire 16
- 4. Débordement..... 16
- 5. Travailler avec des réels 17
- 6. Approximation de réels 18
- 7. Type des variables 19

I. CALCUL BINAIRE ET ENTIERS POSITIFS

Depuis toujours, les ordinateurs calculent en binaire. Il s'agit de la base 2 dans laquelle seuls les symboles 0 et 1 sont utilisés pour écrire les nombres, sans bien sûr que cela limite la taille des nombres représentables. Ceux-ci vont être stockés dans des cases qui, elles, ont une taille fixée. L'ordinateur est donc limité dans ses capacités à exprimer n'importe quel nombre.

I.1 Calcul dans une base quelconque

Notre système décimal et le système binaire ne sont que deux possibilités de représentation des nombres et nous pouvons généraliser cette écriture à une base B quelconque.

Symboles

De même qu'en décimal on utilise dix chiffres et que seuls deux symboles (0 et 1) sont permis en binaire, B symboles différents sont nécessaires pour exprimer un nombre quelconque en base B . Si B est inférieur ou égal à 10, on utilise évidemment les chiffres arabes classiques ; ainsi en base 8, on se sert des chiffres de 0 à 7. Si B est supérieur à 10, il faut définir de nouveaux symboles pour « compter » entre 10 et B . La seule base d'utilisation courante dans laquelle le problème se pose est la base 16 (voir section 1.2), qui utilise les lettres a, b, c, d, e et f pour les nombres de 10 à 15.

Valeur d'un nombre

Une fois les symboles choisis, on peut lire la représentation d'un nombre de n « chiffres » $x_{n-1}x_{n-2} \dots x_1x_0$ de la façon suivante :

$$X = x_{n-1}B^{n-1} + x_{n-2}B^{n-2} + \dots + x_1B + x_0 = \sum_{i=0}^{n-1} x_i B^i$$

Autrement dit, on multiplie chaque symbole avec les puissances successives de la base B .

Remarque

L'écriture d'un nombre est ambiguë si la base n'est pas explicite. Ainsi, la même écriture 101 peut valoir cent un si on lit le nombre en décimal, mais aussi $6^2 + 1$ en base 6, soit trente-sept, ou cinq en binaire ($2^2 + 1$). Pour éviter cela, lorsque la base n'est pas évidente suivant le contexte, il est courant d'indiquer le nombre par la base, ici 101_{10} , 101_6 ou 101_2 .

Valeurs maximales

Il est important de savoir quelle est la valeur maximale que l'on peut exprimer avec n symboles. Cela permet tout de suite de savoir si une case mémoire de l'ordinateur (qui a une taille fixée) sera assez grande pour contenir une valeur. Avec n symboles en base B , le plus grand nombre qu'il est possible de représenter s'écrit avec tous les symboles égaux à $B - 1$, ce qui donne la valeur $B^n - 1$:

$$\begin{aligned} & (B-1)B^{n-1} + (B-1)B^{n-2} + \dots + (B-1) \\ &= (B-1) \sum_{i=0}^{n-1} B^i = (B-1) \frac{B^n - 1}{B - 1} = B^n - 1 \end{aligned}$$

On peut donc écrire tous les nombres positifs de 0 à $B^n - 1$. Par exemple, en décimal, avec quatre chiffres, on peut compter de 0 à 9 999, tandis qu'en binaire, avec dix symboles binaires, on représente les nombres de 0 à $2^{10} - 1 = 1\,023$.

I.2 Bases classiques en informatique

De même que la communication humaine n'utilise que la base 10 (et les bases 24 et 60 pour le décompte du temps), l'informatique ne se sert que de quelques bases bien précises.

Base 2

La première base fondamentale est la base 2, qui permet le calcul binaire. Chaque symbole d'un nombre binaire peut prendre la valeur 0 ou 1 et s'appelle un bit (de l'anglais *Binary digiT*, chiffre binaire). Par exemple, le nombre 10110101_2 se calcule en décimal comme $2^7 + 2^5 + 2^4 + 2^2 + 1 = 181_{10}$.

Définition

Un nombre binaire composé de 8 bits s'appelle un **octet** et peut prendre des valeurs (décimales) de 0 à 255. Cette taille a son importance car c'est ce que peut contenir une case mémoire. Deux cases mémoire ensemble peuvent contenir 16 bits (permettant de stocker une valeur entre 0 et 65 535) et s'appellent parfois « mot binaire », un long mot étant alors quatre cases mémoire, soit 4 octets.

Complément

Le système de calcul binaire n'est évidemment pas naturel. Il s'est imposé parce qu'il est bien adapté aux contraintes des circuits électroniques. Il est en effet simple de distinguer deux valeurs de tension sur un fil. Si l'on avait souhaité reproduire directement le calcul décimal, dix valeurs différentes de la tension auraient été nécessaires, ce qui aurait rendu les circuits beaucoup plus complexes.

Bases 8 et 16

L'inconvénient majeur du calcul binaire est la place que prend l'écriture des nombres ! Pour exprimer un nombre N en binaire, il est nécessaire d'utiliser $\log_2 N$ symboles binaires (plus précisément la valeur entière supérieure de $\log_2 N$), alors que son écriture décimale nécessite autant de chiffres que son logarithme en base 10 ($\log_{10} N$). Le rapport entre ces deux valeurs indique que la représentation binaire d'un nombre emploie au moins trois fois plus de symboles que sa représentation décimale.

Deux autres bases usuelles, prenant moins de place, ont été exploitées dans l'histoire de l'informatique : d'abord la base 8 (octale), qui utilise les chiffres de 0 à 7, puis la base 16 (hexadécimale), qui utilise tous les chiffres décimaux ainsi que les lettres de a à f (puisque'il faut des symboles supplémentaires pour les nombres de 10 à 15). Par exemple, $2176_8 = 2 \times 8^3 + 8^2 + 7 \times 8 + 6 = 1150_{10} = 4 \times 16^2 + 7 \times 16 + 14 = 47e_{16}$.

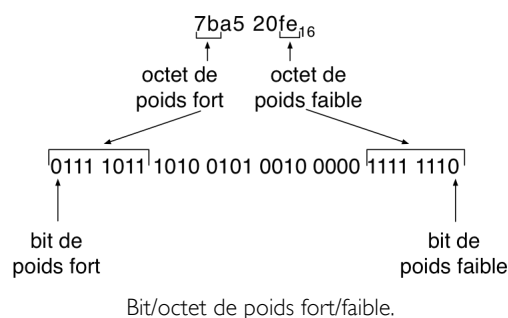
Complément

La base 8 n'est plus utilisée de nos jours alors que la base 16 l'est toujours (on verra pourquoi plus loin). Dans le langage de programmation C, on peut exprimer un nombre en octal en lui préfixant le symbole `\` et un nombre hexadécimal en lui préfixant `0x`.

Définition

Il est souvent utile de pouvoir désigner la partie la plus ou la moins importante d'un nombre (celle qui pèse le plus ou le moins dans le calcul de la valeur de ce nombre). Cela correspond aux symboles les plus à gauche ou les plus à droite dans l'écriture du nombre en question (quelle que soit la base). On parle alors du **bit de poids fort** et du **bit de poids faible** d'un nombre binaire, du chiffre de poids fort ou de poids faible d'un nombre décimal, du symbole hexadécimal de poids fort ou faible, etc. Par extension, on parle de l'**octet de poids fort** d'un nombre binaire écrit sur plusieurs octets, qui correspond aux 8 bits les plus à gauche dans l'écriture binaire de ce nombre, et de l'**octet de poids faible** (voir figure 1.1).

Figure 1.1



1.3 Changement de base

L'ordinateur ne sait calculer qu'en base 2. Malheureusement, l'écriture binaire n'est ni pratique (à cause de la taille des écritures), ni intuitive (le cerveau humain ne calcule facilement qu'en base 10). On doit donc souvent effectuer des changements de base entre la base 2 et les bases 8, 10 ou 16.

Le changement de base le plus simple est le passage entre la base 2 et les bases 8 ou 16. En effet, les valeurs 8 et 16 étant des puissances de 2 ($8 = 2^3$; $16 = 2^4$), chaque bloc de 3 ou 4 bits correspond à un symbole octal ou hexadécimal.

Correspondance base 2 – base 8 ou 16

Prenons un exemple. Pour écrire le nombre binaire 101101000110_2 en base 8, on exprime chaque groupe de 3 bits (en partant des bits de poids faible) dans son équivalent octal. On obtient ici successivement de droite à gauche $110 \rightarrow 6$, $000 \rightarrow 0$, $101 \rightarrow 5$, $101 \rightarrow 5$, ce qui donne : $101101000110_2 = 5506_8$. On opère de la même façon pour le passage vers la base 16, en regroupant les bits quatre à quatre : $(1011\ 0100\ 0110)_2 = b46_{16}$. Si le nombre de bits n'est pas divisible par 3 ou 4, il est alors nécessaire d'ajouter des bits nuls en tête du nombre. Par exemple : $1101010_2 = (001\ 101\ 010)_2 = 152_8$ en octal et $1101010_2 = (0110\ 1010)_2 = 6a_{16}$ en hexadécimal. À l'inverse, pour passer de la base 8 ou 16 à la base 2, on remplace chaque symbole octal ou hexadécimal par les 3 ou 4 bits équivalents : par exemple $327_8 = 11010111_2$ et $4ce_{16} = 10011001110_2$.

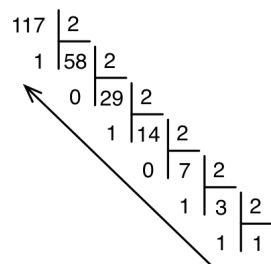
Remarque

La base 16 est bien adaptée pour exprimer les nombres en informatique car, comme l'unité de taille est l'octet, c'est-à-dire 8 bits, il suffit de deux symboles hexadécimaux pour écrire 1 octet. Pour cette raison, la base 8 n'est plus utilisée car il faut au moins trois symboles pour représenter 1 octet. En outre, cela permet (ce qui n'est pas souhaitable) d'écrire des nombres plus grands (sur 9 bits).

Correspondance base 2 – base 10

Le changement de base est ici plus compliqué car il n'y a pas de correspondance directe entre les bits et les chiffres décimaux. Il faut passer par une étape de calcul. Pour avoir la valeur décimale d'un nombre binaire, il suffit d'additionner les puissances de 2 correspondant aux bits mis à 1 dans l'écriture binaire : $101101000110_2 = 2^{11} + 2^9 + 2^8 + 2^6 + 2^2 + 2^1 = 2886_{10}$. Dans l'autre sens, il faut diviser le nombre décimal par 2, garder le reste, qui donne le bit de poids faible, et recommencer avec le quotient, jusqu'à arriver à un quotient égal à 1.

Figure 1.2



Conversion décimal vers binaire.

À la figure 1.2, on divise 117 par 2, ce qui donne un reste de 1 et un quotient de 58. On poursuit la division pour aboutir à un quotient de 1 (qui représente le bit de poids fort du nombre). On lit alors les bits en « remontant les divisions » : $117_{10} = 1110101_2$.

Complément

Le passage de la base 10 vers la base 2 n'étant pas automatique, certains ordinateurs ont historiquement utilisé le codage BCD (Binary Coded Decimal, décimal codé binaire). Dans ce codage, chaque chiffre décimal est directement transformé en 4 bits (3 bits n'étant pas suffisants). Ce n'est plus un codage binaire (les bits ne représentent plus les puissances de 2), mais une simple réécriture des chiffres. Cela permet une transformation facile mais complique la programmation des opérations arithmétiques.

1.4 Arithmétique binaire

Maintenant que nous savons compter en binaire, nous allons étudier les opérations arithmétiques classiques. Elles se réalisent de la même manière qu'en décimal.

Addition

De même que l'on additionne deux nombres décimaux chiffre à chiffre en obtenant un résultat et une retenue à reporter, pour additionner deux nombres binaires, il faut définir l'addition de 2 bits. Il n'y a que quatre cas à examiner :

- $0 + 0$ donne 0 de résultat et 0 de retenue.
- $0 + 1$ donne 1 de résultat et 0 de retenue.
- $1 + 0$ donne 1 de résultat et 0 de retenue.

- 1 + 1 donne 0 de résultat et 1 de retenue (de même qu'il y a une retenue lors de l'addition de deux chiffres décimaux quand le résultat est supérieur à 10).

On effectue ensuite l'addition binaire bit à bit, de droite à gauche, en reportant les retenues, comme dans l'exemple suivant :

$$\begin{array}{r}
 \begin{array}{ccccccc}
 & 1 & & 1 & & & \\
 & & & 1 & & 1 & \\
 & & & & 0 & 1 & \\
 & & & & & 0 & \\
 & & & & & & 1 & \\
 & & & & & & & 1
 \end{array} \\
 + \quad \begin{array}{ccccccc}
 1 & 1 & 0 & 1 & 1 & 1 & 0
 \end{array} \\
 \hline
 \begin{array}{ccccccc}
 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1
 \end{array}
 \end{array}$$

On peut vérifier le résultat en décimal : $43 + 110 = 153$.

Soustraction

Le scénario est identique pour la soustraction de deux nombres binaires : la soustraction de 2 bits donne un bit de résultat et un bit de retenue de report sur la colonne suivante :

- 0 – 0 donne 0 de résultat et 0 de retenue.
- 0 – 1 donne 1 de résultat et 1 de retenue.
- 1 – 0 donne 1 de résultat et 0 de retenue.
- 1 – 1 donne 0 de résultat et 0 de retenue.

On effectue la soustraction binaire bit à bit, de droite à gauche, en reportant les retenues, comme dans l'exemple suivant :

$$\begin{array}{r}
 \begin{array}{ccccccc}
 1 & 1 & 1 & 0 & 1 & 0 & 1
 \end{array} \\
 - \quad \begin{array}{ccccccc}
 & & 1 & 1 & 1 & 0 & 1 & 0
 \end{array} \\
 \hline
 \begin{array}{ccccccc}
 1 & 0 & 1 & 1 & 0 & 1 & 1
 \end{array}
 \end{array}$$

Nous parlerons du problème des nombres négatifs dans la prochaine section.

Multiplication

La multiplication binaire peut s'effectuer comme une suite d'additions successives des produits partiels, comme une multiplication décimale. Cela dit, elle est plus simple à poser que la multiplication décimale car les tables de multiplication sont réduites à leur plus simple expression ! On multiplie soit par 0 (et le résultat est nul) soit par 1 (et on recopie le multiplicateur). Voici un exemple :

$$\begin{array}{r}
 \begin{array}{ccccccc}
 & & & & 1 & 0 & 1 & 0 & 1 & 1 & 0
 \end{array} \\
 \times \quad \begin{array}{ccccccc}
 & & & & & & 1 & 1 & 0 & 1 & 0
 \end{array} \\
 \hline
 \begin{array}{ccccccccccc}
 & & & & & & & & 1 & 0 & 1 & 0 & 1 & 1 & 0 & \bullet
 \end{array} \\
 + \quad \begin{array}{ccccccccccc}
 & & & & 1 & 0 & 1 & 0 & 1 & 1 & 0 & \bullet & \bullet & \bullet
 \end{array} \\
 + \quad \begin{array}{ccccccccccc}
 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & \bullet & \bullet & \bullet & \bullet
 \end{array} \\
 \hline
 \begin{array}{ccccccccccc}
 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0
 \end{array}
 \end{array}$$

Cette manière d'opérer s'implémente facilement car elle consiste à ne faire que des décalages et des additions, opérations classiques sur les processeurs. L'inconvénient de la multiplication est qu'elle allonge fortement les nombres : multiplier deux nombres de n bits peut donner un résultat sur $2n$ bits. Il faut donc faire attention lorsque ces nombres sont stockés dans une case mémoire car le résultat, de taille double, peut ne plus tenir dans une case.

Division

La division binaire est l'opération la plus compliquée. On opère comme en décimal : on soustrait le diviseur du dividende en commençant par les bits de poids fort. Elle nécessite une série de soustractions et de décalages pour donner un quotient et un reste.

2. NOMBRES NÉGATIFS

Nous savons maintenant représenter en binaire des nombres entiers positifs. Mais comment faire pour travailler avec des nombres négatifs ? Dans l'arithmétique classique, ces nombres sont précédés d'un signe moins, mais c'est ici impossible car on ne peut mettre que des bits dans une case mémoire. Il faut donc trouver une écriture, une représentation des nombres négatifs, utilisant les bits disponibles.

2.1 Représentation « signe et valeur absolue »

La première idée est de reproduire le signe en utilisant le bit de poids fort (voir figure 1.1) du nombre pour le représenter.

Définition

Sur n bits, le bit de poids fort (auss appelé « bit de signe ») indique le signe du nombre (selon une convention classique, ce bit est à 1 pour un nombre négatif) et les $n - 1$ bits restants donnent la valeur absolue binaire du nombre. Les mots binaires $01011001 = 89$ et $10110100 = -52$ sont deux exemples de nombres écrits dans cette représentation sur 8 bits.

Caractéristiques

Notez que l'on ne représente pas plus de nombres ainsi : sur n bits, on représente toujours au maximum 2^n valeurs différentes. Ce que l'on a fait est un décalage de la plage des nombres autorisés en passant de l'intervalle $[0, 2^n - 1]$ à l'intervalle $[-(2^{n-1} - 1), 2^{n-1} - 1]$. Il n'y a plus que $2^n - 1$ nombres exprimés car le zéro peut s'écrire de deux manières différentes : $+0$, c'est-à-dire $00...0$, et -0 qui s'écrit $10...0$.

Arithmétique

Dans cette représentation, il est facile de savoir si un nombre est positif ou négatif puisqu'il suffit de regarder le bit de poids fort. Par ailleurs, changer le signe du nombre revient à inverser ce bit. En revanche, les opérations arithmétiques classiques sont plus compliquées qu'avec des nombres positifs car les règles d'addition des bits ne s'appliquent plus : cela reviendrait à additionner les valeurs absolues, et non les nombres eux-mêmes ! Avant d'additionner deux nombres, il faut regarder s'ils sont de signes opposés et, dans ce cas, procéder à une soustraction des valeurs absolues puis retrouver le signe du résultat en fonction de la taille respective des opérandes. Ces contraintes ralentissent les opérations arithmétiques et, pour cette raison, cette représentation n'est plus utilisée.

2.2 Représentation en complément à 2

C'est la représentation standard sur les ordinateurs pour exprimer les nombres entiers négatifs. Quand on parle de représentation signée ou d'entiers signés, ces derniers sont toujours exprimés à l'aide de la représentation en complément à 2.

Définition

Sur n bits, on exprime les nombres de l'intervalle $[-2^{n-1}, 2^{n-1} - 1]$. On retrouve bien les 2^n nombres possibles. Un nombre positif est représenté de façon standard par son écriture binaire. On représente un nombre négatif en ajoutant 1 à son complément à 1 (obtenu en inversant tous les bits) et en laissant tomber une éventuelle retenue finale.

Si l'on reprend l'exemple précédent, 89 (positif) s'écrit toujours 01011001, mais -52 (négatif) s'écrit maintenant 11001100. En effet, en binaire sur 8 bits, 52 s'écrit 00110100, ce qui donne un complément à 1 égal à 11001011 et un complément à 2 égal à 11001100.

Dans le cas d'un nombre négatif, il est important d'indiquer sur combien de bits doit s'écrire le nombre car on ne rajoute pas des zéros en tête mais des uns (suite à l'inversion obtenue lors du calcul du complément à 1). La phrase « représentez -20 en complément à 2 » n'a pas de sens si l'on ne précise pas le nombre de bits de la représentation. Cela peut tout aussi bien être 101100 sur 6 bits que 1101100 sur 8 bits ou 11111111101100 sur 16 bits (voir section 2.3).

Caractéristiques

Les nombres positifs se représentent tels quels, c'est-à-dire par une écriture binaire sur n bits : de $0 = 000...00$ à $2^{n-1} - 1 = 011...11$. Les entiers négatifs correspondent au complément à 2 de ces nombres : de $-2^{n-1} = 100...00$ à $-1 = 111...11$.

1 = 111...11. Il n'y a plus maintenant qu'une seule représentation de 0 ; d'ailleurs, si l'on essaie de prendre l'opposé de 0 = 000...00 en faisant son complément à 2, on retombe sur la même écriture binaire.

Notez que l'on peut représenter plus de nombres strictement négatifs que de nombres strictement positifs, en l'occurrence un de plus. C'est parfaitement normal : avec 2^n possibilités sur n bits, si l'on enlève le zéro, il reste un nombre impair d'écritures possibles ; on ne peut donc pas avoir autant de nombres positifs que de nombres négatifs. On aurait pu décider, pour le nombre binaire 100...00, de représenter 2^{n-1} au lieu de -2^{n-1} , mais la convention choisie permet de considérer le bit de poids fort d'un nombre comme un équivalent de bit de signe : tous les nombres négatifs (de -2^{n-1} à -1) ont ce bit de poids fort à 1 et tous les nombres positifs (de 0 à $2^{n-1} - 1$) ont ce bit à 0. Cela permet de déterminer facilement le signe d'un nombre en représentation signée.

Arithmétique en complément à 2

Le grand intérêt du complément à 2 est de simplifier les additions. Ainsi, on n'a plus à se préoccuper du signe des nombres avant d'effectuer l'opération. Voici un exemple. Pour effectuer l'addition de 118 et de -36, il suffit d'exprimer chacun des nombres en complément à 2 et d'effectuer l'addition bit à bit en laissant tomber une éventuelle retenue :

$$\begin{array}{r}
 118 \quad : \quad \overset{1}{0} \quad \overset{1}{1} \quad \overset{1}{1} \quad \overset{1}{1} \quad \overset{1}{0} \quad 1 \quad 1 \quad 0 \\
 +(-36) \quad : \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 \\
 \hline
 =82 \quad \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 0
 \end{array}$$

Soustraire un nombre revient à additionner son opposé, c'est-à-dire son complément à 2.

En revanche, la multiplication est plus délicate. La méthode la plus simple est de prendre la valeur absolue des deux nombres, de les multiplier par l'algorithme standard et de prendre l'opposé du résultat si les deux nombres étaient de signes contraires. Il existe un autre algorithme, appelé « recodage de Booth », qui profite des spécificités de la multiplication binaire pour accélérer le calcul au prix d'une circuiterie plus compliquée.

Pour diviser, il faut passer par la division des valeurs absolues et repositionner les signes du quotient et du reste si nécessaire.

Note

La représentation en complément à 2 revient à lire un nombre binaire $x_{n-1} \dots x_0$ de la façon suivante :

$$X = x_{n-2} 2^{n-2} + \dots + 2x_1 + x_0 - x_{n-1} 2^{n-1} = \sum_{i=0}^{n-2} x_i 2^i - x_{n-1} 2^{n-1}$$

Propriétés du complément à 2

On a la relation suivante, sur n bits : $X + C_2(X) = 2^n$ (d'où son nom). Cela permet de calculer facilement le complément à 2 d'un nombre en effectuant l'opération $C_2(X) = 2^n - X$. On a une autre propriété intéressante : si l'on prend deux fois le complément à 2 d'un nombre, on retombe sur ce nombre : $C_2(C_2(X)) = X$. Ainsi, pour calculer la valeur décimale d'un nombre binaire négatif, il suffit de prendre son complément à 2 et l'on a son opposé : $C_2(11001100) = 00110100 = 52$ donc $11001100 = -52$.

Par ailleurs, on a une relation intéressante entre les lectures signée et non signée d'un nombre binaire négatif sur n bits. Soit $VS(X)$ la valeur d'un nombre binaire, lu comme un entier négatif dans la représentation en complément à 2 ; soit $VB(X)$ la valeur de ce même nombre binaire, mais lu comme un entier positif (c'est-à-dire en effectuant la simple somme des puissances de 2). On a alors l'égalité $VB(X) - VS(X) = 2^n$. Par exemple, $VB(10101010) = 170$, $VS(10101010) = -86$ et l'on a bien $170 - (-86) = 256 = 2^8$.

2.3 Extension de signe

Parfois, on a besoin d'étendre un nombre de n bits sur davantage de bits. Par exemple, pour additionner un nombre écrit sur 1 octet et un nombre écrit sur 2 octets, il faut étendre le premier nombre sur 16 bits avant d'effectuer l'addition. Un autre exemple est le chargement d'une case mémoire de 1 octet dans un registre (une zone de stockage à l'intérieur du microprocesseur) de 4 octets (32 bits). Comment faire pour que le nombre étendu représente bien la même valeur ? On doit effectuer ce que l'on appelle une extension de signe.

Nombres non signés

Si le nombre n'est pas signé, il suffit d'ajouter des zéros en tête du nombre à étendre : par exemple 11010011 devient 0000000011010011 sur 2 octets.

Représentation « signe et valeur absolue »

Si l'on travaille avec des nombres signés représentés avec la convention « signe et valeur absolue », il suffit de reporter le bit de signe dans le nouveau bit de poids fort et de remplir les autres nouveaux bits par des zéros. 01101000 devient 000000001101000 et 11111011 devient 100000001111011 lorsqu'on les étend sur 2 octets.

Représentation en complément à 2

Pour les nombres signés standard, la règle est de recopier le bit de signe du nombre dans tous les nouveaux bits. En effet, si le nombre est positif, son bit de signe est nul et on ajoute des zéros en tête : 00100110 devient 0000000000100110 ; si le nombre est négatif, le bit de signe (c'est-à-dire le bit de poids fort) est égal à un et il faut le reporter : 10001111 devient 1111111110001111. Dans tous les cas, lorsque l'on calcule la valeur des nombres en complément à 2, on obtient bien le même résultat.

2.4 Débordement

Lorsqu'ils sont stockés en mémoire dans un ordinateur, les nombres binaires ont une taille limitée : souvent 1, 2 ou 4 octets. Parfois, le résultat d'une opération arithmétique, par exemple d'une addition, ne peut pas tenir dans la taille imposée. On dit alors qu'il y a débordement (*overflow*). Comment le détecter ? Cela dépend bien évidemment de la représentation des nombres.

Nombres non signés

Sur n bits, on représente les nombres de 0 à $2^n - 1$. Si la somme de deux nombres dépasse cette valeur, il y a débordement. On peut facilement s'en apercevoir car cela est équivalent à la présence d'une retenue finale lors de l'addition. Voici un exemple sur 8 bits :

$$\begin{array}{r} 156 : 1 \overset{1}{0} \overset{1}{0} \overset{1}{1} 1 1 0 0 \\ + 168 : 1 0 1 0 1 0 0 0 \\ \hline 68? \overset{1}{0} 1 0 0 0 1 0 0 \end{array}$$

Le résultat de $156 + 168$ dépasse 255 et ne peut donc pas s'exprimer sur 8 bits ; cela est matérialisé par la retenue finale.

De la même façon, une soustraction peut amener à un résultat négatif, non représentable avec des nombres non signés. Là encore, une retenue finale indique un débordement.

$$\begin{array}{r} 27 : 0 0 0 1 1 0 1 1 \\ - 100 : \overset{1}{0} \overset{1}{1} 1 0 \overset{1}{0} 1 0 0 \\ \hline 183? \overset{1}{1} 1 0 1 1 0 1 1 \end{array}$$

Dans les deux cas, le résultat obtenu n'est pas le résultat correct : il y a débordement. Il faut pouvoir le détecter pour, par exemple, arrêter le calcul.

Nombre signés en complément à 2

À cause de la présence des nombres négatifs, une opération peut générer une retenue finale sans qu'il y ait débordement. Ainsi, l'opération $-60 + (-61)$ donne le résultat attendu :

$$\begin{array}{r} -60 : 1 \overset{1}{1} 0 0 0 1 0 0 \\ + (-61) : 1 1 0 0 0 0 1 1 \\ \hline = -121 \overset{1}{1} 0 0 0 0 1 1 1 \end{array}$$

Il y a bien retenue finale mais le résultat exprimé sur 8 bits est correct.

Inversement, l'absence de retenue finale ne garantit pas une opération correcte :

$$\begin{array}{r} 77 : 0 \overset{1}{1} 0 \overset{1}{0} \overset{1}{1} 1 0 1 \\ 68 : 0 1 0 0 0 1 0 0 \\ \hline -111? 1 0 0 1 0 0 0 1 \end{array}$$

Il faut en fait comparer les signes des nombres additionnés et le signe du résultat. Les signes des nombres additionnés correspondant aux bits de poids fort, leur addition génère la retenue finale ; le signe du résultat est, quant à lui, généré par la dernière retenue (celle qui s'additionne aux bits de poids fort justement). La règle est simple :

- Si les deux retenues générées par l'addition des deux derniers bits de chaque nombre (ceux de poids fort) sont identiques (11 ou 00), il n'y a pas débordement.
- Si les deux retenues sont différentes (01 ou 10), il y a débordement.

Dans le premier exemple, les deux dernières retenues valent 1 ; il n'y a donc pas débordement (le résultat exprimé sur 8 bits est correct). Dans le second, elles valent 0 et 1, ce qui est la preuve d'un débordement (effectivement, -111 n'est pas le résultat correct de la somme de 77 et 68).

3. NOMBRES RÉELS

Dans de nombreuses applications, on ne peut se contenter de calculer avec les nombres entiers, soit parce que l'on a besoin de nombres sortant de l'intervalle de représentation, soit parce que l'on utilise des nombres décimaux. Comme un ordinateur ne sait manipuler que des bits, il faut définir une représentation des nombres décimaux adaptée aux calculs. L'espace de stockage d'un nombre (son nombre de bits) étant limité, tous les nombres réels possibles ne sont pas représentables et il y a une limite à la précision des calculs.

L'idée principale derrière la représentation des nombres décimaux est de définir l'emplacement d'une virgule séparant la partie entière et la partie décimale et de considérer que les bits à droite de cette virgule correspondent aux puissances négatives de 2 (de même qu'en décimal, les chiffres à droite de la virgule sont des dixièmes, des centièmes...).

3.1 Représentation en virgule fixe

Historiquement la première, la représentation en virgule fixe est facile à comprendre. Au lieu de faire commencer les puissances de 2 à 2^0 au bit de poids faible, comme pour les nombres entiers, on fixe un bit, quelque part dans l'écriture du nombre, à droite duquel se place la virgule. Les bits à gauche de la virgule correspondent alors aux puissances de 2 positives et les bits à droite de la virgule correspondent aux puissances de 2 négatives.

Ainsi, sur n bits, on peut par exemple définir une virgule après les p bits de poids faible, ce qui donne, pour le nombre $x_{n-p-1}x_{n-p-2} \dots x_1x_0x_{-1} \dots x_{-p}$, la valeur :

$$X = x_{n-p-1}2^{n-p-1} + x_{n-p-2}2^{n-p-2} + \dots + x_12 + x_0 + x_{-1}2^{-1} + \dots + x_{-p}2^{-p} = \sum_{i=-p}^{n-p-1} x_i 2^i$$

Le problème évident est que la plage des nombres représentables est assez limitée. On est entre 2^{-p} et $2^{n-p} - 2^{-p}$, avec peu de précision après la virgule. Pour y remédier, on a développé une autre représentation, appelée « représentation en virgule flottante ».

3.2 Représentation en virgule flottante IEEE 754

Il s'agit maintenant d'imiter ce que l'on appelle la notation scientifique. On va écrire le nombre voulu sous la forme $\pm 1, M \times 2^E$, où $1, M$ s'appelle la mantisse du nombre et E l'exposant. Comme la mantisse commence toujours par une partie entière égale à 1, on ne l'écrit pas et on n'exprime que la partie fractionnaire, M , appelée « pseudo-mantisse ».

Puisqu'il y a plusieurs manières d'écrire les différents champs, une normalisation a été adoptée qui définit deux principaux types de nombres en virgule flottante : la simple précision sur 32 bits et la double précision sur 64 bits. Les nombres en simple précision possèdent une pseudo-mantisse sur 23 bits (correspondant aux puissances négatives de 2, de 2^{-1} à 2^{-23}), un exposant sur 8 bits et un bit de signe. Les nombres en double précision ont une pseudo-mantisse sur 52 bits, un exposant sur 11 bits et un bit de signe.

1 bit	8 (ou 11) bits	23 (ou 52) bits
signe	exposant	pseudo-mantisse

Lire et écrire un nombre en virgule flottante

La pseudo-mantisse est la partie fractionnaire du nombre et il suffit d'additionner les puissances négatives de 2 pour calculer le résultat. Afin que l'on puisse exprimer les exposants négatifs, la valeur binaire dans le champ exposant est la valeur réelle décalée par excès d'un biais de 127 (1 023 pour les nombres en double précision). Il faut donc retrancher ce biais de la valeur indiquée pour obtenir l'exposant réel. Le bit de poids fort est un bit de signe, par convention à 1 quand le nombre est négatif.

Voici trois exemples :

$$500 = 1,953125 \times 2^8 = (1 + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-6}) \times 2^8$$

Cela donne 0 1000111 1111010000000000000000. On a un bit de signe égal à 0, un exposant égal à 8 + 127 et les premiers bits de la pseudo-mantisse qui exprime 0,953125.

$$-3,5 = -1,75 \times 2^1 = -(1 + 2^{-1} + 2^{-2}) \times 2^1$$

Cela donne 1 1000000 1100000000000000000000. On a un bit de signe égal à 1 car le nombre est négatif, un exposant égal à 1 + 127 et les premiers bits de la pseudo-mantisse qui exprime 0,75.

$$0,40625 = 1,625 \times 2^{-2} = (1 + 2^{-1} + 2^{-3}) \times 2^{-2}$$

Cela donne 0 0111101 1010000000000000000000. On a un bit de signe égal à 0, un exposant égal à -2 + 127 et les premiers bits de la pseudo-mantisse qui exprime 0,625.

Remarque

Par analogie avec les nombres entiers, on désigne souvent les « nombres représentés en virgule flottante » comme des « nombres flottants ».

Valeurs spéciales

La plage de représentation est à peu près de 10^{-38} à 10^{38} en simple précision et de 10^{-308} à 10^{308} en double précision. Certaines configurations de bits signifient des valeurs spéciales :

Tableau I.1

Exposant	Pseudo-mantisse	Valeur
0	0	± 0
0	différent de 0	nombre dénormalisé
entre 1 et 254	quelconque	nombre normalisé standard
255	0	$\pm \text{infini}$
255	différent de 0	NaN (<i>Not a Number</i>)

Format des nombres flottants dans la norme IEEE 754

Les valeurs de plus ou moins l'infini peuvent apparaître dans les calculs, et les règles arithmétiques correspondantes sont précisées dans la norme. Les nombres dénormalisés permettent d'exprimer un nombre encore plus petit que 10^{-38} en écrivant la mantisse 0,M plutôt que 1,M. Enfin les NaN signalent une erreur (division par zéro par exemple).

Arrondis

En raison du nombre limité de bits de la représentation, les calculs en virgule flottante ne peuvent pas atteindre une précision infinie. Tous les nombres réels ne sont pas représentables et l'un des enjeux de la normalisation a été que les calculs soient reproductibles d'une machine à une autre. Or, en raison de la précision limitée, ceux-ci ne peuvent pas être systématiquement exacts. On a donc défini des mécanismes standard d'arrondi pour être sûr de toujours parvenir au même résultat. Ces mécanismes d'arrondi sont au nombre de quatre :

- arrondi à la valeur la plus proche (mécanisme par défaut) ;
- arrondi vers zéro ;
- arrondi vers plus l'infini ;
- arrondi vers moins l'infini.

Arithmétique en virgule flottante

Les calculs sont ici beaucoup plus compliqués qu'en arithmétique entière. Du fait que les bits des nombres ont des significations différentes suivant leur place, il faut traiter séparément les différents champs. Voici à titre d'exemple les algorithmes pour la multiplication et l'addition.

Multiplication de deux nombres flottants

1. Multiplier les mantisses.
2. Additionner les exposants en soustrayant une fois le biais pour retomber sur un exposant biaisé.
3. Si nécessaire, renormaliser la mantisse sous la forme $1,M$ et décaler l'exposant.
4. Arrondir la mantisse (car la multiplication a généré plus de bits significatifs).
5. Ajuster le signe du résultat en fonction des signes des nombres à multiplier.

Par exemple, multiplions 1,75 par 2,5.

$1,75 = 1,75 \times 2^0$, qui se représente ainsi : 0 01111111 1100000000000000000000.

$2,5 = 1,25 \times 2^1$, qui se représente ainsi : 0 10000000 010000000000000000000000.

La multiplication des mantisses (il faut multiplier $1,11_2$ par $1,01_2$, et pas uniquement les pseudo-mantisses) donne $10,0011_2$ et l'addition des exposants aboutit à un exposant biaisé de 128. On décale la mantisse vers la droite (en ajoutant 1 à l'exposant) pour renormaliser le nombre en $1,00011_2 \times 2^2$, qui s'écrit alors 0 10000001 000110000000000000000000. Cela donne bien $(1 + 2^{-4} + 2^{-5}) \times 2^2 = 4,375$.

La division flottante s'effectue de façon semblable : on divise les mantisses et on soustrait les exposants.

Addition de deux nombres flottants

L'addition de deux nombres flottants ne peut se faire que si les exposants sont égaux. L'algorithme est le suivant :

1. Décaler la mantisse d'un des nombres pour aligner les exposants.
2. Additionner les mantisses (attention au signe).
3. Si nécessaire, renormaliser la mantisse sous la forme $1, M$ et décaler l'exposant.
4. Arrondir la mantisse (car, à cause du décalage initial, la mantisse peut être sur plus de bits que la taille autorisée).

Par exemple, additionnons 1,75 et 2,5.

Comme précédemment, 1,75 se représente ainsi : $1,11_2 \times 2^0$, et 2,5 de la façon suivante : $1,01_2 \times 2^1$. On décale le premier nombre en l'écrivant $0,111_2 \times 2^1$ pour aligner les exposants. L'addition des deux mantisses donne $1,01_2 + 0,111_2 = 10,001_2$. On normalise en décalant vers la droite et en ajoutant 1 à l'exposant, ce qui amène à un résultat égal à $1,0001_2 \times 2^2$, soit 4,25.

On réalise la soustraction de façon identique, en établissant la différence des mantisses.

4. CODAGE DES CARACTÈRES

Les caractères, comme les nombres, doivent être représentés en vue d'être exploités par les ordinateurs. Comme on ne peut mettre que des bits dans les cases mémoire, on a associé un code numérique à chaque caractère. Évidemment, pour pouvoir échanger des informations entre ordinateurs, il faut que cette correspondance soit la même sur toutes les machines.

4.1 Code ASCII

La première tentative de standardisation date de la fin des années 60, avec la création du code ASCII (*American Standard Code for Information Interchange*). Cette table définit les équivalents numériques des caractères majuscules et minuscules de l'alphabet latin, des chiffres et de certains signes de ponctuation. Elle donne également le code de nombreux caractères de contrôle, utilisés à l'époque dans l'échange de données entre terminaux et ordinateurs ; non imprimables, ils servaient à régler les paramètres de la communication. Compte tenu du nombre de caractères à représenter, un code à 7 bits est nécessaire, permettant cent vingt-huit combinaisons différentes (voir tableau 1.2).

Tableau 1.2

	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	\$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(8	H	X	h	x
9	HT	EM)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[k	{
C	FF	FS	,	<	L	\	l	
D	CR	GS	-	=	M]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Table des caractères ASCII

Chaque caractère est codé sur 1 octet par deux symboles hexadécimaux. Le numéro de la colonne donne le symbole hexadécimal de poids fort et le numéro de ligne le symbole de poids faible. Ainsi, le caractère 4 a pour code numérique 34_{16} , L le code $4C_{16}$ et m le code $6d_{16}$.

Ce code est adapté à l'écriture anglo-saxonne : ni lettres accentuées ni lettres spéciales hors des vingt-six classiques. L'unité de stockage étant l'octet, soit 8 bits, les fabricants d'ordinateurs ont décidé d'utiliser le bit supplémentaire pour définir cent vingt-huit autres caractères, comme les lettres accentuées, les lettres d'autres alphabets ou même des caractères graphiques à afficher. Évidemment, chaque fabricant a construit sa propre table étendue et l'on a perdu la compatibilité entre ordinateurs, d'où les problèmes de transfert de fichiers au format texte contenant des caractères accentués. Cela a donné lieu à autre tentative de normalisation qui a aboutit à la norme ISO 8859 codifiant plusieurs alphabets d'Europe occidentale. La norme définit une quinzaine de tables différentes, chacune codant ses caractères sur 1 octet. Mais là encore, pour des raisons historiques de compatibilité, on emploie d'autres tables de correspondance, ce qui rend les transferts d'informations parfois problématiques.

4.2 Code Unicode

À la fin des années 80 démarre un projet visant à codifier de manière unique tous les caractères de toutes les langues écrites, en tenant compte du sens de lecture des langues (de gauche à droite ou l'inverse), des ligatures, etc. Ce travail aboutit au codage Unicode dans lequel chaque symbole est défini sur 2 octets. De portée mondiale, ce codage n'est pas encore universellement appliqué car de nombreux systèmes et applications ne savent pas encore traiter ces caractères étendus. Le langage de programmation C, ancien mais très utilisé, code ses caractères sur 1 octet alors que Java les code sur 2 octets et est donc compatible avec Unicode. Notez que dans un souci d'harmonisation, les deux cent cinquante-cinq premiers caractères Unicode reproduisent les deux cent cinquante-cinq caractères du codage ISO 8859-1 de l'alphabet latin.

5. TYPES ET PROGRAMMATION

Chaque langage de programmation décrit des types de variables et une taille de stockage associée. Lorsque l'on programme, il est important de garder à l'esprit les tailles des variables utilisées et les valeurs minimale et maximale qu'elles peuvent prendre, afin d'éviter de leur attribuer une valeur numérique trop importante suite à un calcul ou une entrée extérieure. Dans le meilleur des cas, si cette erreur est prévue par le langage et que des sécurités sont installées par le compilateur, le programme s'arrête de lui-même en signalant une erreur. Mais le plus souvent, il continue comme si de rien n'était, en tronquant la valeur numérique fautive pour la coder dans la variable ; la valeur est alors faussée, entraînant des calculs incorrects.

5.1 Types usuels et tailles

Voici un résumé des principaux types de variables entières en C et Java avec leurs valeurs extrêmes les plus classiques.

Tableau 1.3

Type C	Type Java	Taille	Valeur min.	Valeur max.
char	byte	1 octet	-128	127
unsigned char	n'existe pas	1 octet	0	255
short	short	2 octets	-32 768	32 767
unsigned short	char	2 octets	0	65 535
long	int	4 octets	env. -2 milliards	env. 2 milliards
unsigned long	n'existe pas	4 octets	0	env. 4 milliards
n'existe pas	long	8 octets	env. -10^{19}	env. 10^{19}

Types de variables entières en C et Java

La norme du langage C n'impose pas ces valeurs et laisse une certaine latitude lors de l'implémentation. Un compilateur peut décider que, par défaut, le type char peut être non signé ou bien que le type long peut être codé sur 64 bits. Nous n'avons pas inclus le type C int dans le tableau. Là encore, il n'est pas précisé dans la norme si les variables de ce type doivent être stockées sur 2 ou 4 octets ; cela dépend du système d'exploitation et du compilateur utilisé. Sur les ordinateurs actuels, le comportement par défaut est de les stocker sur 4 octets mais cela peut amener à des erreurs d'exécution avec d'anciens programmes susceptibles d'être recompilés.

5.2 Programmation

Chaque variable est typée et cette information (le type) est uniquement utilisée par le compilateur. Lorsqu'une valeur numérique est mise en mémoire, il n'y a aucune information sur son type. En d'autres termes, c'est le contexte d'exécution défini par le compilateur qui indique au processeur comment traiter la variable. Par conséquent, si le programmeur demande au compilateur de faire n'importe quoi, celui-ci indiquera au processeur de faire n'importe quoi ! Bien sûr, les langages de programmation sont plus ou moins permissifs et laissent le programmeur faire plus ou moins de bêtises. Mais il ne faut jamais supposer que le processeur connaît le type des variables sur lequel il travaille. C'est au programmeur de faire attention à ce sujet (taille limite, affichage...).

Considérez le programme C suivant :

Listing 1.1

```
int main () {
    float ff = 1.0;
    long ll = 999999999;

    printf( " Valeur de ff : %ld \n", ff);
    printf( " Valeur de ll : %e \n", ll);
}
```

On affiche une valeur, mais laquelle ?

Voici l'affichage après exécution :

```
Valeur de ff : 1072693248
Valeur de ll : 1.418200e-21
```

Remarque

Les valeurs affichées suite à l'exécution du code précédent dépendent de la machine sur laquelle tourne le programme et de la façon dont les nombres sont stockés en mémoire. Mais dans tous les cas, elles seront fantaisistes. Reportez-vous au chapitre 5 pour plus d'explications.

Que se passe-t-il ? Simplement une petite erreur lors de l'affichage... Dans le premier `printf`, on demande bien d'afficher la valeur de `ff` **mais** en indiquant au compilateur, à l'aide du format `%ld`, qu'il s'agit d'un entier long. Celui-ci va donc demander au processeur d'interpréter les 32 bits du nombre comme un entier long signé. De même, le deuxième `printf` affiche un entier long signé `ll` en l'interprétant comme un nombre flottant à afficher en notation scientifique ; les 32 bits de `ll` n'ont alors plus la même signification puisque le processeur va les décrypter comme signe, exposant et pseudo-mantisse... alors qu'ils représentent simplement les puissances de 2 !

Ce genre de problème est typique du langage C, qui impose peu de règles au niveau de la vérification des types, et plus rare en Java, sans toutefois être exclu. Voici d'ailleurs un exemple.

L'opérateur d'addition `+` ne travaille qu'avec des entiers au moins de type `int` ; cela explique que le programme suivant ne compile pas.

Listing 1.2

```
import java.util.*;
public class Pgm_Java {
    public static void main (String args[]) {
        short i = 32767;
        i = i+1;
        System.out.println("Valeur de i : " + i);
    }
}
```

On n'additionne pas des entiers courts

Lors de l'addition, la valeur de la variable `i` est promue au type `int` et le résultat de l'addition, étant du même type, ne peut se mettre dans la variable `i` de type `short`.

En revanche, l'opérateur d'incrément `++` n'a pas de contrainte de type. On peut donc écrire le même programme en remplaçant une ligne :

Listing 1.3

```
import java.util.*;
public class Pgm_Java {
    public static void main (String args[]) {
        short i = 32767;
        i++;
        System.out.println("Valeur de i : " + i);
    }
}
```

Mais on peut les incrémenter

Le résultat (`-32 768`) est alors inattendu mais pas surprenant car un `short` est un entier signé sur 16 bits.

RÉSUMÉ

Les nombres sont au cœur de l'ordinateur. Il est facile de représenter les entiers positifs à l'aide des puissances de 2 et les nombres négatifs à l'aide de la convention de représentation du complément à 2. En écrivant les nombres décimaux sous la forme « mantisse et exposant », on peut également travailler avec un sous-ensemble des nombres réels. Les bits peuvent aussi servir à représenter des caractères même s'il n'y a pas vraiment de correspondance universelle valable sur toutes les machines et tous les systèmes.

Dans tous les cas, il n'y a que des bits en mémoire et la manière de les lire n'est en aucun cas indiquée explicitement. C'est toujours le contexte, décidé par le programmeur, par le compilateur ou le processeur qui permet un calcul correct... ou complètement faux si ledit contexte n'est pas le bon.

Problèmes et exercices

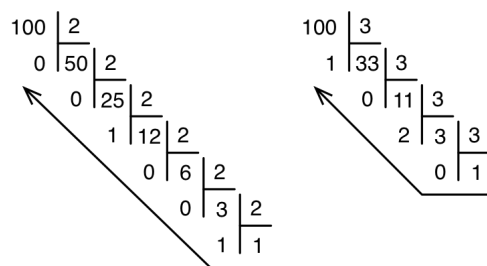
Les exercices suivants permettent de jongler avec les différentes bases, décimale, binaire ou hexadécimale. Vous allez compter en binaire et travailler avec des nombres négatifs et même des décimaux... pour vous apercevoir qu'un ordinateur ne calcule jamais parfaitement, qu'il peut déborder ou arrondir. Pour finir, vous verrez que c'est au programmeur de penser à la taille des valeurs qu'il manipule pour trouver le type de stockage adapté.

Exercice 1 : Écrire dans différentes bases

Exprimez le nombre décimal 100 dans toutes les bases de 2 à 9 et en base 16.

On effectue les divisions successives de 100 par 2 (voir figure 1.3) :

Figure 1.3



100 en bases 2 et 3.

Cela donne $1100100_2 = 100_{10}$. Le travail s'effectue de la même manière en base 3 : $10201_3 = 100_{10}$.

On obtient de façon identique les résultats suivants : $100_{10} = 1210_4 = 400_5 = 244_6 = 202_7 = 144_8 = 121_9 = 64_{16}$.

On peut obtenir certains résultats directement à partir d'autres. Ainsi, en regroupant les bits deux à deux, 01 10 01 00, on obtient le résultat en base 4 ; en les regroupant trois à trois, on obtient la base 8, et quatre à quatre (en n'oubliant pas le zéro en tête), on obtient l'hexadécimal. De même, on trouve la base 9 à partir de la base 3 en regroupant les symboles ternaires deux à deux (car $9 = 3^2$) : 01 02 01 donne bien 121_9 .

Exercice 2 : Représenter des entiers

Exprimez les nombres décimaux 94, 141, 163 et 197 en base 2, 8 et 16. Donnez sur 8 bits les représentations « signe et valeur absolue » et complément à 2 des nombres décimaux 45, 73, 84, -99, -102 et -118.

Toujours par la méthode des divisions et des regroupements de bits, on obtient :

$$94_{10} = 1011110_2 = 136_8 = 5e_{16}$$

$$141_{10} = 10001101_2 = 215_8 = 8d_{16}$$

$$163_{10} = 10100011_2 = 243_8 = a3_{16}$$

$$197_{10} = 11000101_2 = 305_8 = c5_{16}$$

N'oubliez pas d'utiliser les symboles a à f pour représenter les valeurs 10 à 15.

	signe et v.a.	compl. à 2
45	0010 1101	0010 1101
73	0100 1001	0100 1001
84	0101 0100	0101 0100
-99	1110 0011	1001 1101
-102	1110 0110	1001 1010
-118	1111 0110	1000 1010

Attention

Il ne faut pas confondre la représentation en complément à 2 et l'opération qui consiste à prendre le complément à 2, c'est-à-dire à inverser tous les bits du nombre et à ajouter 1. La représentation en complément à 2 consiste à écrire les nombres **négatifs** (et uniquement ceux-là) en prenant le complément à 2 de leur valeur absolue ; quelle que soit la représentation, les nombres positifs sont toujours écrits de la même façon.

Exercice 3 : Calculer en binaire

- 1) Effectuez la soustraction $122 - 43$ dans la représentation en complément à 2 en n'utilisant que l'addition.
- 2) Multipliez les nombres binaires 10111 et 1011 ; vérifiez le résultat en décimal.

1) $122 - 43 = 122 + (-43)$. Il faut donc calculer le complément à 2 de 43. La première question à se poser est de savoir de combien de bits on a besoin pour exprimer -43 et 122. 7 bits permettent bien d'écrire les nombres de 0 à 127 mais, en complément à 2, c'est l'intervalle de -64 à 63 qui est représenté sur 7 bits. Il en faut donc au moins 8.

$$\begin{array}{rcl}
 122 & : & \overset{1}{0} \overset{1}{1} \overset{1}{1} 1 1 0 1 0 \\
 -43 & : & 1 1 0 1 0 1 0 1 \\
 \hline
 =79 & : & 0 1 0 0 1 1 1 1
 \end{array}$$

On laisse bien sûr tomber la retenue finale.

2) $10111_2 = 23_{10}$ et $1011_2 = 11_{10}$.

$$\begin{array}{r}
 1 0 1 1 1 \\
 \times 1 0 1 1 \\
 \hline
 1 1 1 1 1 \\
 1 0 1 1 1 \\
 + 1 0 1 1 1 \bullet \\
 + 1 0 1 1 1 \bullet \bullet \bullet \\
 \hline
 1 1 1 1 1 1 0 1
 \end{array}$$

Le résultat correct est bien 253_{10} .

Exercice 4 : Débordement

On cherche à déterminer les cas de débordement lors d'une addition signée. On dit qu'il y a débordement lorsque le résultat obtenu (limité à la taille autorisée) n'est pas correct.

- 1) Effectuez en binaire sur 8 bits, en représentation en complément à 2, les opérations suivantes : $100 + 100$, $(-1) + (-2)$, $(-1) + 16$, $(-100) + (-100)$. Dans quel(s) cas y a-t-il débordement ?
- 2) On additionne 8 bits $A = a_7 \dots a_0$ avec les 8 bits $B = b_7 \dots b_0$, en obtenant un résultat $S = s_7 \dots s_0$ et une retenue r . Si A et B sont de signes contraires, peut-il y avoir débordement ? Si A et B sont de même signe (distinguez), quelles valeurs peuvent prendre s_7 et r ? Dans quel(s) cas y a-t-il débordement ? On

s'intéresse maintenant aux deux retenues de poids fort : r et r_6 (qui provient de l'addition de a_6 , b_6 et r_5 , et s'ajoute à a_7 et b_7 pour donner s_7 et r). En reprenant la question précédente, donnez l'expression du débordement en fonction de r_6 et r .

1) Voici les calculs :

$$\begin{array}{rcl}
 100 & : & \overset{1}{0} \overset{1}{1} 1 0 \overset{1}{0} 1 0 0 \\
 + 100 & : & 0 1 1 0 0 1 0 0 \\
 \hline
 & : & 1 1 0 0 1 0 0 0 \rightarrow -56 \\
 \\
 -1 & : & \overset{1}{1} \overset{1}{1} \overset{1}{1} 1 1 1 1 1 \\
 + 16 & : & 0 0 0 1 0 0 0 0 \\
 \hline
 & : & 0 0 0 0 1 1 1 1 \rightarrow 15 \\
 \\
 -1 & : & \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{1} 1 \\
 + -2 & : & 1 1 1 1 1 1 1 0 \\
 \hline
 & : & 1 1 1 1 1 1 0 1 \rightarrow -3 \\
 \\
 -100 & : & 1 0 \overset{1}{0} \overset{1}{1} \overset{1}{1} 1 0 0 \\
 + -100 & : & 1 0 0 1 1 1 0 0 \\
 \hline
 & : & 0 0 1 1 1 0 0 0 \rightarrow 56
 \end{array}$$

Il y a débordement dans le premier et le dernier cas. On pourrait croire que le résultat de $100 + 100$, c'est-à-dire 11001000_2 , est correct car il se lit bien $2^7 + 2^6 + 2^3 = 200$. Mais il ne faut pas oublier qu'on est en représentation en complément à 2 et qu'on ne peut donc pas choisir de le lire de cette façon-là. Il faut lire le résultat comme un nombre négatif.

2) Il ne peut pas y avoir de débordement lorsque l'on additionne deux nombres de signes contraires car la valeur absolue du résultat est toujours inférieure aux valeurs absolues des deux opérandes ; elle est donc toujours représentable.

Si A et B sont positifs, a_7 et b_7 valent 0. r vaut donc toujours 0 et s_7 peut valoir 0 (le résultat est donc positif, c'est correct) ou 1 (le résultat est négatif, il y a débordement). Si A et B sont négatifs, a_7 et b_7 valent 1. r vaut donc toujours 1 et s_7 peut valoir 1 (le résultat est donc négatif, c'est correct) ou 0 (le résultat est positif, il y a débordement). On a donc débordement si les deux opérandes sont de mêmes signes et si s_7 et r sont complémentaires.

Si les deux opérandes sont de signes contraires, l'un des deux bits de poids fort vaut 1 et l'autre 0. r_6 et r sont alors égaux et il n'y a jamais débordement. Si les deux opérandes sont positifs, le bit de poids fort des deux opérandes vaut 0 (donc r vaut 0) et le signe du résultat est identique à r_6 . Autrement dit, il y a débordement quand r_6 vaut 1 et r vaut 0. Si les deux opérandes sont négatifs, le bit de poids fort vaut 1 (donc r vaut 1) et le signe du résultat a la même valeur que r_6 . Il y a donc débordement quand r vaut 1 et r_6 vaut 0 (donnant un résultat, incorrect, positif). Au final, on a bien débordement quand r et r_6 sont complémentaires.

Exercice 5 : Travailler avec des réels

On considère une représentation simplifiée des réels en virgule flottante. Un nombre réel X est représenté par 10 bits $s\ eeee\ mmmmm$ où $X = (-1)^s \times 1, m \times 2^{e-7}$ avec un exposant sur 4 bits ($0 < e \leq 15$, un exposant égal à zéro désigne le nombre zéro quelle que soit la mantisse) et une pseudo-mantisse sur 5 bits (représentant les puissances négatives de 2).

- 1) Quels sont le plus grand nombre et le plus petit nombre strictement positifs représentables ?
- 2) Comment se représente le nombre 1 ? La précision ε d'une représentation est l'écart entre 1 et le nombre représentable suivant. Combien vaut ε pour cette représentation ?
- 3) Peut-on représenter 7,2 et 57,6 ? Quels sont les nombres qui encadrent 7,2 et 57,6 dans cette représentation ?
- 4) Que donne en décimal la multiplication de 7,25 et 28,5 ? Écrivez 7,25 et 28,5 dans la représentation proposée et effectuez la multiplication. Quels sont les deux arrondis possibles pour le résultat et leur valeur décimale ?

1) Le plus grand nombre représentable est $0\ 1111\ 1111 = (1 + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-5}) \times 2^8 = 504$. Le plus petit nombre strictement positif est $0\ 0001\ 0000 = 1 \times 2^{-6} = 0,015625$.

2) $1 = 1 \times 2^0$, donc 1 se représente $0\ 0111\ 0000$. Le nombre représentable suivant est $0\ 0111\ 00001$, qui vaut $(1 + 2^{-5}) \times 2^0$, donc ε vaut 2^{-5} .

3) $7,2 = 1,8 \times 2^2$. Mais on ne peut pas écrire une pseudo-mantisse égale à 0,8. Elle se situe entre 11001_2 (qui vaut 0,78125) et 11010_2 (qui vaut 0,8125). Les deux nombres qui encadrent 7,2 dans cette représentation sont donc $1,78125 \times 2^2 = 7,125$ et $1,8125 \times 2^2 = 7,25$. De même, $57,6 = 1,8 \times 2^5$ et on se retrouve dans la même situation, les deux nombres encadrant 57,6 étant alors $1,78125 \times 2^5 = 57$ et $1,8125 \times 2^5 = 58$. Remarquez que l'écart entre deux nombres de la représentation n'est pas constant mais augmente avec la valeur des nombres.

4) $7,25 \times 28,5 = 206,625$. On a vu que 7,25 se représente exactement par $0\ 1001\ 11010$ ($1,8125 \times 2^2$) et que 28,5 se représente exactement par $0\ 1011\ 11001$ ($1,78125 \times 2^4$). La multiplication des deux mantisses $1,11010_2$ et $1,11001_2$ donne $11,001110101_2$ comme résultat, qu'il faut normaliser à $1,1001110101_2$ en ajoutant 1 à l'exposant, qui devient $2 + 4 + 1 = 7$. Le problème est que cette mantisse a trop de bits significatifs pour cette représentation et qu'il faut donc l'arrondir. On peut tronquer la mantisse à $1,10011_2$ ou arrondir supérieurement à $1,10100_2$. le premier cas donne $1,10011_2 \times 2^7 = 204$ et le second $1,10100_2 \times 2^7 = 208$. On s'aperçoit alors que même si deux nombres sont exactement représentables en virgule flottante, leur produit ne l'est pas forcément, d'où la nécessité d'avoir des règles d'arrondi très précises pour être sûr que les résultats soient reproductibles d'une machine à une autre.

Exercice 6 : Approximation de réels

On considère la suite numérique :

$$U_{n+1} = -\frac{3}{8}U_n^2 + \frac{9}{4}U_n - \frac{3}{8}$$

- 1) Montrez que si $U_0 = 3$ ou si $U_0 = 1/3$, la suite est constante.
- 2) Programmez le calcul de la suite en C ou en Java et affichez les cent premières itérations.
- 3) Comment interprétez-vous les résultats ?.
- 4) Refaites l'exercice avec la suite :

$$U_{n+1} = -\frac{4}{9}U_n^2 + \frac{26}{9}U_n - \frac{4}{9}$$

- 1) On calcule simplement U_1 et, dans les deux cas, on aboutit à $U_1 = U_0$:

$$-\frac{3}{8} \times 3^2 + \frac{9}{4} \times 3 - \frac{3}{8} = 3 \quad \text{et} \quad -\frac{3}{8} \times \left(\frac{1}{3}\right)^2 + \frac{9}{4} \times \left(\frac{1}{3}\right) - \frac{3}{8} = \frac{1}{3}$$

La suite est donc constante.

- 2) Voici le code affichant les cent premières itérations en Java :

Listing 1.4

```
import java.util.*;
public class test_flottant {

    public static void Iterer(double u0) {
        double u = u0;
        int n;
        for (n = 0; n < 100; n++) {
            System.out.println("u" + n + " = " + u);
            u = -3.0*u*u/8.0 + 9.0*u/4.0 - 3.0/8.0;
        }
    }

    public static void main (String args[]) {
        double u;

        System.out.println("Hello World!");
        Iterer(3.0);
        System.out.println("Et maintenant pour 1/3");
        Iterer(1.0/3.0);
    }
}
```

Calcul d'une suite

Le comportement est sans surprise pour $U_0 = 3$, on obtient bien la valeur 3 pour les cent premières itérations. En revanche, lorsque $U_0 = 1/3$, les premières valeurs tournent autour de $1/3$ puis s'en éloignent et la valeur calculée finit par converger vers 3.

3) La suite est de la forme $U_{n+1} = f(U_n)$ et, si elle converge, c'est vers un point fixe de f , c'est-à-dire une solution de l'équation $f(X) = X$. En résolvant cette équation du second degré, on trouve que f admet deux points fixes, 3 et $1/3$.

Le point fixe 3 est stable (la dérivée de f en 3 vaut 0), ce qui implique que le calcul est numériquement stable : si on itère f à partir d'une valeur proche de 3, les valeurs successivement obtenues se rapprochent du point fixe 3 ; c'est bien ce que l'on obtient avec le programme précédent.

En revanche, le point fixe $1/3$ est instable (la dérivée de f en $1/3$ vaut 2). En d'autres termes, des itérations commencées à une valeur proche de $1/3$ s'en éloignent. Dans le programme précédent, on démarre les itérations avec la valeur $1/3$; on devrait donc rester sur le point fixe, mais $1/3$ n'est pas exactement représentable en puissances négatives de 2. Par conséquent, le programme débute le calcul avec une valeur binaire très proche de $1/3$ mais pas parfaitement égale à $1/3$. Cela suffit pour faire diverger le calcul vers l'autre point fixe, 3.

4) Il y a toujours deux points fixes, 4 et $1/4$. On peut écrire le même programme pour calculer les itérations et, contrairement au premier exemple, que l'on commence avec 4 ou $1/4$, toutes les itérations gardent la valeur de départ. Cela s'explique par le fait que maintenant, $1/4$ se représente exactement en machine à l'aide des puissances de 2 : le calcul se fait à partir du point fixe et n'a donc aucune raison de diverger.

Exercice 7 : Type des variables

1) On considère l'opération décimale $101 + 99$. Expliquez pourquoi un programme peut « donner » deux résultats différents. Quels sont ces résultats et de quoi dépend le fait que le programme donne l'un ou l'autre ?

2) Un nouveau programme est en train de compter le nombre de caractères d'un fichier, qui en réunit environ cinquante mille. Finalement le programme s'arrête et affiche -14 532. Quel bogue avez-vous fait et quel est le nombre de caractères du fichier ?

3) On considère le programme C suivant :

Listing 1.5

```
#include <stdio.h>
main() {
    char c;
    char str[500];

    gets(str);      /* récupère une chaîne et la met dans str */
    c = strlen(str); /* renvoie la longueur de la chaîne      */
    printf("longueur : %d\n", c);
}
```

Un caractère, c'est peu

Ce programme affiche-t-il toujours la bonne valeur ? Quelles sont ses limites de fonctionnement ?

4) On considère le programme C suivant :

Listing 1.6

```
#include <stdio.h>
main() {
    short i;
    unsigned short j;

    i = -1;
    j = i;
    printf("%d\n", j);
}
```

Signé ou non signé, il faut choisir

Quelle est la valeur affichée ? Quelle serait la valeur affichée si la représentation des nombres était « signe et valeur absolue » ?

- 1) Si tout se passe bien, le résultat est celui attendu, c'est-à-dire 200. Mais si les deux nombres 101 et 99 sont stockés sur un octet signé, le résultat est aussi stocké sur un octet signé, qui ne peut pas représenter 200 et donne comme résultat -56. La représentation binaire du résultat est toujours la même (11001000) mais le programme « lit » cette valeur de deux manières différentes suivant le type de la variable. Le seul type posant problème est l'octet signé (char en C) car il représente les nombres de -128 à 127 ; tous les autres types donnent un résultat correct, entre autres le type unsigned char. En Java, l'addition porte forcément sur des entiers de type int, donc sur 32 bits. Le résultat du calcul proposé est donc toujours correct. Mais s'il avait dépassé 2^{31} , le même problème se poserait.
- 2) En C ou en Java, on utilise une variable de type short qui stocke un nombre signé sur 16 bits, donc une valeur de -32 768 à 32 767. Il est impossible d'afficher une valeur proche de 50 000. Le nombre affiché est donc le complément à 2 du nombre réel, qui est de $65\,536 - 14\,532 = 51\,004$ caractères.
- 3) Une variable de type char en C peut stocker une valeur de -128 à 127. Si la chaîne récupérée fait moins de cent vingt-huit caractères, le programme affiche la bonne valeur ; si elle fait plus, le programme affiche soit une valeur négative (par exemple, si la chaîne a une longueur entre 128 et 255), soit le modulo 256 de la longueur (par exemple, si la chaîne a une longueur entre 256 et 384), voire une vague combinaison des deux si la chaîne est plus grande... Si l'on veut pouvoir mettre une chaîne de longueur inférieure à 255, il faut au minimum utiliser une variable de type unsigned char ; si la chaîne peut être plus longue, un short ou un int est nécessaire. Ici, *théoriquement*, la chaîne est limitée à cinq cents caractères par la taille du tableau mais gets() ne fait aucune vérification. Si l'utilisateur entre une chaîne plus longue, il y a des problèmes d'écriture mémoire sur une zone non prévue, mais ce n'est pas le sujet de l'exercice.
- 4) -1 se représente en binaire comme 11...11 (sur 16 bits). Comme j est un unsigned short, on doit le lire comme un nombre positif ; le programme affiche donc 65 535. Si la représentation était « signe et valeur absolue », -1 s'écrirait 100...001 et donc le programme afficherait 32 769.

Chapitre 2

Circuits logiques

Le transistor, un interrupteur commandé par une tension, est le composant de base des circuits électroniques au cœur de l'ordinateur. À l'aide des transistors, on construit des circuits logiques élémentaires, des portes logiques, effectuant des opérations simples. Ces circuits sont analysés grâce à l'algèbre de Boole, outil mathématique puissant qui permet de développer des circuits plus complexes. Ces mêmes portes logiques sont utilisées pour réaliser des circuits séquentiels, faisant intervenir une composante temporelle et introduisant ainsi la notion de mémorisation de valeurs dans les systèmes.

Circuits logiques

- 1. Fonctions booléennes..... 22
- 2. Circuits combinatoires..... 30
- 3. Circuits logiques séquentiels..... 34

Problèmes et exercices

- 1. Construire une fonction logique..... 40
- 2. Un circuit qui calcule..... 41
- 3. Additionneur/soustracteur..... 42
- 4. Multiplicateur..... 44
- 5. Additionneur à retenue anticipée..... 46
- 6. Comparateur de nombres..... 47
- 7. Bascules équivalentes..... 48
- 8. Bascule maître-esclave..... 49

L'information véhiculée sur les fils internes de l'ordinateur peut prendre deux valeurs, 0 ou 1, correspondant à deux niveaux de tension. Classiquement, le 0 est associé au niveau bas de la tension, proche de 0 V, alors que le 1 est associé au niveau haut, historiquement de 5 V, mais qui a diminué et est maintenant proche de 1 V. Cette information circule à travers les différents circuits de l'ordinateur, travaillant en logique numérique, qui sont modélisés par l'algèbre de Boole.

Cette logique numérique est à opposer au calcul analogique, dans lequel l'entrée peut prendre n'importe quelle valeur d'un intervalle continu. Le calcul analogique s'est développé au XIX^e siècle et au début du XX^e siècle mais a dû céder sa place en raison d'un manque de précision : comme toutes les valeurs sont autorisées, la moindre perturbation change le calcul ; à l'inverse, comme le numérique n'autorise que deux valeurs, une légère fluctuation de tension sur un fil ne modifie rien.

Tous les circuits électroniques sont composés de transistors, de l'ordre du milliard dans les processeurs actuels. Comme il est impossible d'étudier leur comportement individuel, ils ont été modélisés et regroupés sous forme de circuits élémentaires que nous allons présenter. Tous les circuits intégrés sont composés, sur une grande échelle, de ces circuits élémentaires.

Nous allons distinguer deux types de circuits logiques dont le fonctionnement est différent : les circuits combinatoires et les circuits séquentiels. Les premiers expriment simplement leurs sorties à partir de leurs entrées ; les seconds font dépendre leurs sorties de leurs entrées mais également des sorties précédentes.

I. FONCTIONS BOOLÉENNES

L'analyse des circuits logiques nécessite l'utilisation d'une algèbre spécifique, dite « booléenne », fruit des travaux du mathématicien anglais du XIX^e siècle George Boole. Ces travaux ont défini un ensemble d'opérateurs de base, ainsi que leurs propriétés, qui composent une algèbre permettant de concevoir tout type de circuit. La construction de fonctions logiques répondant à des contraintes précises et leur représentation sous forme de circuits électroniques se font à l'aide d'opérateurs élémentaires.

I.1 Définitions

Alors que dans l'algèbre classique les variables et les fonctions peuvent prendre n'importe quelles valeurs, elles sont ici limitées aux valeurs 0 et 1. Une fonction de n variables booléennes va être définie de $\{0,1\}^n$ dans $\{0,1\}$. Elle est même entièrement définie par les valeurs qu'elle prend sur les 2^n combinaisons possibles de ses variables d'entrée ; comme chaque valeur de la fonction ne peut être que 0 ou 1, il y a 2^{2^n} fonctions différentes de n variables. On peut alors décrire une fonction (à n variables) donnée en explicitant ses 2^n valeurs, par exemple par sa table de vérité. Il s'agit d'un tableau à 2^n lignes, qui liste pour chaque combinaison possible des variables d'entrée (une ligne) la valeur prise par la fonction. Voici un exemple :

Tableau 2.1

a	b	c	F(a,b,c)
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Une table de vérité

L'ordre des lignes n'a pas d'importance mais, par tradition, et aussi pour faciliter la lecture et les comparaisons, on écrit souvent les 2^n combinaisons dans l'ordre numérique, comme si elles représentaient un nombre binaire (de 0 à 7 dans le tableau 2.1).

Il peut être fastidieux de donner une table de vérité pour une fonction de plus de trois variables. Il est alors possible d'exprimer la fonction par son expression algébrique à partir de fonctions de base que nous allons présenter plus loin.

Note

La table de vérité décrit intégralement la fonction alors qu'une expression algébrique permet de la calculer. On peut donc comparer directement deux tables de vérité pour décider de l'égalité de deux fonctions. En revanche, comme deux expressions algébriques différentes peuvent représenter les mêmes fonctions, il est moins facile sur la base de ces expressions de décider de l'égalité des fonctions en question.

Un circuit logique a souvent plusieurs sorties. Dans le cas des circuits combinatoires, les sorties, ne dépendant que des entrées, sont donc indépendantes. On caractérise alors le circuit par plusieurs fonctions indépendantes, chacune représentant une sortie. La table de vérité du circuit contient toujours 2^n lignes et autant de colonnes de sortie que de fonctions.

1.2 Fonctions et portes logiques élémentaires

Un certain nombre de fonctions booléennes forment les fonctions logiques de base qui permettent d'en construire des plus complexes.

Ces fonctions de base peuvent être définies par leur table de vérité ou leur expression algébrique, mais elles ont également été implémentées sous forme de circuits électroniques, appelés « portes logiques » (de l'anglais *gate*), qui ont des symboles graphiques normalisés. Deux principales normes existent pour représenter ces portes de base : la première, la plus ancienne, affecte une forme géométrique différente à chacune des fonctions. La seconde dessine chacun des circuits sous une forme rectangulaire et les distingue via un symbole placé à l'intérieur du rectangle ; elle permet alors de représenter beaucoup plus de circuits différents. Nous présentons ici, pour chaque porte, les deux symboles, mais nous utiliserons ensuite la nouvelle norme.

Fonction NON (NOT)

Cette fonction, la plus simple, a une entrée booléenne et une sortie, qui se définit simplement comme le complémentaire de l'entrée. Son expression algébrique est :

$$NON(a) = \bar{a}$$

Elle se lit « a-barre » ou « non-a ». On dit aussi que la variable a est « complémentée ».

Tableau 2.2

a	NON(a)
0	1
1	0

Table de vérité du NON

Les deux représentations graphiques de la porte NON sont données à la figure 2.1. Dans ces deux formes, le symbole important est en fait le petit rond se trouvant à la sortie et qui indique la négation (on le retrouvera dans d'autres symboles).

Fonction OU (OR)

Cette fonction de deux variables prend la valeur 1 si l'une ou l'autre de ses entrées (ou les deux) est à 1. Elle vaut 0 si les deux entrées sont à 0. Son expression algébrique est :

$$OU(a, b) = a + b$$

Attention, elle se lit bien « a ou b », et non « a plus b » (cela n'a rien à voir avec l'addition).

Les deux représentations graphiques de la porte OU sont données à la figure 2.1 et sa table de vérité au tableau 2.3.

Fonction ET (AND)

Cette fonction de deux variables prend la valeur 1 si ses entrées sont l'une et l'autre à 1. Elle vaut 0 si au moins une des deux entrées est à 0. Son expression algébrique est :

$$ET(a, b) = a.b = ab$$

Elle se lit « a et b ». Le point séparant les deux variables est presque toujours omis.

Tableau 2.3

a	b	ET(a,b) = ab	OU(a,b) = a + b
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Tables de vérité du ET et du OU

Les deux représentations graphiques de la porte ET sont données à la figure 2.1.

Fonction OU-exclusif (XOR)

Cette fonction de deux variables prend la valeur 1 si l'une ou l'autre de ses entrées est à 1, mais pas les deux. Elle vaut 0 si les deux entrées sont égales (à 0 ou à 1). Son expression algébrique est :

$$XOR(a, b) = a \oplus b$$

Elle se lit « a ou-exclusif b ».

Tableau 2.4

a	b	XOR(a,b)
0	0	0
0	1	1
1	0	1
1	1	0

Table de vérité du OU-exclusif

Les deux représentations graphiques de la porte OU-exclusif sont données à la figure 2.1.

Fonction NON-OU (NOR)

En combinant les quatre fonctions de base précédentes, on peut construire des fonctions composées. En voici deux très utiles.

La première est NON-OU. Elle correspond à une fonction OU suivie d'un NON. Elle prend donc la valeur 1 uniquement quand ses deux entrées sont à 0. Son expression algébrique est :

$$NON-OU(a, b) = \overline{a + b}$$

Les deux représentations graphiques de la porte NON-OU sont données à la figure 2.1 et sa table de vérité au tableau 2.5. On retrouve dans ces dessins le symbole de la porte OU suivi du petit rond qui représente la porte NON.

Fonction NON-ET (NAND)

La seconde est la fonction NON-ET. Elle correspond à une fonction ET suivie d'un NON. Elle prend donc la valeur 1 lorsqu'au moins une de ses deux entrées est à 1. Son expression algébrique est :

$$NON-ET(a, b) = \overline{a \cdot b} = \overline{ab}$$

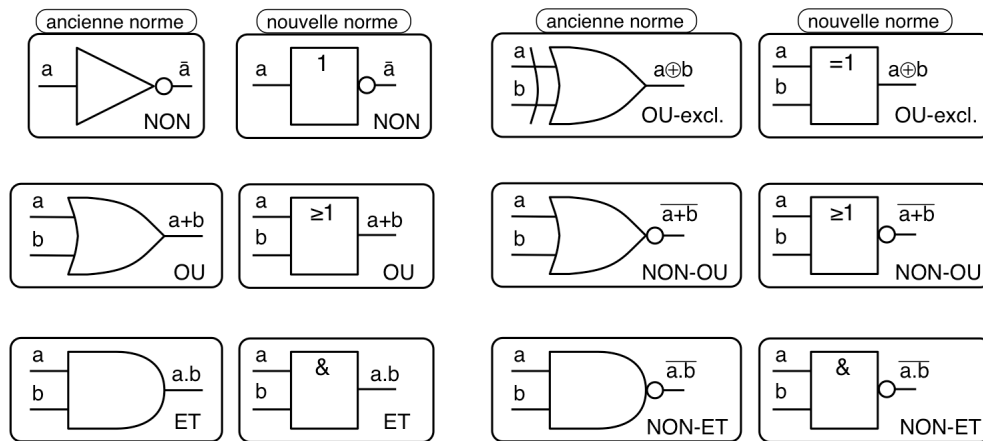
Tableau 2.5

a	b	NON-ET(a,b)	NON-OU(a,b)
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

Tables de vérité du NON-ET et du NON-OU

Les deux représentations graphiques de la porte NON-ET sont données à la figure 2.1. On a encore ici la première porte ET suivie du petit rond pour le NON.

Figure 2.1



Symboles des portes logiques.

1.3 Propriétés des fonctions

Les fonctions et les circuits logiques étant souvent définis à partir d'expressions algébriques, il est important de pouvoir simplifier celles-ci afin de réduire la complexité des circuits. À cet effet, un certain nombre de propriétés algébriques peuvent être obtenues à partir des définitions des fonctions. Pour prouver ces égalités, il suffit d'écrire les tables de vérité de chaque expression et de constater qu'elles sont identiques.

Remarque

Dans les expressions algébriques, l'opérateur ET est prioritaire sur l'opérateur OU. On peut donc souvent enlever les parenthèses et écrire $ab + c$ plutôt que $(ab) + c$.

Tableau 2.6

Commutativité	Associativité	Distributivité
$a + b = b + a$ $ab = ba$ $a \oplus b = b \oplus a$	$a + (b + c) = (a + b) + c = a + b + c$ $a(bc) = (ab)c = abc$ $a \oplus (b \oplus c) = (a \oplus b) \oplus c = a \oplus b \oplus c$	$a + (bc) = (a + b)(a + c)$ $a(b + c) = (ab) + (ac) = ab + ac$ $a(b \oplus c) = (ab) \oplus (ac) = ab \oplus ac$
Élément neutre	Élément absorbant	Idempotence
$a + 0 = a$ $1.a = a$ $a \oplus 0 = a$	$a + 1 = 1$ $0.a = 0$	$a + a = a$ $aa = a$
Complémentaire	Lois de De Morgan	Divers
$a + \bar{a} = 1; \quad a\bar{a} = 0$ $\overline{aa} = \bar{a}; \quad \overline{a + a} = \bar{a}$ $\bar{\bar{a}} = a$ $a \oplus \bar{a} = 1$ $a \oplus 1 = \bar{a}$	$\overline{ab} = \bar{a} + \bar{b}$ $\overline{a + b} = \bar{a}\bar{b}$	$a + ab = a(a + b) = a$ $a + (\bar{a}b) = a + b; \quad a(\bar{a} + b) = ab$ $a \oplus a = 0$ $a \oplus \bar{b} = \bar{a} \oplus b = \overline{a \oplus b}$ $\bar{a} \oplus \bar{b} = a \oplus b$ $a \oplus b = \bar{a}\bar{b} + \bar{a}b$ $\overline{a \oplus b} = ab + \bar{a}\bar{b}$

Propriétés algébriques des opérateurs

Les lois de De Morgan sont très importantes car elles permettent de transformer un opérateur en un autre dans une expression algébrique. On peut donc, dans une expression, remplacer tous les opérateurs OU par des opérateurs ET (ou vice-versa). L'intérêt est par exemple de pouvoir choisir l'opérateur utilisé pour construire la fonction à partir de son expression, selon des portes logiques disponibles.

La propriété d'associativité permet de définir les opérateurs ET, OU, NON-ET et NON-OU à plus de deux entrées :

- La fonction OU donne un résultat égal à 1 si au moins une de ses entrées est à 1.
- La fonction ET donne un résultat égal à 1 si toutes ses entrées sont à 1.
- La fonction NON-OU donne un résultat égal à 1 si aucune de ses entrées n'est à 1.
- La fonction NON-ET donne un résultat égal à 1 si au moins une de ses entrées est à 0.

Les schémas des circuits sont comparables qu'il y ait deux entrées ou plus : on ajoute simplement autant de lignes que de variables d'entrée supplémentaires.

Le OU-exclusif peut également se généraliser à plus de deux entrées mais cela est beaucoup moins classique. Un circuit OU-exclusif à n entrées fournit une sortie égale à 1 s'il y a un nombre impair d'entrées qui ont la valeur 1 (voir tableau 2.7).

Tableau 2.7

a	b	c	OU(a,b,c)	ET(a,b,c)	NON-OU(a,b,c)	NON-ET(a,b,c)	XOR(a,b,c)
0	0	0	0	0	1	1	0
0	0	1	1	0	0	1	1
0	1	0	1	0	0	1	1
0	1	1	1	0	0	1	0
1	0	0	1	0	0	1	1
1	0	1	1	0	0	1	0
1	1	0	1	0	0	1	0
1	1	1	1	1	0	0	1

Fonctions à trois entrées

I.4 Construction d'une fonction quelconque

Une fonction booléenne est entièrement définie par sa table de vérité mais cela n'est pas d'une grande aide pour implémenter cette fonction sous forme de circuit combinatoire. En effet, les seuls circuits élémentaires à disposition sont les portes logiques et celles-ci reproduisent les opérateurs de base (NON, OU, ET...) qui sont dans l'expression algébrique d'une fonction. Il faut donc transformer la table de vérité de la fonction en expression algébrique afin de pouvoir la calculer et surtout de pouvoir physiquement l'implémenter à l'aide des portes logiques.

Mintermes et forme canonique disjonctive

Soit une fonction définie par sa table de vérité, par exemple la fonction de trois variables suivante :

Tableau 2.8

a	b	c	F(a,b,c)
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

La table de vérité de F

Nous allons nous intéresser aux lignes pour lesquelles la fonction vaut 1. Prenons par exemple la quatrième : la fonction vaut 1 quand a vaut 0, et b et c valent 1. Il existe donc une expression qui vaut 1 quand les variables ont exactement ces valeurs. Il s'agit de $\bar{a}bc$. En effet, comme nous sommes devant un ET, il faut que les trois termes prennent la valeur 1 pour que le résultat soit 1 ; la variable a étant complémentée, elle doit en fait valoir 0.

Cette expression s'appelle un « minterme ». C'est un ET de toutes les variables d'entrée de la fonction, où chaque variable est éventuellement complémentée. Un minterme prend la valeur 1 uniquement pour la combinaison indiquée des variables (si la variable est écrite directement, elle doit valoir 1 ; si elle est complémentée, elle doit valoir 0).

On peut maintenant écrire tous les mintermes correspondant aux cas où la fonction vaut 1. Il s'agit de : $\bar{a}\bar{b}\bar{c}$, $\bar{a}\bar{b}c$, $\bar{a}b\bar{c}$ et $ab\bar{c}$. On termine en exprimant le fait qu'il suffit d'être dans un de ces cas pour que F vaille 1 ; on doit se trouver dans le premier cas, ou le deuxième, ou le troisième, etc.

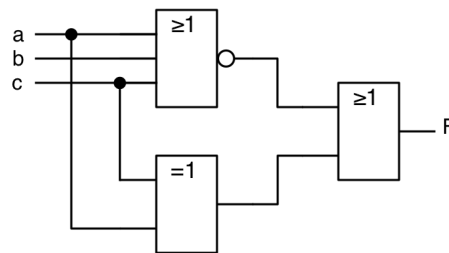
$$F(a,b,c) = \bar{a}\bar{b}\bar{c} + \bar{a}\bar{b}c + \bar{a}b\bar{c} + ab\bar{c}$$

Cette écriture de la fonction s'appelle la « forme canonique disjonctive » ou « somme canonique ». Il est tout à fait possible (et même souhaitable) de simplifier ensuite l'expression algébrique de la fonction à l'aide des propriétés booléennes. On pourrait ainsi réduire F à l'expression suivante (plusieurs expressions réduites sont possibles, toutes équivalentes) :

$$\begin{aligned} F(a,b,c) &= \bar{a}\bar{b}\bar{c} + \bar{a}\bar{b}c + \bar{a}b\bar{c} + ab\bar{c} \\ &= \bar{a}\bar{b}\bar{c} + (\bar{a}c + a\bar{c})\bar{b} + (\bar{a}c + a\bar{c})b \\ &= \bar{a}\bar{b}\bar{c} + (a \oplus c)(\bar{b} + b) = \bar{a}\bar{b}\bar{c} + (a \oplus c) \\ &= a + b + c + (a \oplus c) \end{aligned}$$

Cela permet de dessiner le circuit combinatoire équivalent (voir figure 2.2).

Figure 2.2



Circuit combinatoire équivalent à F .

Opérateur complet

Toute fonction logique (sauf la fonction nulle) peut se mettre sous sa forme canonique disjonctive, comme OU de mintermes. Quels sont alors les circuits nécessaires pour construire une fonction logique ?

- Les variables de chaque minterme sont éventuellement complémentée, il faut donc des portes NON.
- Chaque minterme est un ET des variables, il faut donc des portes ET à plusieurs entrées (ou par associativité, plusieurs étages successifs de portes ET à deux entrées).
- Pour finir, il faut une porte OU à plusieurs entrées afin de faire la « somme » de tous les mintermes.

On peut donc calculer n'importe quelle fonction logique à l'aide de portes NON, ET et OU. Notez que l'on peut exprimer le NON à l'aide d'un NON-ET en dédoublant l'entrée : $\bar{a} = a.a$. On peut faire de même pour le OU, à l'aide de NON et de NON-ET, en utilisant les lois de De Morgan. Pour preuve, l'expression suivante, qui n'emploie que des NON-ET :

$$a + b = \overline{\bar{a} \cdot \bar{b}} = \overline{a \cdot b}$$

De même, le ET n'est qu'un NON-ET suivi d'un NON :

$$ab = \overline{\overline{ab}} = \overline{a \cdot b}$$

Les trois portes de base peuvent donc chacune être remplacée par une ou plusieurs portes NON-ET.

Comme il suffit de ces trois portes pour exprimer n'importe quelle fonction, on en conclut que le NON-ET est un opérateur complet, permettant d'écrire n'importe quelle fonction en n'utilisant que des portes NON-ET dans le circuit. Cette propriété peut être intéressante lors de la construction de circuits combinatoires pour minimiser le nombre de types différents de portes.

Remarque

On a exactement la même propriété pour l'opérateur complet NON-OU qui peut remplacer des NON, des ET et des OU.

Tableaux de Karnaugh

Pour une fonction de deux ou trois variables, il est simple d'utiliser la table de vérité pour exprimer et ensuite simplifier l'expression algébrique de la fonction. Si l'on a une fonction de quatre ou cinq variables, il existe une méthode graphique permettant d'arriver au même résultat : le tableau de Karnaugh.

Il s'agit de représenter la table de vérité sous forme matricielle, en réunissant deux variables sur quatre lignes et deux ou trois variables sur quatre ou huit colonnes.

Chaque ligne et chaque colonne représente un état (0 ou 1) de la variable, une case du tableau étant alors la valeur de la fonction pour la combinaison indiquée des variables (voir deux exemples à la figure 2.3).

Figure 2.3

ba	dc			
	00	01	11	10
00				
01				
11				
10				

cba	ed							
	000	001	011	010	110	111	101	100
00								
01								
11								
10								

Tableaux de Karnaugh pour quatre et cinq variables.

Les lignes et colonnes ne sont pas disposées au hasard mais de telle sorte que, entre deux cases adjacentes (horizontalement ou verticalement), il n'y ait qu'une variable qui change d'état.

On remplit ce tableau à l'aide des valeurs que prend la fonction. On peut donc développer la forme canonique en extrayant toutes les cases (et les mintermes correspondants) où la fonction vaut 1. Mais le principe de ces tableaux est de regrouper ces cases par ensembles de deux, quatre, huit, seize, etc., pour simplifier les mintermes. En regroupant deux cases contiguës, on peut éliminer une variable d'un minterme, en regroupant quatre, en ligne ou en carré, on en élimine deux, et ainsi de suite. Considérons les deux tableaux de la figure 2.4.

Figure 2.4

ba	dc			
	00	01	11	10
00	1	1	0	0
01	0	1	1	1
11	0	1	1	0
10	0	0	0	1

ba	dc			
	00	01	11	10
00	1	0	0	1
01	1	1	0	0
11	0	0	1	1
10	1	0	0	1

Simplification de tableaux.

Dans le premier tableau, on a choisi d'effectuer deux regroupements : le premier, en haut à gauche, permet d'éliminer la variable a dans deux mintermes pour aboutir au terme $\bar{b}\bar{c}\bar{d}$; le second regroupement, celui de quatre cases, élimine les variables b et d et donne ac . Avec les deux cases isolées restantes, on aboutit à l'expression algébrique de la fonction :

$$F_1 = \bar{b}\bar{c}\bar{d} + ac + \bar{a}b\bar{c}\bar{d} + \bar{a}b\bar{c}d$$

Le second tableau montre un regroupement sur les cases extrêmes : la propriété d'adjacence (une seule variable change d'état) est également vérifiée entre les cases de bords opposés ; le tableau est en fait un tore (c'est-à-dire un carré dans lequel les bords opposés seraient voisins). On a deux regroupements de deux cases donnant les termes $\bar{b}c\bar{d}$ et bcd , et les quatre cases extrêmes formant un carré, ce qui génère le terme $\bar{a}\bar{c}$. L'expression de la fonction est donc :

$$F_2 = \bar{b}c\bar{d} + bcd + \bar{a}\bar{c}$$

Il est plus simple de travailler avec les tableaux de Karnaugh que sur les mintermes d'une fonction. À titre d'exemple, le tableau 2.9 donne la table de vérité de la première fonction. Il y a donc huit mintermes à écrire et à simplifier, pour aboutir à l'expression de la fonction obtenue directement à l'aide du tableau.

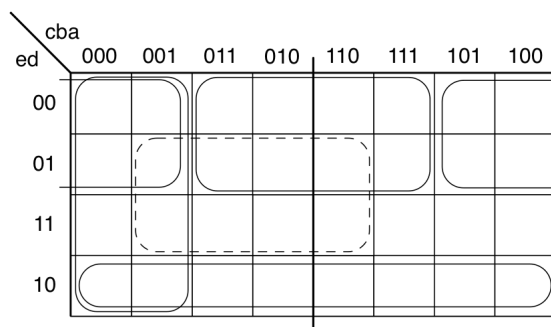
Tableau 2.9

a	b	c	d	$F_1(a,b,c,d)$
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

La table de vérité de la première fonction

Le troisième tableau de Karnaugh (voir figure 2.5), à cinq variables, est un peu spécial. Il faut le considérer comme deux sous-tableaux 4×4 superposés pour choisir les regroupements. Ceux de huit cases peuvent se faire dans un même sous-tableau ou sous forme de « cube », deux paquets superposés de quatre cases, un dans chaque sous-tableau. Dans l'exemple de la figure 2.5, on voit quatre possibilités de groupes de huit en traits pleins. Le groupe en pointillés n'est pas valide car il ne correspond pas aux mêmes cases dans les deux sous-tableaux ; il faut le découper en deux carrés de quatre cases.

Figure 2.5



Regroupements dans un tableau à cinq variables.

Au-delà de cinq variables, on ne peut plus utiliser de tableaux de Karnaugh. Il faut de nouveau partir de la table de vérité en espérant pouvoir simplifier l'expression, probablement « monstrueuse », obtenue pour la fonction.

Note

Table de vérité incomplète

Parfois, une fonction n'est pas complètement définie. Plus précisément, pour certaines combinaisons des entrées, les valeurs présentes sur certaines sorties n'ont pas d'importance. On peut alors jouer avec cela pour simplifier l'expression de la fonction. En forçant intelligemment les valeurs « libres » à 0 ou 1, on peut créer des regroupements dans le tableau de Karnaugh correspondant.

Optimisations

On peut avoir plusieurs objectifs lorsque l'on cherche à simplifier l'expression algébrique d'une fonction. On peut souhaiter n'utiliser qu'un seul type de porte pour simplifier la fabrication du circuit. Si l'on veut au contraire réduire la place qu'il occupe, il faut minimiser le nombre de portes. Malheureusement, ce n'est pas aussi facile que cela car toutes les portes ne sont pas équivalentes en termes d'occupation physique. Paradoxalement, en raison de leur construction électronique, ce sont les portes NON-ET et NON-OU qui sont les plus petites ; une porte ET correspond souvent à un circuit NON-ET suivi de l'opérateur NON. On ne peut donc pas optimiser la fonction sans avoir des détails sur l'implémentation physique des circuits.

Enfin, l'objectif de l'optimisation peut être la minimisation du temps de propagation électrique afin d'obtenir un circuit plus rapide. Nous avons jusqu'à présent fait la supposition que les portes réagissaient instantanément aux entrées alors qu'en réalité l'établissement de la ou des sorties prend un certain temps. Certes, ce temps peut sembler très faible, de l'ordre de la dizaine de nanosecondes, mais il est non négligeable à l'échelle de temps informatique. On peut donc souhaiter avoir le moins de portes entre les entrées et une sortie pour obtenir le délai de propagation le plus faible possible. Cela peut amener à redessiner le circuit autrement, avec plus de portes au total, mais disposées en parallèle, ce qui leur permet de travailler en même temps.

2. CIRCUITS COMBINATOIRES

La plupart des différentes opérations effectuées au sein d'un ordinateur sont implémentées sous la forme de circuits logiques construits à partir de portes logiques. Il est bien sûr impossible d'illustrer tous les circuits utilisés. C'est pourquoi nous nous contenterons de présenter dans cette section quelques circuits classiques que l'on retrouve systématiquement dans les machines, afin de montrer comment ils s'élaborent.

2.1 Décodeur

Le décodeur est un circuit qui permet de sélectionner une ligne à partir de son adresse binaire. Plus précisément, le décodeur n vers 2^n a n entrées et 2^n sorties. Lorsqu'un nombre binaire sur n bits se présente sur les entrées, le décodeur met un 1 sur la sortie dont le numéro correspond à cette valeur binaire.

Une des principales applications du décodeur est la sélection de boîtiers mémoire. En effet, lorsque l'ordinateur fait référence à une case mémoire, il la désigne par son adresse binaire. Celle-ci doit être décodée pour qu'un et un seul boîtier mémoire soit sélectionné. Pour ce faire, on peut utiliser un circuit décodeur qui, à partir de l'adresse mémoire, met à 1 une et une seule des lignes de validation en sortie, lignes reliées aux différents boîtiers.

Tableau 2.10

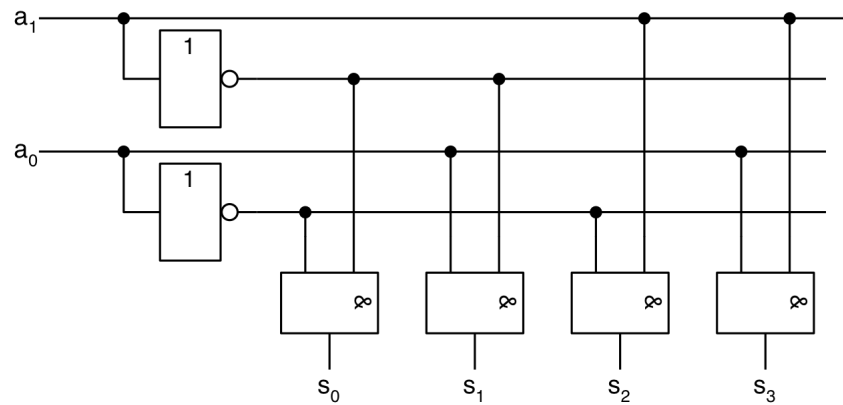
a_1	a_0	s_0	s_1	s_2	s_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Table de vérité du décodeur

Il est simple de construire un décodeur, car chaque sortie est à 1 uniquement sur une combinaison précise des entrées. Par exemple, la sortie 0 est active lorsque toutes les entrées sont à 0, donc pour le minterme

$\bar{a}_{n+1}\bar{a}_{n+2}\dots\bar{a}_1\bar{a}_0$. On a donc un simple ET à n entrées qui commande chaque sortie, les entrées de ce ET étant les variables d'entrée, éventuellement complémentées. La figure 2.6 montre un décodeur 2 vers 4. Les deux entrées a_1a_0 peuvent prendre les valeurs 0 ou 1, donnant quatre combinaisons possibles, chacune sélectionnant une sortie.

Figure 2.6



Décodeur 2 vers 4.

2.2 Multiplexeur

Le multiplexeur 2^n vers 1 permet de choisir une entrée parmi 2^n et de la recopier sur la sortie. Pour la sélection de l'entrée, il y a n lignes de commande qui codent le numéro en binaire de l'entrée voulue.

L'intérêt du multiplexeur est de pouvoir connecter plusieurs entrées à un même circuit et de sélectionner l'entrée voulue simplement en indiquant son adresse sur les lignes de commande. On peut ainsi relier plusieurs composants entre eux en minimisant les fils de connexion.

Ce circuit est basé sur le même principe qu'un décodeur : il y a autant de portes ET que d'entrées mais chacune n'est activée que lorsque la combinaison correspondante est présente sur les lignes de commande. On réunit ensuite les sorties des ET sur un OU pour récupérer celle qui est valide.

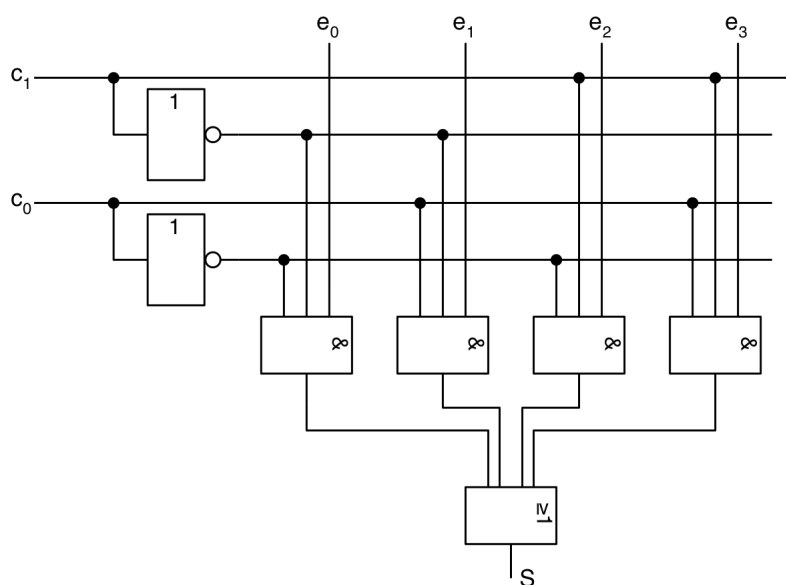
La figure 2.7 illustre un multiplexeur 4 vers 1 où les deux lignes de commande c_1c_0 indiquent quelle entrée parmi les quatre (e_1, e_2, e_3 ou e_4) doit être transférée sur la sortie.

Tableau 2.11

c_1	c_0	S
0	0	e_0
0	1	e_1
1	0	e_2
1	1	e_3

Table de vérité du multiplexeur

Figure 2.7



Multiplexeur 4 vers 1.

2.3 Additionneur

Un ordinateur fait des calculs et utilise pour cela des circuits logiques. La plus simple est l'addition qui nécessite deux types de circuits.

Demi-additionneur

Pour additionner deux nombres binaires, il faut d'abord additionner les 2 bits de poids faible puis additionner les bits suivants sans oublier les retenues. On commence donc par écrire la table de vérité d'un circuit additionnant 2 bits (voir section 1.4).

Tableau 2.12

a	b	S	R
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Addition de 2 bits

Dans cette table, on reconnaît la fonction OU-exclusif pour la somme et le ET pour la retenue, d'où le demi-additionneur de la figure 2.8.

Additionneur complet

Il faut maintenant être capable d'additionner 2 bits plus une retenue. Voici la table de vérité d'un tel circuit.

Tableau 2.13

a	b	r	S	R'
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

L'additionneur complet

La sortie est à 1 s'il y a un nombre impair de bits à 1 en entrée ; cela est équivalent à prendre le OU-exclusif des entrées :

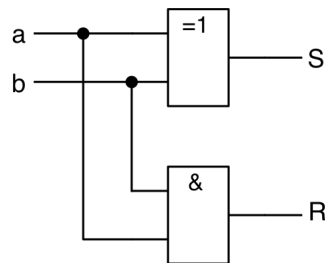
$$S = a \oplus b \oplus r$$

La retenue R' se calcule par sa forme canonique :

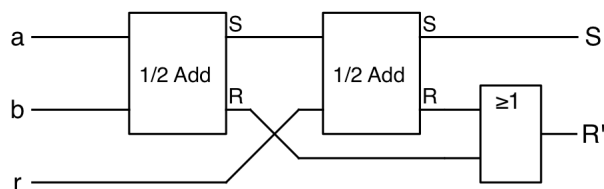
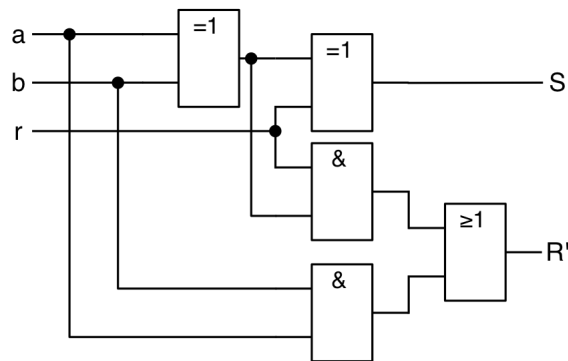
$$\begin{aligned} R' &= \bar{a}br + a\bar{b}r + ab\bar{r} + abr \\ &= r(\bar{a}b + a\bar{b}) + ab(\bar{r} + r) \\ &= r(a \oplus b) + ab \end{aligned}$$

L'intérêt de la mettre sous cette forme est d'utiliser le même circuit pour S et R' . La figure 2.8 donne les schémas classiques de l'additionneur complet 1 bit, sous forme de portes logiques et de demi-additionneur.

Figure 2.8



Demi-additionneur



Additionneur complet

Demi-additionneur et additionneur complet.

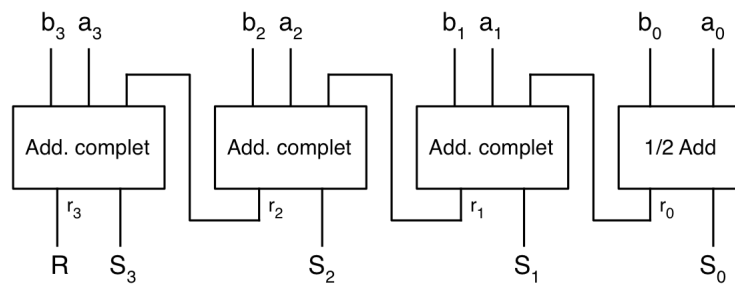
Additionneur 4 bits

Pour finir, donnons un exemple de circuit plus complexe, un additionneur 4 bits, qui fait la somme de deux nombres binaires $a_3a_2a_1a_0$ et $b_3b_2b_1b_0$.

Le construire directement à partir de la table de vérité est impossible : il y a huit entrées, soit deux cent cinquante-six combinaisons possibles, donnant chacune quatre sorties et une retenue finale. Il faut

simplement le construire en posant l'addition binaire et en effectuant la somme bit à bit. On retrouve alors les deux circuits que l'on vient de construire, le demi-additionneur pour le bit de poids faible et l'additionneur complet pour les autres. Il faut juste reporter la retenue d'un étage à l'autre comme à la figure 2.9.

Figure 2.9



Additionneur 4 bits.

Le circuit précédent a le mérite de la simplicité mais pas de la rapidité. À cause de la propagation de la retenue d'un étage à l'autre, la retenue finale n'est pas immédiatement disponible. D'autres circuits pour l'additionneur ont été proposés, dont un que vous trouverez en exercice.

Tous les circuits arithmétiques présents au sein d'un processeur sont développés de la même manière, soit directement à partir de la table de vérité, soit par regroupement de fonctions plus simples.

Complément

On ne peut pas facilement connecter entre elles plusieurs sorties de circuits logiques sous peine d'éventuels conflits si des valeurs différentes sont présentes. Pour pouvoir le faire, on dote parfois les circuits d'une ligne de commande, qui permet de désactiver les sorties en les déconnectant des entrées. On parle alors de circuits fonctionnant en **logique 3 états** : deux correspondant aux valeurs 0 et 1, et un « état indéfini » dans lequel la sortie n'est plus reliée aux entrées.

3. CIRCUITS LOGIQUES SÉQUENTIELS

Un circuit séquentiel est un circuit construit à partir de portes logiques, dans lequel on introduit une rétroaction des sorties sur les entrées. Un rebouclage permet à une ou plusieurs sorties d'être renvoyées sur les entrées des portes.

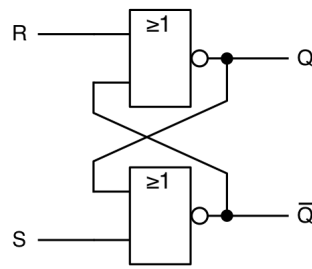
On ne peut alors plus calculer les valeurs de sortie uniquement à partir des valeurs des entrées comme pour les circuits combinatoires, puisqu'il faut également tenir compte des valeurs antérieures des sorties répercutées sur les entrées. Un nouveau paramètre temporel entre en jeu lors de l'étude du circuit.

3.1 Bascules élémentaires

Bascule RS

De même que les portes logiques constituent les briques de base des circuits combinatoires, les circuits séquentiels sont construits à partir de cellules élémentaires servant à mémoriser un bit, connues sous la dénomination générique de *bascules*, car elles possèdent deux états stables. Considérons le circuit de la figure 2.10, formé de deux portes NON-OU.

Figure 2.10



Bascule RS asynchrone.

Supposons que $S = R = 0$. Si $Q = 0$, la porte NON-OU inférieure génère 1 sur \bar{Q} , qui est renvoyé sur l'autre porte NON-OU, confirmant le 0 de la sortie Q . De même, si, $Q = 1$, on obtient $\bar{Q} = 0$, renforçant la valeur de Q . Ces deux états sont donc stables et il n'en existe pas d'autre, Q et \bar{Q} ne pouvant prendre la même valeur si S et R valent 0.

Si S passe à 1 (R restant à 0), \bar{Q} est forcé à 0 quelle que soit sa valeur précédente et Q passe à 1. Si R passe à 1 (S restant à 0), le résultat est inversé : \bar{Q} passe à 1 et Q à 0.

Si les deux entrées sont mises à 1, Q et \bar{Q} deviennent nuls. Un problème survient lorsque les deux entrées repassent à 0. Le comportement est indéterminé : suivant celle qui passe à 0 en dernier (car il y a toujours au moins un infime décalage entre les deux entrées), Q peut valoir 0 ou 1. C'est pourquoi on considère le cas $S = R = 1$ comme interdit.

Tableau 2.14

R	S	Q^*
0	0	Q (mémoire)
0	1	1 (mise à 1) – Set
1	0	0 (mise à 0) – Reset
1	1	X (interdit)

Table de vérité de la bascule RS

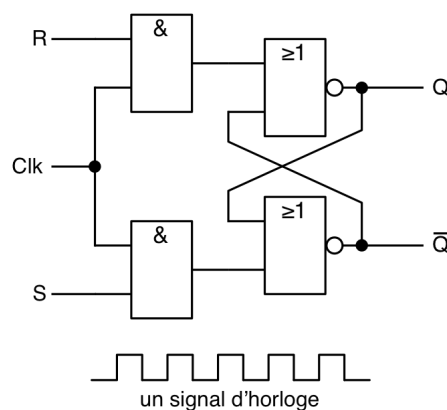
La bascule RS (pour Set et Reset) mémorise donc la dernière commande sur S ou R ; soit une mise à 1, soit une mise à 0, valeur que l'on retrouve sur Q .

La bascule présentée à la figure 2.10 est appelée « bascule RS asynchrone » car elle réagit dès que les entrées sont modifiées.

Horloge

On a souvent besoin d'effectuer des changements d'état à des instants déterminés, rythmés par une horloge, signal carré prenant régulièrement les valeurs 0 et 1 (voir à la figure 2.11 un exemple de signal d'horloge). On peut facilement modifier la bascule RS pour la rendre synchrone, c'est-à-dire ne permettre de changements que lorsque le signal d'horloge vaut 1 (voir la figure 2.11 où l'horloge est reliée à l'entrée Clk, Clock).

Figure 2.11



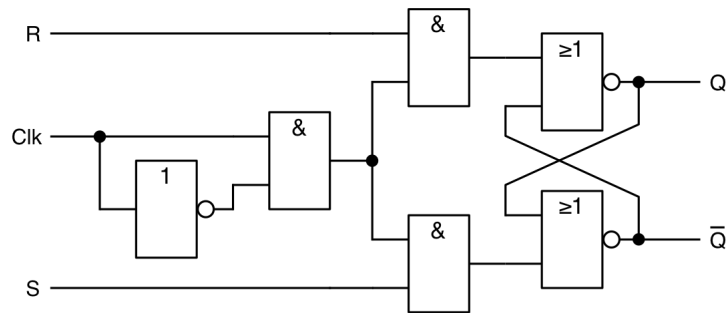
Basculer RS synchrone.

Si l'horloge vaut 0, les deux portes ET sont au repos et les deux entrées des portes NON-OU sont à 0, laissant la bascule dans son état stable, indépendamment de R et S. Si l'horloge vaut 1, les deux entrées R et S sont transférées aux entrées des NON-OU, permettant le fonctionnement habituel.

On peut bien sûr complémenter l'entrée d'horloge pour rendre son état bas (valeur 0) actif.

Le défaut de cette entrée d'horloge est de rester active pendant un long intervalle de temps, celui pendant lequel le signal est au niveau haut. Il serait plus intéressant d'avoir une activation à des instants bien précis, pour obtenir une synchronisation exacte des circuits. Pour ce faire, on peut utiliser le retard induit par le passage dans une porte logique (voir figure 2.12).

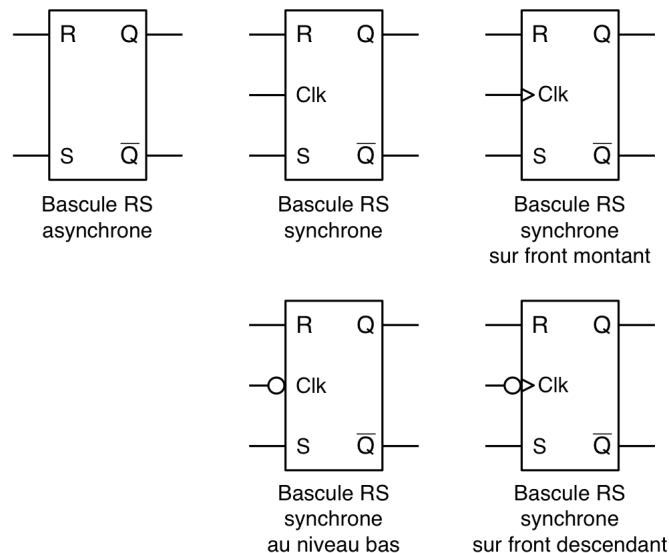
Figure 2.12



Bascule RS synchrone sur front montant.

L'entrée *Clk* étant complémentée avant la porte ET, en théorie elle restitue toujours 0 en sortie, bloquant les entrées R et S. En pratique, lorsque *Clk* passe à 1, la porte NON change d'état avec un très léger retard, laissant 1 en sortie pendant un bref instant après le changement de l'entrée *Clk*. Pendant ce court instant, la porte ET voit ses deux entrées à 1 et génère une sortie à 1, activant R et S. On dit que l'horloge est active sur son front montant (lorsqu'elle passe de 0 à 1 et uniquement à cet instant très précis). Là encore, on peut complémenter cette entrée d'horloge pour commander la bascule sur un front descendant. À la figure 2.13 se trouvent les symboles graphiques de ces bascules.

Figure 2.13



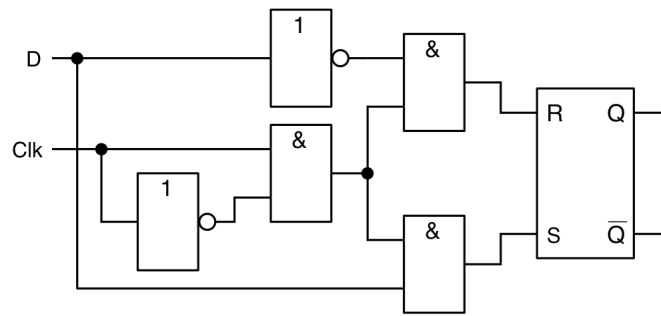
Symboles des bascules synchrones et asynchrones.

Les bascules synchronisées par des niveaux (haut ou bas) sont dites « latch » alors que les bascules déclenchées par un front sont dites « flip-flop ».

Bascule D

La bascule D (D pour *Data*) est simplement une bascule RS dans laquelle on a forcé R et S à être complémentaires. Une des deux entrées R ou S étant toujours à 1, une bascule D asynchrone ne ferait que reproduire son entrée D sur la sortie et cela n'aurait pas vraiment d'intérêt. Il est plus intéressant d'échantillonner l'entrée D à des instants bien précis. Pour cela, il est préférable que la bascule soit synchronisée sur un front (voir figure 2.14).

Figure 2.14



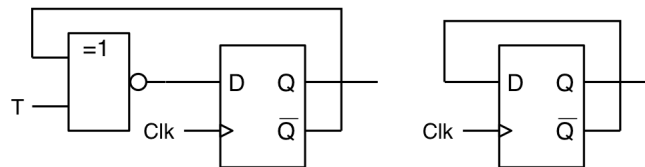
Bascule D.

Lorsque l'horloge passe de 0 à 1, l'entrée D se retrouve sur S et \bar{D} sur R . Il y a donc mémorisation de D dans la bascule et sur Q à ce moment-là. Le reste du temps, quelle que soit la valeur du signal d'horloge, la bascule est bloquée.

Bascule T

Une bascule T synchrone (T pour *Toggle* ou basculement) possède deux entrées d'activation, une entrée T et une entrée d'horloge : lorsque T vaut 0, la bascule est isolée, les sorties ne changent pas ; lorsque T est à 1, les sorties changent d'état (de 0 vers 1 et réciproquement) à chaque front montant de l'horloge. Souvent l'entrée T n'existe pas et la bascule change simplement d'état à chaque front montant (les deux constructions à partir d'une bascule D sont données à la figure 2.15).

Figure 2.15

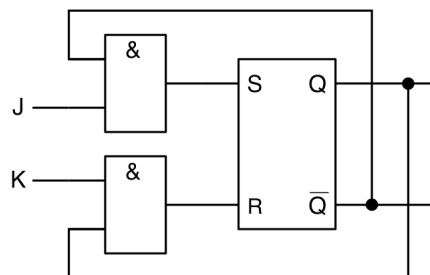


Bascules T.

Bascule JK

La bascule JK (J et K n'ont aucune signification) asynchrone est simplement une bascule RS pour laquelle la combinaison « interdite » des entrées $J = K = 1$ provoque une inversion de la sortie (voir la figure 2.16, où l'on a inversé l'emplacement des entrées de la bascule RS pour simplifier le dessin).

Figure 2.16



Bascule JK asynchrone.

Si $J = K = 0$, les deux entrées de la bascule RS sont à 0 et rien n'est modifié. Si J passe à 1, soit Q était égal à 1 et rien ne change, soit Q valait 0 et un 1 est transmis (via le 1 sur \bar{Q}) à l'entrée S forçant Q à 1. Le comportement est opposé si K vaut 1, forçant Q à 0. Dans les trois cas, la bascule est identique à une bascule RS.

Si maintenant $J = K = 1$ et si Q vaut 1, l'entrée R est activée, remettant Q à 0, et si Q vaut 0, c'est l'entrée S qui s'active, mettant Q à 1. On a alors une série de basculements de la sortie qui vaut alternativement 0 et 1, tant que $J = K = 1$.

Tableau 2.15

K	J	Q*
0	0	Q (mémoire)
0	1	1 (mise à 1)
1	0	0 (mise à 0)
1	1	\bar{Q} (basculer)

Table de vérité de la bascule JK

La bascule JK synchrone (sur front montant) se comporte comme la bascule RS synchrone sauf dans le cas $J = K = 1$ où la bascule change l'état de la sortie au moment du front.

Complément

Toutes les bascules précédentes peuvent se voir dotées d'entrées asynchrones permettant de forcer la valeur de la sortie. Ainsi il peut exister une entrée fixant la sortie principale Q à 1, souvent appelée Preset, indépendamment des valeurs des autres entrées et de l'horloge. De même, l'entrée asynchrone Clear permet de forcer la sortie Q à 0, là encore sans tenir compte ni des entrées principales ni de l'horloge. Ces entrées sont utiles pour fixer la valeur de sortie de la bascule avant utilisation.

3.2 Circuits séquentiels classiques

De même que les portes logiques servent à développer les circuits combinatoires, voici quelques exemples d'utilisation de bascules élémentaires au sein de circuits séquentiels classiques utilisés dans les ordinateurs.

Diviseur de fréquence

En créant une cascade de bascules T, on réalise un diviseur de fréquence (voir figure 2.17) : lors de chaque front montant de l'horloge, la première sortie Q change d'état ; on a donc un front montant de Q tous les deux fronts montants de l'horloge. Le phénomène se reproduit sur chacune des bascules, divisant à chaque étage la fréquence par 2 (sans que cela soit très précis à cause des retards de propagation induits par les circuits logiques des bascules).

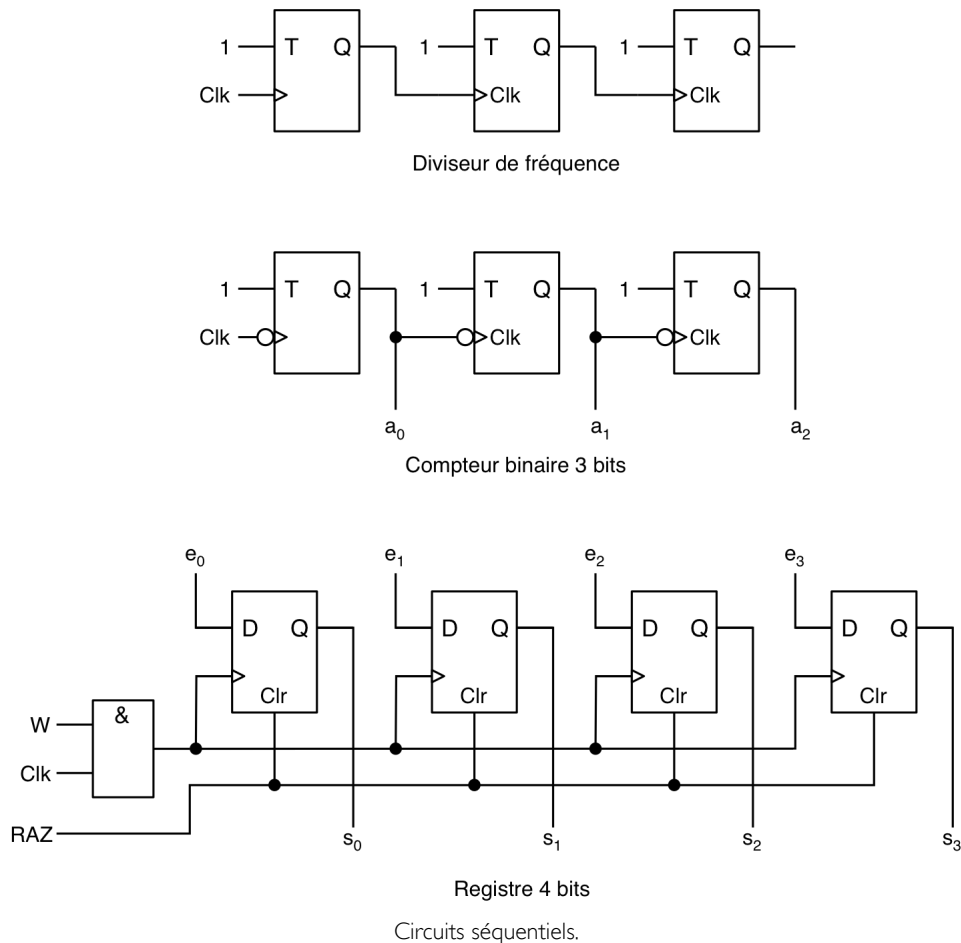
Compteur binaire

En utilisant des bascules T activées par un front descendant, on peut construire un compteur binaire (voir figure 2.17) : lorsqu'une sortie Q passe de 1 à 0, la bascule suivante change d'état (elle voit un front descendant) comme s'il y avait propagation d'une retenue. Avec n bascules, les sorties $a_0 \dots a_{n-1}$ prennent successivement les valeurs binaires de 0 à $2^n - 1$ à chaque front descendant de l'horloge.

Registre

Un registre est un élément de stockage qui permet la mémorisation de n bits en parallèle. Cet élément est constitué sur la base d'une mise en parallèle de n bascules mémorisant chacune 1 bit. La figure 2.17 montre un registre de stockage de 4 bits. Il utilise des bascules D ayant une entrée asynchrone de remise à 0 (Clr).

Figure 2.17



Si W vaut 0, l'horloge n'est jamais transmise aux bascules et celles-ci mémorisent indéfiniment la valeur initialement enregistrée. Si W vaut 1 (*Write*), au prochain front montant de l'horloge, les bascules vont stocker la valeur $e_0e_1e_2e_3$ présente sur les entrées, valeur qui restera disponible quand W repassera à 0. L'entrée RAZ sert à remettre toutes les bascules à 0 via leur entrée asynchrone.

Complément

En plus de l'algèbre de Boole, l'analyse des circuits séquentiels fait appel à la théorie des automates, qui permet de tenir compte des évolutions temporelles des circuits à travers leurs états. L'utilisation de chronogrammes, donnant l'évolution temporelle complète de tous les signaux en continu, permet l'étude fine des portes logiques (et donc des problèmes de retard et de délai de propagation), mais n'est pas très utile pour avoir une vision synthétique d'un circuit.

RÉSUMÉ

Les valeurs de tension sur les fils sont interprétées comme des bits, ce qui permet d'utiliser l'algèbre de Boole comme outil d'analyse. Celle-ci permet de définir une algèbre sur des variables pouvant prendre deux valeurs, 0 ou 1. À l'aide des fonctions élémentaires sur ces variables, NON, ET, OU et XOR, n'importe quelle fonction logique peut être définie et son expression logique explicitée. Ces mêmes fonctions élémentaires existent physiquement sous la forme de portes logiques, qui modifient les caractéristiques électriques des signaux et donc des bits, permettant ainsi, toujours grâce à l'algèbre de Boole, d'implémenter des fonctions logiques de base, décodage, addition, etc., et de construire des circuits plus complexes.

Les circuits logiques séquentiels utilisent la possibilité de réinjecter des sorties sur les entrées pour mémoriser des bits dans les portes. On peut ainsi construire des circuits, appelés « registres », mémorisant des informations.

Problèmes et exercices

Les exercices suivants permettent de construire des circuits logiques soit à partir d'expressions booléennes, soit directement à partir de circuits classiques existants. Vous allez ainsi étudier des opérations arithmétiques telles que la soustraction, la multiplication ou la comparaison de nombres sous l'angle logique pour écrire leur table de vérité et dessiner le circuit équivalent à base de portes logiques. L'un des exercices fait intervenir le délai de propagation à travers une porte, ce qui oblige à concevoir différemment le circuit pour ne pas être pénalisé d'un point de vue des performances.

Exercice I : Construire une fonction logique

Soit un entier de 0 à 7 représenté par 3 bits $b_2b_1b_0$. Soit F la fonction logique dont les entrées sont ces 3 bits et qui prend la valeur 1 si la valeur de l'entrée vaut 0, 3, 4 ou 7 (codée en binaire), et 0 sinon.

- 1) Donnez la table de vérité de F , écrivez-la sous sa forme canonique et simplifiez l'expression obtenue.
- 2) Retrouvez le résultat précédent en écrivant la table de Karnaugh de F .
- 3) Dessinez le circuit logique calculant F , uniquement à l'aide de portes NON-ET.

Tableau 2.16

b_2	b_1	b_0	F
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

La table de vérité de F

On obtient alors la forme canonique de F et sa simplification :

$$\begin{aligned} F &= \bar{b}_2 \bar{b}_1 \bar{b}_0 + \bar{b}_2 b_1 b_0 + b_2 \bar{b}_1 \bar{b}_0 + b_2 b_1 b_0 \\ &= (\bar{b}_2 + b_2)(\bar{b}_1 \bar{b}_0) + (\bar{b}_2 + b_2)(b_1 b_0) \\ &= \bar{b}_1 \bar{b}_0 + b_1 b_0 \\ &= \overline{b_1 \oplus b_0} \end{aligned}$$

La table de Karnaugh de F est donnée à la figure 2.18.

Figure 2.18

b_1b_0	00	01	11	10
b_2				
0	1	0	1	0
1	1	0	1	0

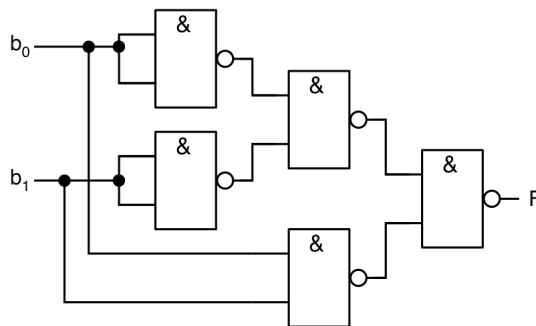
Table de Karnaugh de F .

En effectuant les deux regroupements, on retrouve bien le résultat $F = \bar{b}_1\bar{b}_0 + b_1b_0$.

La loi de De Morgan permet de transformer le OU en ET ; on obtient la formule et le circuit correspondants (voir figure 2.19).

$$F = \overline{\bar{b}_1\bar{b}_0 \cdot b_1b_0}$$

Figure 2.19



Circuit logique équivalent à F .

Exercice 2 : Un circuit qui calcule

Soit une machine qui travaille sur des nombres binaires signés de 4 bits. La valeur signée d'un mot $A = a_3a_2a_1a_0$ est égale à $a_0 + 2a_1 + 4a_2 - 8a_3$ (c'est la représentation classique en complément à 2). On désire construire un circuit qui donne en sortie $B = b_3b_2b_1b_0 = -2 \times A$ (par exemple, si l'on a la valeur 2 en entrée, on veut la valeur binaire -4 en sortie).

- 1) Toutes les valeurs binaires sont-elles autorisées en entrée ? Autrement dit, le circuit donne-t-il toujours une valeur correcte ? Donnez la table de vérité du circuit pour toutes les valeurs autorisées.
- 2) Donnez les expressions logiques des 4 bits de sortie en fonction des 4 bits d'entrée en ne tenant pas compte des valeurs interdites.
- 3) On désire avoir un bit de débordement (voir chapitre 1) qui indiquerait qu'une valeur interdite est présente sur les entrées. Écrivez la table de Karnaugh de ce bit et donnez-en une expression logique.

1) Seules les valeurs de -3 à 4 sont autorisées en entrée car, au-delà, on ne peut pas représenter le résultat sur 4 bits signés, comme c'est le cas, par exemple, pour $-2 \times 4 = 8$ et $-2 \times 5 = -10$. On peut donc écrire une table de vérité tronquée en ne mettant pas les valeurs interdites :

Tableau 2.17

A	$a_3a_2a_1a_0$	$b_3b_2b_1b_0$	B
0	0000	0000	0
1	0001	1110	-2
2	0010	1100	-4
3	0011	1010	-6

4	0100	1000	-8
-3	1101	0110	6
-2	1110	0100	4
-1	1111	0010	2

La table de vérité

2) On a $b_0 = 0$ et $b_1 = a_0$. En écrivant la forme canonique de b_2 , on obtient :

$$\begin{aligned}
 b_2 &= \bar{a}_3 \bar{a}_2 \bar{a}_1 a_0 + \bar{a}_3 \bar{a}_2 a_1 \bar{a}_0 + a_3 a_2 \bar{a}_1 a_0 + a_3 a_2 a_1 \bar{a}_0 \\
 &= \bar{a}_3 \bar{a}_2 (a_1 \oplus a_0) + a_3 a_2 (a_1 \oplus a_0) \\
 &= (\bar{a}_3 \oplus a_3) (\bar{a}_2 \oplus a_2) (a_1 \oplus a_0) \\
 &= (a_3 \oplus a_2) (a_1 \oplus a_0)
 \end{aligned}$$

On peut en fait simplifier en :

$$b_2 = (a_1 \oplus a_0)$$

en ne tenant pas compte des cas non autorisés (puisque les sorties peuvent alors prendre les valeurs que l'on veut). On voit également que le signe s'inverse dans tous les cas sauf pour 0. b_3 est donc l'inverse de a_3 sous réserve qu'au moins un des autres bits soit à 1 :

$$b_3 = \bar{a}_3 (a_2 + a_1 + a_0)$$

3) La table de Karnaugh du bit de débordement est présentée à la figure 2.20.

Figure 2.20

		$a_1 a_0$			
		00	01	11	10
$a_3 a_2$	00	0	0	0	0
	01	0	1	1	1
	11	1	0	0	0
	10	1	1	1	1

Table du bit de débordement.

Les regroupements indiqués donnent l'expression suivante :

$$e = a_3 \bar{a}_2 + a_3 a_2 \bar{a}_1 \bar{a}_0 + \bar{a}_3 a_2 \bar{a}_1 a_0 + \bar{a}_3 a_2 a_1$$

On a donc réussi à obtenir l'expression algébrique des différentes sorties du circuit, ce qui permet de le construire, ainsi que l'expression du bit de débordement, indiquant que le circuit ne peut pas donner la valeur correcte.

Exercice 3 : Additionneur/soustracteur

On construit un circuit qui a deux entrées (a et b) et deux sorties (s et r) et une ligne de commande F tel que :

- Si $F = 0$, le circuit effectue une addition ($a + b$ avec une sortie s et une retenue r).
- Si $F = 1$, le circuit effectue une soustraction ($a - b$ avec une sortie s et une retenue r).

1) Donnez la table de vérité de ce circuit demi-additionneur/soustracteur. Montrez que s se calcule facilement avec une porte et r avec deux.

2) On construit maintenant un additionneur/soustracteur complet, c'est-à-dire un circuit à trois entrées (a , b et r), deux sorties (s et r') et une ligne de commande F tel que :

- Si $F = 0$, le circuit effectue une addition ($a + b + r$ avec une sortie s et une retenue r').
- Si $F = 1$, le circuit effectue une soustraction ($a - (b + r)$ avec une sortie s et une retenue r').

Écrivez la table de vérité de s pour la soustraction et comparez-la à celle de l'additionneur complet. Donnez l'expression logique de s . Écrivez la table de Karnaugh ainsi qu'une expression logique de r' .

3) À partir des deux circuits précédents, proposez une construction pour un additionneur/soustracteur sur n bits, c'est-à-dire un circuit avec deux fois n bits en entrée et qui effectue l'addition ou la soustraction de ces deux nombres suivant la valeur d'une ligne de commande.

1) Voici la table de vérité de l'addition et la soustraction de 2 bits.

Tableau 2.18

a	b	F	s	r
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	0	0
0	1	1	1	1
1	0	1	1	0
1	1	1	0	0

Le demi-additionneur/soustracteur

On obtient les deux expressions logiques :

$$s = a \oplus b$$

$$r = ab\bar{F} + \bar{a}bF = b(a\bar{F} + \bar{a}F) = b(a \oplus F)$$

2) La table de vérité de s , dans le cas de la soustraction de 3 bits, est identique à celle de l'additionneur complet. On obtient donc $s = a \oplus b \oplus r$. La table de Karnaugh de r' se trouve à la figure 2.21 (avec le dessin du circuit complet) et permet d'obtenir l'expression suivante :

$$\begin{aligned}
 r' &= br + \bar{a}\bar{b}\bar{F}r + \bar{a}b\bar{F}\bar{r} + ab\bar{F}\bar{r} + \bar{a}bF\bar{r} \\
 &= br + (b\bar{r} + \bar{b}r)(\bar{a}F + a\bar{F}) \\
 &= br + (b \oplus r)(a \oplus F)
 \end{aligned}$$

Figure 2.21

Fr \ ba				
	00	01	11	10
00	0	0	1	0
01	0	1	1	1
11	1	0	1	1
10	0	0	0	1

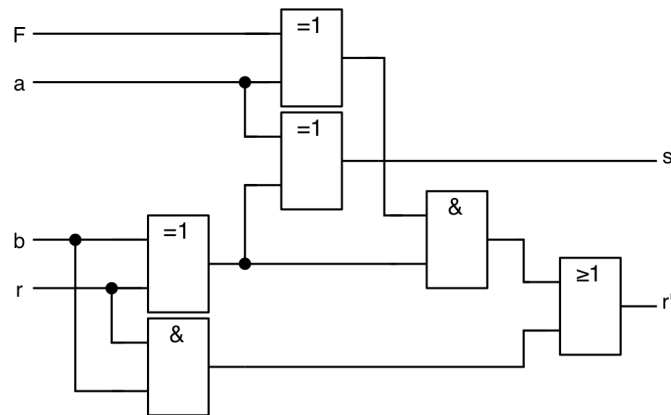
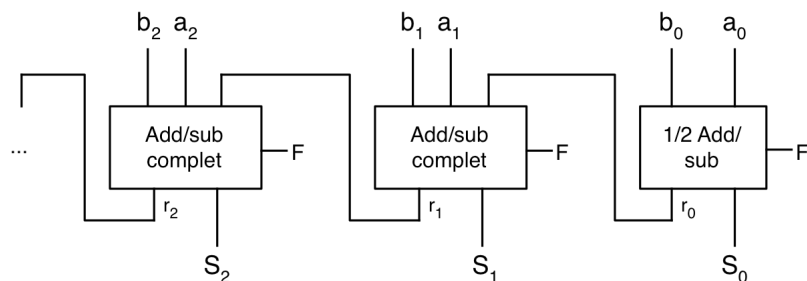


Table de r' et circuit complet.

3) On construit l'additionneur/soustracteur n bits exactement comme l'additionneur n bits, en reportant les retenues d'un circuit élémentaire à l'autre (voir figure 2.22).

Figure 2.22



Additionneur/soustracteur n bits.

On a donc réussi à construire un circuit commandé qui effectue l'addition ou la soustraction de deux nombres de n bits. Ce circuit pourrait servir à l'intérieur d'un ordinateur pour réaliser les opérations mathématiques de base.

Exercice 4 : Multiplicateur

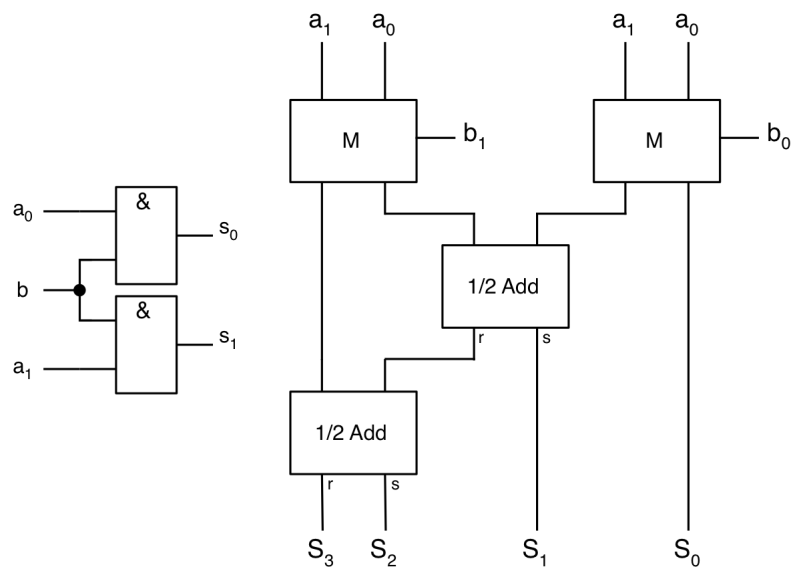
- 1) On souhaite construire un multiplicateur 2 bits par 1 bit (représentant des nombres entiers positifs), c'est-à-dire un circuit à trois entrées a_0 , a_1 et b , et deux sorties s_0 et s_1 , tel que si $b = 0$, $s_0 = s_1 = 0$ et si $b = 1$, $s_0 = a_0$ et $s_1 = a_1$. Donnez le schéma d'un tel circuit (appelé M) en utilisant deux portes ET.
- 2) On veut maintenant construire un multiplicateur 2 bits par 2 bits. En décomposant cette multiplication en multiplications plus simples et en additions, montrez que l'on peut le construire avec deux circuits M et deux demi-additionneurs.
- 3) Donnez le schéma d'un multiplicateur 2 bits par n bits utilisant des circuits M, demi-additionneurs et additionneurs complets.

- 1) Il suffit de placer b à l'une des entrées des deux portes ET pour construire le circuit M (voir figure 2.23).
- 2) Pour effectuer une multiplication 2 bits par 2 bits, on la pose en multipliant bit par bit :

$$\begin{array}{r}
 \begin{array}{cc}
 a_1 & a_0 \\
 \hline
 b_1 & b_0 \\
 \hline
 a_1 b_0 & a_0 b_0 \\
 a_1 b_1 & a_0 b_1 \\
 \hline
 s_3 & s_2 & s_1 & s_0
 \end{array}
 \end{array}$$

Cela donne le circuit de la figure 2.23.

Figure 2.23

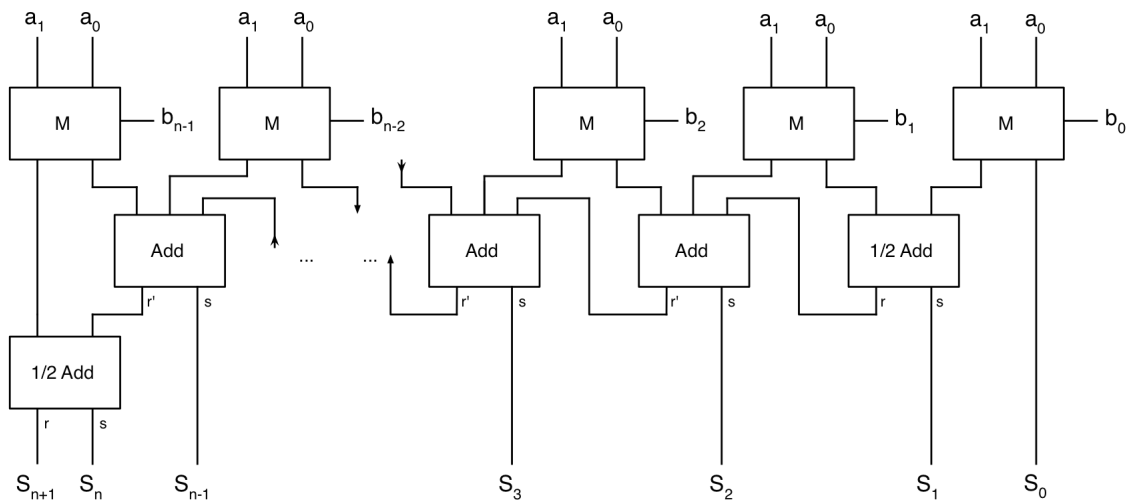


Multiplicateurs 2 par 1 et 2 par 2.

- 3) La multiplication 2 bits par n bits se pose également bit par bit, ce qui permet, en prenant chacune des multiplications et des additions élémentaires, de construire le circuit correspondant (voir figure 2.24) avec n circuits M, deux demi-additionneurs et $n - 2$ additionneurs complets.

$$\begin{array}{r}
 \begin{array}{ccccccc}
 & & & & a_1 & a_0 \\
 & & & & b_1 & b_0 \\
 \hline
 b_{n+1} & \dots & \dots & & a_1 b_0 & a_0 b_0 \\
 & & & & a_1 b_1 & a_0 b_1 \\
 & & & & a_0 b_2 & \\
 & & & & \dots & \\
 & & & & a_1 b_{n+2} & \\
 & & & & a_1 b_{n+1} & a_0 b_{n+1} \\
 \hline
 s_{n+1} & s_n & s_{n+1} & \dots & s_2 & s_1 & s_0
 \end{array}
 \end{array}$$

Figure 2.24



Multiplicateur 2 bits par n bits.

Là encore, on a construit un circuit capable d'implémenter une opération que l'ordinateur aura peut-être à effectuer.

Exercice 5 : Additionneur à retenue anticipée

- 1) On suppose que le passage d'un signal électrique dans une porte « coûte » 10 nanosecondes (c'est un ordre de grandeur réaliste compte tenu des technologies actuelles). On dit alors qu'un circuit travaille en p ns si tous les signaux en sortie sont disponibles en un maximum de p ns. En combien de temps un demi-additionneur, un additionneur complet et un additionneur 4 bits travaillent-ils ?
- 2) On va construire un additionneur 4 bits à retenue anticipée, c'est-à-dire un circuit où le calcul des retenues intermédiaires se fait plus rapidement. Soit une fonction de génération de retenue $G = ab$ et une fonction de propagation de retenue $P = a + b$. Montrez que la retenue de sortie d'un additionneur complet peut se calculer par la formule $r' = Pr + G$, où a et b sont les entrées et r la retenue d'entrée.
- 3) Dans l'additionneur 4 bits, on note $G_i = a_i b_i$ et $P_i = a_i + b_i$, et r_i la retenue intermédiaire d'un étage, normalement calculée à partir des entrées a_i, b_i et de la retenue précédente r_{i-1} . Exprimez r_0, r_1, r_2 et $r_3 = r$ en fonction de $G_0, G_1, G_2, G_3, P_0, P_1, P_2$ et P_3 .
- 4) Supposons que l'on dispose de plusieurs portes OU et ET à deux, trois ou quatre entrées, et que chacune travaille en 10 ns. En combien de temps se fait le calcul des G_i, P_i et r_i ? Est-ce intéressant ? Quelle différence y a-t-il entre le temps de travail d'un additionneur 8 bits normal et d'un additionneur 8 bits à retenue anticipée (avec les bonnes portes OU et ET) ?

1) Le demi-additionneur travaille en 10 ns car chaque sortie passe par une porte (un XOR pour la somme et un ET pour la retenue). Un additionneur complet travaille en 30 ns pour r' et 20 ns pour s , soit 30 ns au total (mais il faut différencier les sorties pour la question suivante).

Pour l'additionneur 4 bits, il ne faut pas se contenter d'additionner les valeurs des circuits le composant mais détailler au niveau de chaque porte, car les entrées ne sont pas disponibles en même temps sur chaque circuit. On a les chiffres suivants :

s_0 : 10 ns	s_1 : 20 ns	s_2 : 40 ns	s_3 : 60 ns
r_0 : 10 ns	r_1 : 30 ns	r_2 : 50 ns	r_3 : 70 ns

Soit 70 ns au total pour l'additionneur 4 bits.

2) Le calcul de la retenue de sortie de l'additionneur complet est le suivant :

$$r = ab + ar + br + abr = (a + b)r + ab(r + 1) = Pr + G$$

3) De même, les différentes retenues se calculent de la façon suivante :

$$r_0 = a_0 b_0 = G_0$$

$$r_1 = G_1 + r_0 P_1 = G_1 + G_0 P_1$$

$$r_2 = G_2 + r_1 P_2 = G_2 + (G_1 + G_0 P_1) P_2 = G_2 + G_1 P_2 + G_0 P_1 P_2$$

$$r_3 = G_3 + r_2 P_3 = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3$$

4) $G_i = a_i b_i$ et $P_i = a_i + b_i$, donc chacun est disponible en 10 ns (passage par une porte). Les r_i sont formés de OU à partir de ET sur les G_i et P_i , et se calculent donc tous en 30 ns. Donc s_0 se calcule en 10 ns, s_1 en 20 ns, s_2 en 40 ns et s_3 en 40 ns également. C'est plus rapide comparativement à l'additionneur 4 bits standard.

Pour un additionneur 8 bits, le temps normal est de 150 ns. Avec un additionneur à retenue anticipée, le calcul se fait toujours en 40 ns, mais il faut des portes ET et OU à huit entrées.

Exercice 6 : Comparateur de nombres

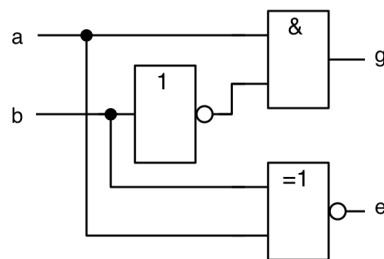
On veut concevoir un circuit permettant de comparer deux nombres A et B de 4 bits (valant donc de 0 à 15), $A = a_3 a_2 a_1 a_0$ et $B = b_3 b_2 b_1 b_0$. Le circuit a deux sorties : G valant 1 si $A > B$ et E valant 1 si $A = B$.

1) Soit a et b deux nombres de 1 bit. Soit un circuit à deux sorties : g valant 1 si $a > b$ et e valant 1 si $a = b$. Donnez les expressions logiques de g et e et dessinez le circuit correspondant.

2) En utilisant des circuits de la question précédente ainsi que des portes logiques OU, ET (à deux, trois ou quatre entrées) et NON, concevez un circuit permettant de comparer deux nombres de 4 bits.

1) On dénombre un seul cas où a est plus grand que b : quand a vaut 1 et b vaut 0. On a donc $g = a\bar{b}$. L'égalité des 2 bits se calcule par $e = ab + \bar{a}\bar{b} = a \oplus b$, ce qui donne le circuit de la figure 2.25.

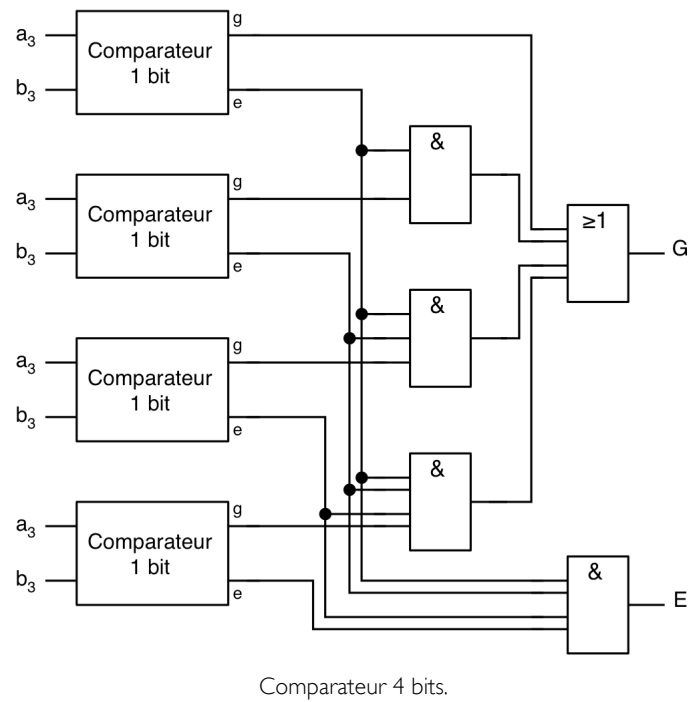
Figure 2.25



Comparateur 1 bit.

2) Pour que les deux nombres de 4 bits soient égaux, il faut qu'il y ait égalité deux par deux des bits composant les nombres. Pour savoir si A est plus grand que B , il faut comparer les bits en partant du bit de poids fort : soit a_3 est plus grand que b_3 , soit a_3 est égal à b_3 et a_2 est plus grand que b_2 , soit on a aussi a_2 égal à b_2 et a_1 est plus grand que b_1 , soit a_1 aussi est égal à b_1 et a_0 est plus grand que b_0 . Cela donne le circuit de la figure 2.26.

Figure 2.26

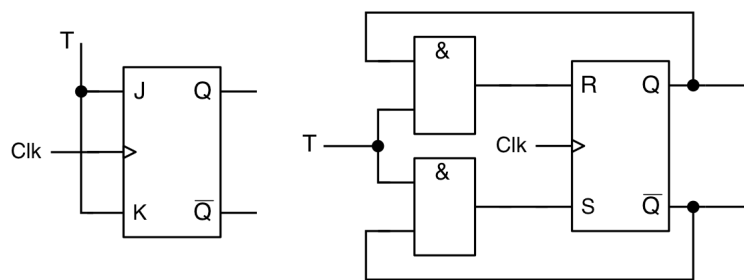


Exercice 7 : Bascules équivalentes

Comment peut-on utiliser une bascule JK pour construire une bascule T, et une bascule RS pour construire une bascule T ?

La bascule T change d'état à chaque front d'horloge quand T vaut 1. C'est aussi le comportement de la bascule JK quand les deux entrées sont à 1. Il suffit donc de relier les deux entrées de la bascule JK entre elles. Quant à la bascule RS, il faut la mettre à 0 quand la sortie est à 1 (donc relier Q à R) et la mettre à 1 quand elle est à 0 (donc relier \bar{Q} à S) pour avoir le même phénomène de bascule d'état. Il suffit de commander ces deux liens par deux portes ET pour avoir la commande sur T (voir les deux circuits à la figure 2.27).

Figure 2.27

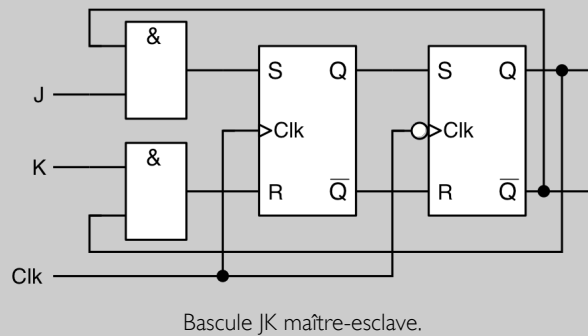


Basculs T avec JK et RS.

Exercice 8 : Bascule maître-esclave

Décrivez le fonctionnement du circuit de la figure 2.28 (attention, les commandes se font à fronts inversés sur les deux bascules). Quelle peut être son utilité ?

Figure 2.28



Le circuit ressemble à une bascule JK avec le retour des sorties sur la commande des entrées. Cependant, la première bascule est commandée par un front montant alors que la seconde par un front descendant. Cela permet de découpler temporellement les entrées et sorties : une fluctuation des valeurs d'entrée au moment du front montant provoquera une fluctuation des sorties de la première bascule mais elle n'aura pas d'influence sur la seconde, qui ne sera activée que sur le front descendant. De même, une fluctuation lors du front descendant ne sera pas prise en compte par la première bascule, inactivée.

Une autre utilisation est envisageable dans le cadre d'un couplage de bascules commandées par une même horloge. Si toutes les bascules sont reliées à une même horloge, à cause du délai de propagation au passage des portes, il est probable que les sorties des premières bascules arriveront trop tard sur les entrées des suivantes ou provoqueront ces fameuses fluctuations. Cette bascule JK maître-esclave (la première bascule RS est maître, la seconde esclave) permet d'éviter ces problèmes en ne commandant pas l'activation des entrées en même temps que la validation des sorties.

Chapitre 3

Ordinateur et processeur

L'architecture d'un ordinateur est fondée sur le modèle de von Neumann : la machine est construite autour d'un processeur, véritable tour de contrôle interne, d'une mémoire stockant les données et le programme, et d'un dispositif d'entrées/sorties nécessaire pour l'échange avec l'extérieur.

Le processeur exécute une par une les instructions stockées sous forme numérique en mémoire, écrites par le programmeur ou le compilateur, en utilisant ses éléments internes : séquenceur, registres et unité arithmétique et logique. Vous allez découvrir dans ce chapitre comment sont constitués les processeurs, de quelle manière ils exécutent les instructions et quelles évolutions ont permis d'améliorer leurs performances.

Ordinateur et processeur

1. Architecture de von Neumann	52
2. Les instructions.....	59
3. UAL et registres	65
4. Séquenceur	68
5. Architectures évoluées	70

Problèmes et exercices

1. Compilation ou interprétation	78
2. Débit d'informations.....	78
3. Décalages et multiplication.....	79
4. Sauts et contrôle	79
5. Instructions à une, deux ou trois données	80
6. Divers modes d'adressage.....	82
7. Registre d'état et comparaison	83
8. Pile et paramètre de fonction.....	83
9. Pipeline et branchements.....	84
10. Exécution multithread.....	87

I. ARCHITECTURE DE VON NEUMANN

L'ordinateur est né pendant la Seconde Guerre mondiale des travaux de plusieurs ingénieurs et théoriciens. Il n'y a pas eu un pôle unique de développement mais plusieurs centres indépendants qui ont chacun tâtonné en essayant de construire des machines semblables à aucune autre existant à l'époque.

Avant-guerre, Alan Turing (Angleterre) et Claude Shannon (États-Unis) avaient posé les fondements de la théorie du calcul, ouvrant la voie aux réalisations : Konrad Zuse (Allemagne), John Atanasoff (États-Unis), Alan Turing et son équipe (Angleterre) ont développé entre 1940 et 1945 les premiers calculateurs. À base de relais électriques ou de tubes électroniques, programmables ou non, ces prototypes ont montré la faisabilité du concept d'ordinateur. Souvent isolés, ces inventeurs n'ont pas été soutenus mais ont permis la construction après-guerre des premiers ordinateurs, lorsque les ingénieurs ont de nouveau pu s'y intéresser.

Même si des réalisations antérieures, plus ou moins semblables, ont existé, on attribue le titre de premier ordinateur (de manière quand même un peu arbitraire) à l'ENIAC (*Electronic Numerical Integrator and Computer*), œuvre de John W. Mauchly et de Prosper Eckert au sein de l'université de Pennsylvanie (États-Unis), finalisé en 1946.

En raison de la très large publicité qui a été faite, l'ENIAC a attiré de nombreux visiteurs qui s'emploieront, de retour dans leurs laboratoires, à développer leurs propres machines, faisant ainsi avancer les connaissances.

L'un de ces visiteurs les plus connus a été le mathématicien John von Neumann, qui a rédigé à la suite un rapport où il exposait ses vues sur l'organisation interne d'un ordinateur. Il a mis, entre autres, l'accent sur le concept de programme enregistré. L'architecture décrite dans son rapport est à la base des ordinateurs depuis 65 ans.

I.1 Typologie historique

Jusque dans les années 80, il était assez facile de caractériser les ordinateurs en fonction de leur taille. Les ordinateurs centraux, descendant des gros systèmes des années 60, occupaient une pièce entière et plusieurs techniciens étaient nécessaires pour assurer leur bon fonctionnement. Ces machines complexes, utilisées par les services de gestion, traitaient souvent de grosses bases de données : service du personnel, paie, logistique, stocks, etc.

Le progrès technologique avait fait de ces ordinateurs des machines coûteuses mais puissantes, parfois même trop puissantes pour la tâche voulue. En réaction, des ingénieurs ont conçu à la fin des années 60 le mini-ordinateur. Beaucoup plus petit, moins puissant, mais plus simple et assez bon marché, il avait sa place dans un laboratoire ou un petit service, où il pouvait être utilisé par quelques utilisateurs à la fois, pour des travaux peu complexes, comme l'exécution de simples calculs mathématiques.

Le même phénomène s'est reproduit dans les années 70 quand la disponibilité et la baisse des coûts des composants ont permis à quelques pionniers de construire des petites cartes électroniques à base de microprocesseurs.

On ne voyait pas très bien à quoi ces micro-ordinateurs (« micro » est alors un terme péjoratif), peu puissants, sans affichage élaboré, sans logiciel, pouvaient servir. Il a fallu attendre le développement des premières applications, traitement de texte et tableur, pour s'apercevoir que ces machines convenaient bien à un usage individuel.

Les années 80-90 voient la montée en puissance de ces micro-ordinateurs. Aidés par l'immensité du marché potentiel, ils finissent par faire disparaître les mini-ordinateurs et même par marcher sur les terres des gros systèmes.

À l'heure actuelle, on peut grossièrement découper l'univers informatique en quatre catégories :

- Les micro-ordinateurs, représentant l'immense majorité du contingent, sont adaptés à un usage individuel. Leurs performances sont sans cesse améliorées par le progrès technologique, soutenu par des entreprises de taille mondiale (Intel, AMD, IBM, HP, Apple, Dell, Microsoft...).
- Les serveurs, aussi appelés « stations de travail », sont souvent de simples micro-ordinateurs, gonflés par l'ajout de mémoire, de disques accélérés, de cartes réseau plus rapides, de processeurs supplémentaires, etc. Ils sont pilotés par plusieurs utilisateurs simultanément et permettent un affichage graphique élaboré ou des accès rapides.
- Les gros systèmes sont toujours présents, mais relégués à de grosses applications de gestion dans des services de taille importante.

- Enfin depuis toujours, il existe de superordinateurs, capables de prouesses mathématiques, au prix d'un surcoût important.

Les micro-ordinateurs s'améliorent continuellement et restreignent les autres catégories à la portion congrue. Les serveurs sont fabriqués à partir de micro-ordinateurs et ces derniers, lorsqu'ils sont connectés par milliers, permettent de construire certains superordinateurs, à un prix nettement inférieur à celui des « vrais » supercalculateurs. Cependant, de nouveaux usages apparaissent ; le micro-ordinateur est parfois « trop puissant » et l'avancée technologique permet de trouver des débouchés pour des appareils plus petits : *smartphones* et *netbooks* par exemple.

I.2 Programme enregistré

On programait les premiers ordinateurs en reliant physiquement par des câbles leurs différents composants et en positionnant manuellement des commutateurs sur les éléments fonctionnels. Suivant la nature et l'ordre de ces liaisons, la machine effectuait les opérations voulues par l'utilisateur, opérations dont la séquence était également contrôlée par le jeu de ces commutateurs. L'inconvénient majeur de cette technique était bien sûr que le passage à une autre tâche faisait perdre l'ensemble des connexions installées, qui étaient remplacées par celles nécessaires à l'exécution des nouveaux calculs. Si l'on voulait plus tard revenir à la première application, il fallait recâbler l'ensemble de la machine à la main avec le risque de se tromper à chaque fois.

Après sa visite de l'ENIAC, von Neumann a été le premier (même si l'idée apparaissait déjà dans les travaux théoriques de C. Babbage au XIX^e siècle) à décrire la notion de programme enregistré. Il ne fallait plus voir la suite des opérations à effectuer sur la machine comme une composante matérielle de celle-ci mais comme une série d'instructions à lui donner. Ces instructions devaient se mettre sous forme de codes numériques que l'on allait pouvoir laisser dans la mémoire de l'ordinateur où le processeur irait les chercher pour les traiter.

L'avantage de cette technique était de pouvoir stocker, lorsque le travail est fini, cette suite d'instructions sur un support adéquat, cartes perforées à l'époque, disque dur maintenant, pour la réutiliser sans avoir à tout réécrire.

Un autre intérêt était de pouvoir écrire des programmes automodifiants. Comme une suite d'instructions composant un programme était traduite sous forme de codes numériques et stockée en mémoire, telles des données, pour être exécutée, il était tout à fait envisageable d'avoir des instructions modifiant les données en mémoire mais également les propres instructions du programme. Cette technique permettait de simuler des instructions non disponibles et d'économiser la place occupée en mémoire (ressource rare au début de l'informatique) en adaptant lesdites instructions au cours de l'exécution du programme.

De nos jours, les contraintes à l'origine de cette technique n'existant plus, les programmes automodifiants ne sont plus de mise et sont même considérés comme une mauvaise pratique. En effet, ils compliquent fortement le débogage (les instructions prévues sur papier sont modifiées en cours d'exécution et on a du mal à prévoir ce qui va se passer) et empêchent d'avoir les mêmes instructions exécutées par deux programmes en même temps (car chacun essaie de les modifier). De plus, des techniques récentes d'amélioration des performances telles que le préchargement des instructions (section 5) ou la mémoire cache (chapitre 6) sont difficilement compatibles avec les programmes automodifiants.

I.3 Cycle d'un programme

Assembleur

Les premiers programmes étaient écrits directement en code machine, c'est-à-dire que les codes numériques correspondant aux instructions étaient entrés un par un en mémoire par l'utilisateur. Très rapidement, on est passé à la programmation en langage assembleur, dans laquelle des expressions symboliques, appelées « mnémoniques », remplaçaient les codes numériques. Ainsi, au lieu de saisir le code $5ef4_{16}$ en hexadécimal (ou son équivalent binaire), on écrivait `ADD A,B`. Cette dernière formulation était beaucoup plus compréhensible pour le programmeur, qui voyait tout de suite qu'il s'agissait d'une addition entre deux nombres stockés quelque part. Cependant, l'ordinateur ne savait toujours travailler qu'avec des codes numériques et, même si ceux-ci avaient chacun un équivalent symbolique, il fallait les transformer pour qu'ils puissent être exécutés. C'était le rôle du programme d'assemblage, qui prenait comme entrée un autre fichier décrivant un programme à l'aide de mnémoniques et qui générait le code numérique équivalent (ou code binaire), permettant ainsi à l'ordinateur d'exécuter les instructions (voir figure 3.1).

Note

La programmation à l'aide des mnémoniques s'appelle parfois « programmation en langage d'assemblage », et plus couramment « programmation en assembleur ». L'ensemble des codes numériques s'appelle classiquement « langage machine », même si pour certains, celui-ci fait par extension également référence au langage assembleur (par opposition aux langages évolués comme C, C++, Pascal, Java...).

La programmation en assembleur a quand même de sérieux inconvénients. D'abord, il est fastidieux d'écrire des lignes de code car, les instructions étant souvent peu puissantes, il faut en aligner beaucoup pour obtenir un résultat, même simple. Ensuite, les instructions sont spécifiques à un modèle de processeur et tout changement de machine implique une réécriture plus ou moins complète du code.

Compilateur

Pour pallier ces défauts, des langages évolués (par exemple C, C++, Java, Pascal, Ada et leurs nombreuses variantes) ont été développés et, de nos jours, il est rare d'avoir à programmer directement en assembleur. Qu'ils soient orientés objet, déclaratifs ou encore impératifs, tous les langages évolués disposent de structures de données et de contrôle évoluées permettant aux développeurs de se concentrer sur l'algorithmique des applications, plutôt que sur l'écriture fastidieuse du code assembleur.

Le prix à payer est bien sûr que ces instructions ne sont plus directement compréhensibles par l'ordinateur. Une phase de traduction en langage machine pour obtenir un programme exécutable est nécessaire. C'est le rôle du compilateur, un programme qui prend en entrée un fichier texte contenant le code source écrit en langage évolué pour produire un fichier exécutable formé de codes numériques propres à la machine. Au passage, on a gagné en portabilité. Les langages évolués sont relativement normalisés et le code source d'un programme peut facilement être « porté » sur un autre ordinateur, à la seule condition d'avoir le compilateur prévu pour ce langage, générant des instructions en langage machine adaptées à la nouvelle machine.

Interpréteur

L'étape de compilation, nécessaire lors de chaque modification du programme, peut prendre un certain temps, d'autant plus grand que la taille du code est importante. D'autres langages (comme PHP, Perl ou Basic) proposent une autre méthode, qui consiste, non pas à traduire les instructions, mais à les exécuter au fur et à mesure via un interpréteur. Pour chaque ligne de code, ce dernier « simule » son exécution sur l'ordinateur. Il n'y a plus de délai de compilation, celle-ci n'existant plus, mais une exécution directe du programme, qui, malheureusement, tourne beaucoup plus lentement que s'il était compilé. En effet, l'interpréteur doit simuler le fonctionnement des instructions au lieu de laisser l'ordinateur le faire directement (voir figure 3.1). Pour cette raison, les langages interprétés sont plutôt réservés à l'écriture de petits programmes pour lesquels la facilité de développement (on modifie, on exécute), prime sur la rapidité d'exécution.

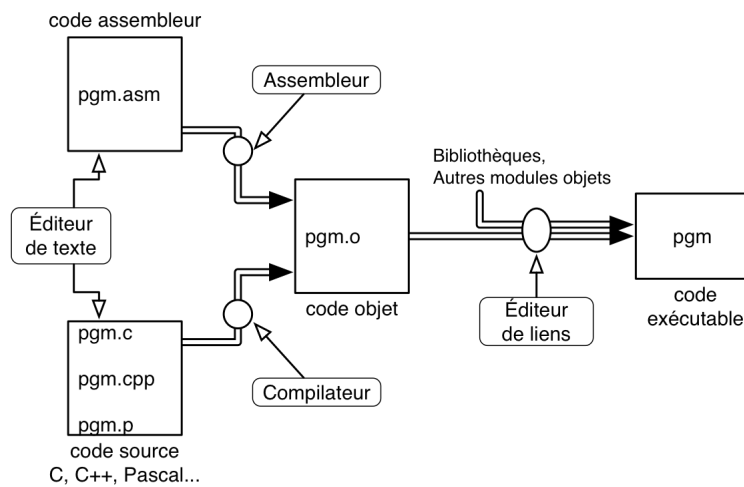
La programmation Java utilise les deux techniques précédentes. Un fichier source en Java est d'abord compilé, mais au lieu de générer des instructions directement pour le processeur, le compilateur produit du code spécifique (appelé *Bytecode*) qui va ensuite être interprété par la machine virtuelle Java (comme ces instructions sont proches du langage assembleur, l'interprétation peut se faire rapidement). L'avantage se trouve dans la portabilité accrue : au lieu de devoir recompiler le programme sur toutes les machines où l'on désire le faire tourner, il suffit de diffuser le code compilé, sous réserve de disposer d'une machine virtuelle Java. Celle-ci peut être facilement installée sur tous les types d'ordinateurs, assistants personnels, téléphones portables, etc. (voir figure 3.1). Le langage Python fonctionne de manière similaire : chaque ligne de code d'un programme Python est traduite en *Bytecode* Python qui est immédiatement interprété par la machine virtuelle Python. Mais à la différence de Java, il n'y a pas séparation de ces deux étapes pour l'utilisateur.

Éditeur de liens

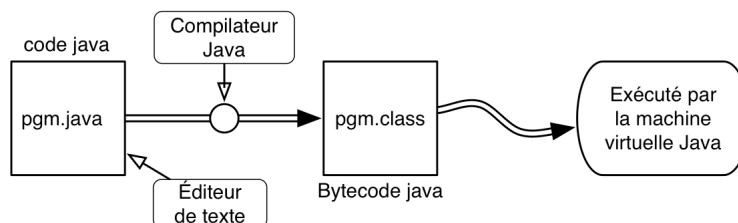
À la figure 3.1, le compilateur génère ce que l'on appelle un fichier objet, qui n'est pas immédiatement exécutable. Ce fichier est bien constitué d'instructions machine, mais ne constitue pas forcément l'intégralité du programme. Celui-ci est souvent découpé en modules séparés, chacun sous la forme de code source. Une fois chacun d'entre eux compilé (sous forme de fichier objet), il faut les réunir en une seule application : c'est le rôle de l'éditeur de liens. Ce programme prend tous les fichiers objet, récupère diverses bibliothèques standard (parties de programme déjà écrites en langage machine) et regroupe le tout en gérant les dépendances (références intermodules) pour fournir au final une application complète, directement exécutable.

Figure 3.1

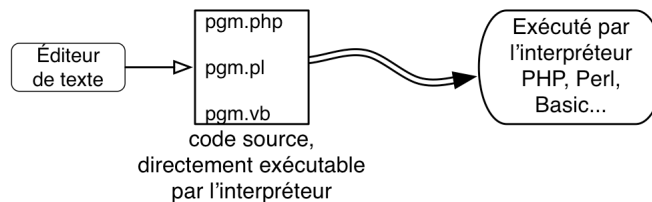
Cycle d'un programme compilé



Cycle d'un programme Java



Cycle d'un programme interprété



Cycle d'un programme.

1.4 Structure simplifiée d'un ordinateur

La figure 3.2 illustre la structure classique d'un ordinateur actuel en séparant les différents composants en entités fonctionnelles.

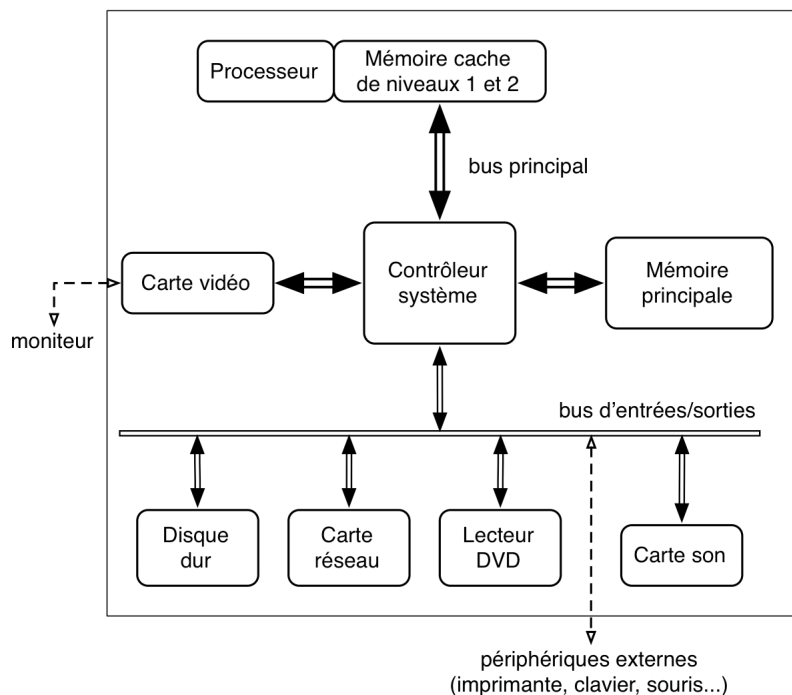
Processeur

Aussi appelé CPU (*Central Processing Unit*), le processeur est le véritable cerveau de l'ordinateur. Toutes les avancées technologiques se concentrent sur ce composant, qui travaille toujours plus vite et effectue des opérations de plus en plus compliquées.

Autrefois agglomérat de circuits physiquement séparés, le microprocesseur est né en 1971 (là encore, le préfixe « micro », à l'origine synonyme de petite taille par rapport aux processeurs sur les gros systèmes, a disparu des dénominations courantes). On est passé de deux mille trois cents transistors (le composant de base des circuits informatiques) pour le premier microprocesseur à plusieurs centaines de millions actuellement.

Le rôle du processeur est d'exécuter les instructions composant le programme. Il se charge de tous les calculs mathématiques et des transferts de données internes et externes. Il décide (en fonction bien sûr des instructions du programme en cours d'exécution) de ce qui se passe à l'intérieur de l'ordinateur.

Figure 3.2



Structure d'un ordinateur.

Stockage

Des données peuvent être stockées dans plusieurs endroits sur l'ordinateur. La zone la plus importante en taille est le disque dur, où données et programmes sont archivés de façon permanente. Cependant, sa relative lenteur (par rapport à la vitesse de fonctionnement du processeur) ne permet pas son utilisation lors de l'exécution des programmes.

Pour pouvoir exécuter une application, il faut amener les instructions la composant ainsi que les données dont elle a besoin dans une mémoire beaucoup plus rapide (c'est-à-dire d'où on peut extraire très vite une information précise) ; il s'agit de la mémoire principale. Elle est constituée de circuits intégrés (pièce électronique dans un boîtier composée de nombreux transistors gravés dans un matériau semi-conducteur) dont la cellule de base est une bascule (voir chapitres 2 et 5). L'unité de base est l'octet (8 bits) et on peut voir la mémoire comme un ensemble ordonné d'octets dans lequel chaque case mémoire (donc chaque octet) a une adresse numérique.

La mémoire principale est un composant passif dans le sens où elle ne fait qu'obéir à des commandes. Le processeur peut lui demander de stocker une donnée à une adresse précise (écriture mémoire) ou de fournir une donnée se trouvant à telle ou telle adresse (lecture mémoire). Dans les deux cas, la mémoire effectue l'opération et attend la demande suivante.

Le processeur est relié à la mémoire via le bus principal de l'ordinateur (ensemble de fils véhiculant les informations et les commandes). Comme le processeur est en permanence en train de travailler avec la mémoire (car il y récupère tout ce dont il a besoin, données et instructions), le bus doit être le plus efficace possible. L'augmentation des performances de l'ordinateur passe également par l'amélioration de la bande passante de ce bus. Celle-ci indique simplement le débit d'informations pouvant circuler sur le bus (en octets par seconde) ; plus il est important, plus le processeur peut travailler rapidement avec la mémoire.

Le progrès technologique a permis de construire des boîtiers mémoire de grande capacité et pouvant répondre rapidement aux sollicitations du processeur. Mais cette amélioration est restée bien en deçà de la formidable accélération des processeurs. Ceux-ci passaient une bonne partie de leur temps à attendre que la mémoire réagisse aux demandes, d'où une perte d'efficacité importante due à ces temps d'attente. Il a donc fallu compliquer un peu le schéma en intercalant entre la mémoire principale et le processeur une mémoire encore plus rapide (mais plus chère et de capacité moindre), appelée « mémoire cache » ou simplement « cache ». Comme la contrainte majeure est la vitesse de réaction et de transfert, ce cache est directement en prise avec le processeur, comme illustré à la figure 3.2. Cette mémoire spéciale permet d'accélérer encore plus les échanges de données avec le processeur, utilisant celui-ci au maximum de ses capacités (voir chapitre 6).

Entrées/sorties

Réduit au processeur et à la mémoire, un ordinateur ne présenterait guère d'intérêt dans ce sens qu'il lui manquerait la possibilité de communiquer avec l'extérieur. Le système d'entrées/sorties regroupe l'ensemble des composants (appelés « périphériques ») permettant à l'ordinateur d'échanger de l'information avec le monde extérieur (clavier, souris, scanner, écran, carte son, imprimante...) et de stocker des informations de manière permanente et de les relire (disque dur, CD/DVD, bandes magnétiques, clés amovibles...).

Les entrées/sorties se caractérisent par leur variété et leur flexibilité. Il faut pouvoir connecter à l'ordinateur divers périphériques, répondant à des commandes différentes et fonctionnant sur une plage de « vitesses » extrêmement large. De plus, les entrées/sorties sont la partie la plus mouvante de l'ordinateur. Alors que le système processeur-mémoire forme un socle à peu près inchangé dans la vie d'un ordinateur, fréquemment l'utilisateur change de périphérique ou ajoute un nouveau dispositif d'entrées/sorties.

Pour ces raisons, on ne peut pas relier directement les entrées/sorties au bus principal car cela figerait complètement les possibilités d'évolution. De plus, on peut sans inconvénient découpler tout le système d'entrées/sorties, qui fonctionne à des vitesses bien moins importantes que le reste, et le bus principal, donc la caractéristique première est de permettre les échanges de données les plus rapides possibles. On branche donc les différentes entrées/sorties sur un bus d'entrées/sorties secondaire, normalisé suivant des standards internationaux, afin de garantir une compatibilité de tous les matériels et une évolution aisée.

La seule exception à cette règle est maintenant la carte vidéo car les besoins en visualisation ont augmenté de façon exponentielle. La complexité des affichages graphiques actuels (taille de l'écran, profondeur des couleurs, plusieurs images par seconde, textures réalistes, 3D...) nécessite un énorme débit d'informations entre le processeur et la carte vidéo (chargée de piloter l'affichage), débit que ne pourrait offrir le bus d'entrées/sorties beaucoup trop lent. Ainsi, la carte vidéo se retrouve connectée directement au bus principal de l'ordinateur via le contrôleur système (voir figure 3.2).

Un dernier composant apparaît à la figure 3.2 : il s'agit du contrôleur système. Avec l'ensemble des éléments qui essaient de communiquer avec le processeur, il risquerait d'y avoir des conflits d'accès au bus principal. Le contrôleur gère donc les différentes demandes pour garantir à chacun un libre accès au bus. Il permet également le passage des informations du bus d'entrées/sorties au bus principal (et réciproquement) en gérant les différences de format de données et de vitesse de transfert entre les deux bus.

1.5 Structure interne d'un processeur

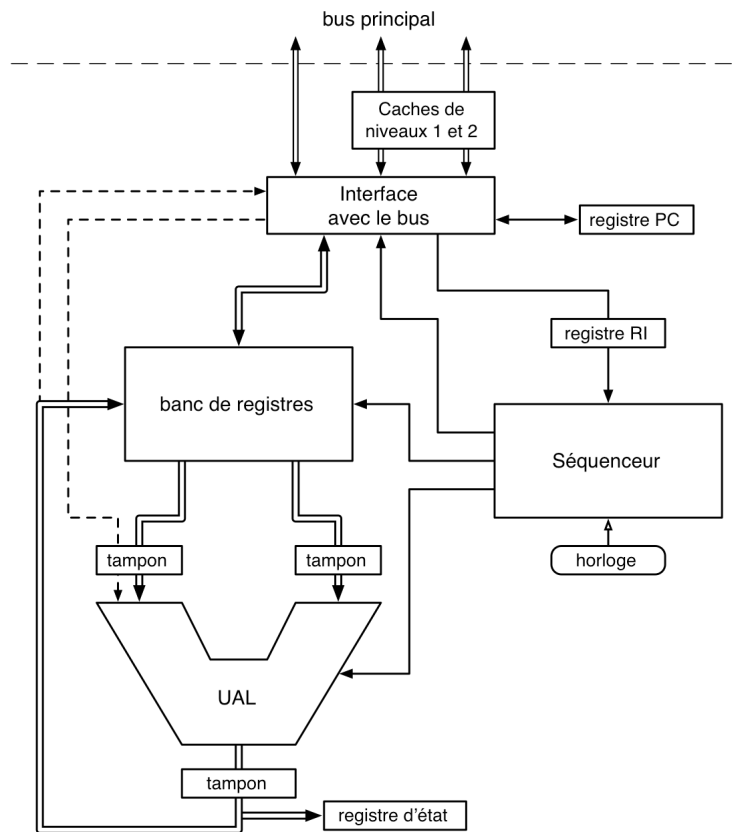
Sur la puce en silicium regroupant quelques centaines de millions de transistors, on trouve à peu près les éléments présentés sur le schéma de la figure 3.3 : une unité de calcul (UAL), une zone de stockage des données (registres) et un élément dirigeant le tout (séquenceur).

Calcul et mémorisation

On peut distinguer des zones fonctionnelles bien définies à l'intérieur du processeur. Citons d'abord l'unité arithmétique et logique (UAL), où s'effectuent les opérations mathématiques. Comme on peut le voir sur la figure, l'UAL prend les opérandes des calculs depuis les registres et renvoie le résultat également dans les registres. Ceux-ci correspondent à une zone de stockage interne du processeur, ce qui permet un accès rapide (beaucoup plus rapide que si les données provenaient directement de la mémoire). Pour alimenter ces registres, le processeur cherche les données en mémoire principale et les ramène *via* le bus principal. À l'arrivée, ces informations sont traitées par le composant marqué « interface avec le bus », qui adapte les signaux circulant entre le processeur et l'extérieur. En fonction de ce qui arrive, il dispatche les bits sur l'une ou l'autre des zones internes. Ainsi, si une donnée provenant de la mémoire est acheminée jusqu'au processeur pour un calcul mathématique, elle est d'abord envoyée dans l'un des registres pour ensuite se retrouver au niveau de l'UAL. Réciproquement, le résultat d'un calcul est temporairement mis dans l'un des registres (ce qui permet une réutilisation immédiate pour un autre calcul) et peut ensuite être renvoyé vers le bus pour un stockage de plus longue durée en mémoire principale. Tous ces liens sont indiqués à la figure 3.3 en traits doubles car ils correspondent à un transfert, non pas de 1 bit, mais de plusieurs en parallèle (leur nombre étant lié à la taille des registres et de l'UAL).

L'UAL, à base de circuits combinatoires, n'a pas de capacité propre de mémorisation. Il faut cependant garantir la présence sur ses entrées des opérandes pendant le temps nécessaire au calcul. C'est le rôle des tampons (*buffer*) situés avant l'UAL. Simples mini-registres de stockage, ils assurent la présence (en les mémorisant temporairement) aux entrées de l'UAL des données provenant des registres principaux. De même, un tampon en sortie de l'UAL permet d'être sûr de disposer du résultat le temps qu'un registre le mémorise, libérant au passage l'UAL.

Figure 3.3



Structure interne d'un processeur.

Commande

Les registres et l'UAL sont régis par des commandes. L'UAL doit savoir quel calcul effectuer (addition, soustraction, multiplication...) et un registre sélectionné peut avoir à envoyer l'information mémorisée sur l'une des entrées de l'UAL ou bien à la transférer vers le bus principal. Le séquenceur, aussi appelé « unité de contrôle », est le chef d'orchestre du processeur. Il a en charge l'envoi des commandes aux différents circuits internes en fonction des instructions à exécuter. À cet effet, il récupère les instructions (en mémoire principale) depuis l'interface via le bus et les stocke une à une dans le registre d'instructions (RI) pendant l'exécution. Il décide alors quels sont les transferts de données nécessaires (registre vers UAL, bus vers registre...) et les commandes à envoyer (lecture ou écriture dans un registre, opération de calcul...), d'où les liens qui le relie à tous les autres circuits.

Le séquenceur est LE composant actif du processeur : il décide ce qui doit se passer à l'intérieur. C'est non seulement le plus important mais le plus compliqué des circuits internes. Plus le processeur est puissant, plus il peut effectuer d'opérations, et plus le séquenceur doit être complexe car il doit gérer des ordres internes élaborés.

Le travail du séquenceur est régi par une horloge, c'est-à-dire un circuit électronique qui rythme l'activité de l'unité de contrôle par des tops électroniques à intervalles réguliers. Le rôle du séquenceur est alors de produire les commandes pour les autres circuits internes du processeur. Plus l'horloge est rapide, plus le séquenceur et donc le processeur peuvent travailler vite. Mais, bien sûr, cela nécessite des circuits plus compliqués, prenant plus de place sur la puce, consommant plus de courant et chauffant davantage.

Pour finir, on trouve à la figure 3.3 un registre PC, qui sert à récupérer la prochaine instruction à exécuter (voir sections suivantes) ainsi que la mémoire cache de niveaux 1 et 2 (historiquement séparée du processeur mais maintenant intégrée à la puce) qui sert à accélérer les transferts entre la mémoire principale et le processeur (voir chapitre 6).

2. LES INSTRUCTIONS

Chaque processeur est capable d'exécuter des instructions se trouvant en mémoire principale. Si elles n'y sont pas, on va d'abord les y mettre en les chargeant depuis leur support de stockage (disque dur, CD/DVD, bande magnétique, clé amovible...).

Une instruction se caractérise par son code numérique correspondant à l'opération requise. Les codes sont stockés en mémoire à la suite les uns des autres dans des cases mémoire successives. On peut ainsi repérer chaque instruction par son adresse, en l'occurrence celle de sa case de stockage.

Pour exécuter une instruction, le processeur envoie un ordre de lecture à la mémoire en indiquant l'adresse de l'instruction souhaitée. La mémoire renvoie alors le code numérique au processeur, qui exécute l'instruction et passe à la suivante. L'ensemble des instructions qu'un processeur peut exécuter, autrement dit son jeu d'instructions (*Instruction Set*), et la correspondance entre chacune d'entre elles et son code numérique, sont définis par le constructeur du processeur et lui sont propres. Ainsi, un programme, c'est-à-dire une suite de codes numériques d'instructions, ne peut s'exécuter que sur un ou plusieurs modèles précis de processeurs. Classiquement, un constructeur cherche une compatibilité ascendante lors de l'évolution de sa gamme : un processeur plus récent voit son jeu d'instructions élargi par de nouvelles fonctionnalités, qui s'ajoutent aux précédentes, ce qui permet d'exécuter des programmes prévus pour un processeur plus ancien. Parfois un constructeur fabrique volontairement un processeur ayant un jeu d'instructions identique à celui d'un processeur concurrent : mêmes instructions et mêmes codes numériques. Il peut ainsi exploiter les mêmes programmes.

2.1 Types d'instructions

Nous allons présenter, dans les sections suivantes, les différents types d'instructions que l'on trouve classiquement dans le jeu d'instructions des processeurs. Nous les avons regroupées selon cinq catégories, qui correspondent à des grands domaines d'actions différents.

Transfert de données

Le programmeur a la possibilité de demander explicitement le transfert de données entre la mémoire et un registre (ou entre deux registres). L'instruction spécifie la case mémoire cible, le registre, ainsi que la taille de la donnée à transférer. En effet, chaque case mémoire correspond à 1 octet alors que les registres ont souvent une taille égale à 4 ou 8 octets. Le programmeur doit donc indiquer dans l'instruction le nombre d'octets à transférer : si c'est plus d'un, on va les chercher dans des cases mémoire successives. Quand on ne remplit pas complètement un registre, il faut indiquer s'il est nécessaire de prévoir une extension de signe pour compléter les bits du registre (voir chapitre 1, section 2.3).

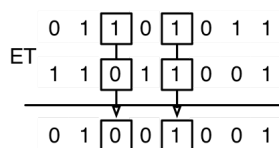
Opérations arithmétiques et logiques, décalages et rotations

Les instructions arithmétiques et logiques permettent d'effectuer les calculs sur des données se trouvant dans les registres. Il est ainsi possible de faire une addition, une soustraction, une multiplication ou une division sur des nombres entiers. Les mêmes opérations existent sur des nombres flottants, ainsi que des fonctions mathématiques classiques (racine carré, fonctions trigonométriques et exponentielles...).

Chaque instruction permet une opération. On définit des calculs plus élaborés instruction par instruction en veillant aux transferts des différents opérandes et des résultats intermédiaires entre registres et UAL.

L'UAL est également capable de réaliser les opérations booléennes de base (NON, ET, OU, XOR) sur des données se trouvant dans les registres. Comme ces opérateurs travaillent au niveau du bit, la donnée du registre n'est plus considérée comme un nombre entier mais comme une suite de bits indépendants. Ainsi, un ET entre deux registres prend chacune des deux valeurs stockées et effectue l'opération sur chacun des bits en parallèle (voir figure 3.4).

Figure 3.4

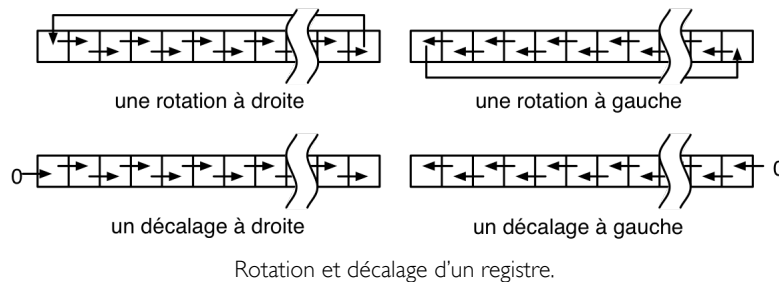


ET sur 8 bits en parallèle.

L'UAL est également capable d'exécuter des instructions de décalage ou de rotation des bits de la donnée d'un registre. La rotation consiste à remplacer chaque bit par celui situé directement à gauche ou à droite (suivant le sens de l'opération). Le bit qui n'a pas de voisin « fait le tour » et prend la place du bit opposé (voir figure 3.5).

Hormis cette dernière étape, le décalage est identique à la rotation, à ceci près qu'un bit disparaît et un 0 (éventuellement un 1, voir chapitre 4 pour un exemple) est mis dans le premier bit (voir figure 3.5). Cette opération est intéressante car un décalage à gauche représente une multiplication par 2 (chaque bit vient occuper la place correspondant à la puissance de 2 supérieure), qui s'exécute beaucoup plus rapidement qu'une opération de multiplication proprement dite. De même, un décalage vers la droite est équivalent à une division par 2 (mais beaucoup plus rapide que la version classique) de la valeur sur laquelle s'opère le décalage.

Figure 3.5



Tests et comparaisons

Les instructions de tests permettent de comparer deux données (égalité, inégalité) afin de définir un comportement différent suivant le résultat de la comparaison. Les données ne sont pas modifiées, mais le résultat du test est indiqué dans un registre spécial (le registre d'état, voir section 3) que l'on pourra examiner lors d'instructions suivantes pour permettre des exécutions différenciées.

Ruptures de séquence

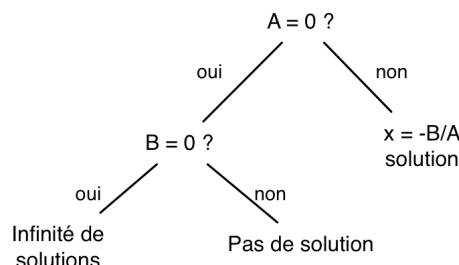
Le processeur exécute normalement les instructions successives en mémoire. Cependant, ce comportement ne permet pas d'avoir des exécutions différentes suivant les données. Il est nécessaire de s'affranchir de cette linéarité en autorisant des ruptures de séquence. Il s'agit d'instructions dont le seul objectif est d'indiquer l'adresse de la prochaine instruction à exécuter (en lieu et place de celle, en mémoire, qui aurait dû l'être, rompant ainsi le comportement par défaut).

Prenons pour exemple la résolution d'une équation du premier degré $Ax + B = 0$. L'algorithme pour trouver la solution est le suivant :

1. Tester si A est nul.
2. Si c'est le cas, tester si B est nul.
3. Si c'est le cas, il y a une infinité de solutions, sinon il n'y en a pas.
4. Si A est différent de 0, la solution de l'équation est $x = -\frac{B}{A}$.

On peut en donner une représentation graphique (voir figure 3.6).

Figure 3.6

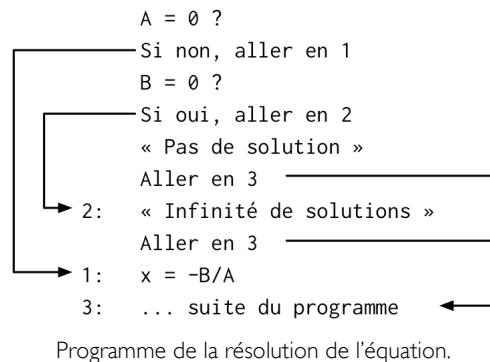


Organigramme de la résolution de l'équation.

Il est cependant nécessaire de la transposer sous une forme linéaire pour pouvoir charger le programme en mémoire. Il faut alors écrire les différentes branches d'exécution les unes après les autres. Pour autant, il ne s'agit pas de les exécuter à la suite, mais d'en sauter certaines compte tenu des résultats des tests. On dispose

de deux types de ruptures de séquence : les sauts (aussi appelés « branchements ») inconditionnels, qui transfèrent systématiquement l'exécution à une autre adresse (au lieu que s'exécute l'instruction suivante en mémoire), et les sauts conditionnels, qui s'effectuent uniquement si une condition est vérifiée (par exemple, si certains bits du registre d'état ont des valeurs précises) ; dans le cas contraire, le saut n'est pas effectué et l'exécution se poursuit normalement à l'instruction suivante en mémoire. La figure 3.7 présente l'écriture « linéaire » de l'algorithme de résolution de l'équation (les flèches illustrent les sauts mais ne font pas partie du programme, bien sûr).

Figure 3.7



Les symboles 1, 2 et 3 désignent un endroit dans la séquence d'instructions, c'est-à-dire une instruction précise ; on les appelle « étiquettes ». Lors de la traduction en langage machine d'un programme, le programme d'assemblage associe à chaque étiquette l'adresse mémoire de l'instruction correspondante afin de pouvoir remplacer l'étiquette, dans les instructions de branchement, par l'adresse de l'instruction cible.

Instructions divers

Les autres instructions d'un jeu standard concernent les entrées/sorties (possibilité d'envoyer directement une commande à un périphérique), le contrôle du processeur (arrêter ou redémarrer un processeur), la gestion des interruptions (voir chapitre 8), etc.

2.2 Format des instructions

Une instruction processeur peut être découpée en deux parties : l'opération elle-même et les données sur lesquelles elle porte. La première partie, appelée « code opération », correspond au mnémonique utilisé par l'assembleur (ADD, MOV, SUB...), alors que la seconde doit être spécifiée *via* des modes d'adressage qui permettent d'indiquer où sont les données (en mémoire, dans un registre, mises explicitement dans l'instruction...).

Chacune de ces deux parties intervient dans le code numérique final de l'instruction. Par exemple, les 8 premiers bits de l'instruction peuvent indiquer le code opération voulu tandis que les 24 autres bits décrivent les données. Ce découpage est le fait du constructeur lorsqu'il définit le jeu d'instructions de son processeur.

Code opération

Ce code indique au processeur quelle opération effectuer parmi toutes celles autorisées. Le mnémonique correspondant se traduit en une valeur numérique dont les bits spécifient quelles actions (transfert de données, calcul...) doivent intervenir à l'intérieur du processeur pour l'exécution de l'instruction. Ces bits du code opération sont utilisés par le séquenceur pour l'envoi des commandes aux différents composants internes du processeur.

Nombre d'opérandes

La plupart des opérations arithmétiques nécessite de spécifier trois données : les deux sur lesquelles portent l'opération (addition, soustraction...) ainsi que l'emplacement où ranger le résultat (qui, sinon, est perdu). Par exemple, le concepteur d'un jeu d'instructions peut décider que ADD r1,r2,r3 additionne les valeurs se trouvant dans les registres r2 et r3, et place le résultat dans le registre r1. C'est une convention assez classique d'indiquer, comme ici, la destination en premier dans la liste des données d'une instruction, mais le concepteur peut décider qu'il faut écrire ADD r2,r3,r1 pour obtenir le même résultat.

Quasiment tous les processeurs actuels proposent cette souplesse d'utilisation, en autorisant la mention de plusieurs registres dans les instructions, mais cela a deux inconvénients : d'abord, la gestion interne des transferts de données au sein du processeur s'en trouve compliquée puisque tous les chemins deviennent

possibles pour une donnée, obligeant à avoir un séquenceur plus conséquent ; ensuite, il faut prévoir une taille suffisante pour les codes numériques des instructions puisqu'il y a trois données à spécifier par des valeurs numériques. Historiquement, la mémoire principale n'était pas conséquente dans les ordinateurs et ce dernier désavantage était ressenti comme un gros handicap. Il fallait absolument minimiser la place mémoire occupée par un programme et l'une des méthodes était de minimiser la taille des instructions. Celles-ci ne pouvaient bien souvent expliciter que deux adresses de données, l'une des deux servant implicitement de destination : `ADD r1,r2` additionnait la valeur des registres `r1` et `r2` et rangeait le résultat dans `r1` (par convention). Ce que l'on gagnait en place mémoire (deux données à spécifier au lieu de trois, d'où des codes numériques d'instructions plus courts), on le perdait en souplesse d'utilisation puisque le programmeur devait jongler avec l'affectation des registres.

Remarque

Parmi les premiers processeurs, certains avaient poussé le concept encore plus loin en imposant un registre spécial (appelé « accumulateur »), qui servait implicitement pour toutes les opérations arithmétiques et logiques. On ne devait plus le spécifier dans l'instruction : `ADD r1` additionnait le registre `r1` avec l'accumulateur et mettait le résultat dans ce dernier. Les possibilités de programmation étaient encore plus rigides (il fallait constamment transférer les données entre la mémoire et les registres), mais les instructions prenaient peu de place et le séquenceur était simple, ce qui était important à une époque où la place sur la puce (en nombre de transistors) était restreinte.

Taille des instructions

Toutes les instructions ne font pas intervenir trois données. Certaines opérations arithmétiques (prendre l'opposé d'un nombre par exemple) ou logique (calculer le NON d'une valeur binaire) n'en nécessitent que deux : le nombre source sur lequel s'exécute l'opération, et la destination. De même, les transferts de données se font entre deux endroits à spécifier. Les sauts, quant à eux, n'ont besoin que d'une information, l'adresse cible où poursuivre l'exécution.

Il y a donc un nombre variable de données pour les instructions et, de plus, toutes les données ne nécessitent pas le même nombre de bits (il en faut beaucoup plus pour indiquer des adresses mémoire que des registres, moins nombreux). Il est donc naturel de prévoir des instructions de longueur variable utilisant le nombre de bits nécessaire : peu s'il y a peu de données dans l'instruction, beaucoup si elles sont nombreuses ou compliquées. Et c'est effectivement ainsi que fonctionnaient les anciens processeurs, la mémoire étant occupée de façon efficace par un programme. Malheureusement, cela voulait aussi dire qu'il était lent et compliqué pour le processeur d'exécuter une instruction car il ne pouvait pas savoir à l'avance ce qui l'attendait lorsqu'il récupérait un code numérique : fallait-il charger 2, 3, 4, voire plus d'octets pour avoir l'instruction en totalité ? Où étaient les limites des adresses de chaque donnée dans cette suite de bits ? Le résultat était un séquenceur lent et complexe, qui devait être capable de traiter tous les cas.

Le désir d'accélérer les processeurs a conduit à une simplification radicale du jeu d'instructions de la plupart des processeurs (aidée par l'augmentation des capacités mémoire qui rendait moins problématique l'accroissement de la taille des programmes) sous la forme d'instructions de taille fixe, souvent 32 bits. En rationalisant l'expression des données des instructions, on a paradoxalement simplifié le séquenceur et donc permis son optimisation.

2.3 Modes d'adressage

Chaque processeur possède plusieurs modes d'adressage pour les instructions, qui permettent de préciser, dans celles-ci, la localisation des données. L'existence de modes d'adressage variés pour l'accès aux données s'explique souvent par un souci de simplification des accès aux structures de données complexes des langages évolués, permettant ainsi au compilateur une traduction plus facile des programmes en code assembleur.

Adressage immédiat

Le plus simple est de pouvoir préciser la donnée directement dans l'instruction, sans avoir à la stocker quelque part (mémoire ou registre) avant. Ainsi, une instruction `ADD r1,r2,#3` ajoute la valeur 3 (précisée dans l'instruction) au registre `r2` et range le résultat dans `r1`. Le terme « adressage immédiat » vient du fait que la donnée est immédiatement disponible dans l'instruction sans que le processeur ait besoin de faire un nouvel accès mémoire pour la récupérer. L'inconvénient est qu'elle doit être indiquée dans l'instruction et ne peut donc pas dépendre d'un calcul antérieur ; c'est donc forcément une constante. Il est assez classique d'utiliser le caractère # pour symboliser une donnée immédiate, et pas un numéro de registre ou une adresse mémoire (même si certains programmes d'assemblage ont une convention différente). De même, l'écriture de la constante est souvent en hexadécimal par défaut.

Adressage par registre

L'un des endroits où une valeur peut être stockée est bien sûr l'un des registres du processeur. Il faut donc avoir la possibilité de le désigner comme source ou destination dans les instructions. Tous les programmes d'assemblage permettent d'utiliser le mode d'adressage par registre simplement en indiquant le nom du registre voulu. Chaque processeur a sa propre convention de nommage : des modèles anciens, avec peu de registres, attribuaient à chacun une simple lettre comme A ou B (ADD A,B affecte la somme du registre A et du registre B à A) et leur confiaient des rôles spécifiques (certains pour les opérations arithmétiques, d'autres pour les accès mémoire) ; mais il est maintenant plus courant d'avoir des registres ayant des fonctions identiques, et portant un numéro : D0 à D7, r0 à r255, etc.

Adressage direct

On peut faire référence à une case mémoire précise directement dans l'instruction en donnant son adresse. L'instruction `MOVE r1,3afc` (notez l'absence de #) prend la donnée se trouvant en mémoire à l'adresse hexadécimale 3afc₁₆ pour la mettre dans le registre r1. Dans le modèle de processeur de la figure 3.3, l'UAL ne pouvant être alimentée que par les registres, on ne pourra pas utiliser ce mode d'adressage direct pour des instructions arithmétiques, mais seulement pour des instructions de transfert mémoire vers un registre, ou l'inverse.

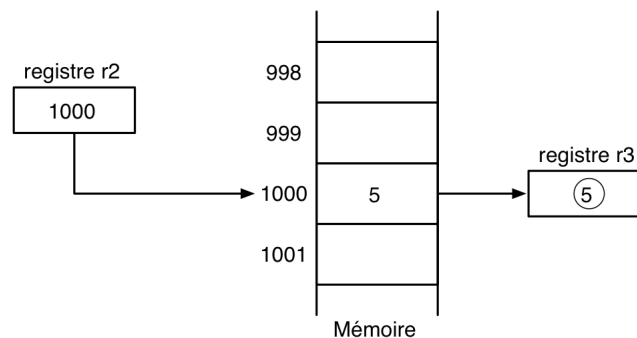
L'inconvénient de ce mode d'adressage est bien sûr qu'il faut connaître l'adresse souhaitée lors de l'écriture du programme. Or celle-ci peut dépendre de l'emplacement où va être chargé le programme au moment de son exécution. On peut alors remplacer l'écriture littérale de l'adresse par une étiquette (préalablement définie et associée à une case mémoire) et laisser le programme d'assemblage faire la correspondance avec l'adresse mémoire. Un autre inconvénient de ce mode d'adressage est qu'il oblige à avoir beaucoup de bits dans l'instruction pour coder l'adresse. C'est pourquoi on lui préfère souvent le mode d'adressage qui suit pour les références mémoire.

Adressage indirect par registre

Au lieu d'utiliser directement une adresse mémoire dans l'instruction, on peut indiquer où trouver cette adresse, dans un registre par exemple. Il faut distinguer ce mode d'adressage indirect du mode d'adressage par registre. En mode registre, la donnée utilisée dans l'instruction est la valeur contenue dans le registre alors qu'en adressage indirect, c'est l'adresse mémoire de la donnée qui est dans le registre (voir figure 3.8).

Figure 3.8

L'instruction `MOVE r3,(r2)`



Adressage indirect par registre.

Pour les distinguer, on note le mode d'adressage indirect en mettant le registre entre parenthèses ou entre crochets : `MOVE r3, (r2)` charge dans le registre r3 la valeur située en mémoire à l'adresse se trouvant dans r2.

Il y a deux avantages à spécifier ainsi les adresses : d'abord l'instruction nécessite moins de bits car il n'y a qu'un numéro de registre à indiquer ; ensuite il est facile de changer cette adresse par une simple opération arithmétique sur le registre. Si l'on veut parcourir un tableau d'octets en mémoire, il suffit d'initialiser le registre avec l'adresse du premier élément, puis de l'incrémenter pour qu'il « pointe » sur l'octet suivant (c'est-à-dire qu'il contienne son adresse).

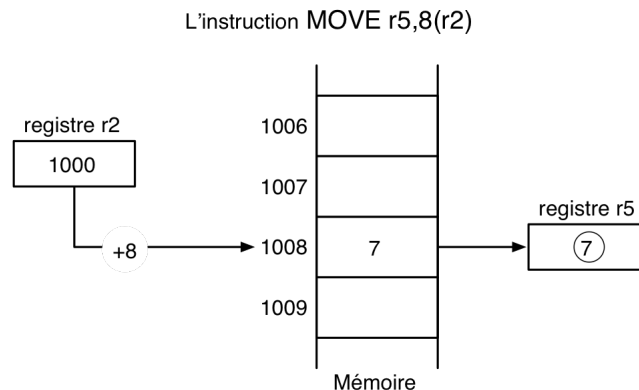
Les modes d'adressage précédents sont universels et existent sur tous les processeurs. On peut également trouver, suivant les modèles, des modes d'adressage plus « exotiques » et donc moins systématiques.

Adressage indirect avec déplacement

On peut combiner le mode d'adressage indirect avec un décalage. C'est le mode indirect avec déplacement. L'adresse de la donnée est la somme de l'adresse contenue dans un registre (mode indirect) et d'un

déplacement immédiat indiqué dans l'instruction. `MOVE r5,8(r2)` met dans `r5` la valeur dont l'adresse mémoire est obtenue en ajoutant 8 à celle contenue dans `r2` (voir figure 3.9)

Figure 3.9



Adressage indirect avec déplacement.

Une utilisation classique est l'accès à une structure de données composée de plusieurs champs. On initialise le registre avec l'adresse mémoire du début de la structure et les différentes valeurs du déplacement permettent d'accéder aux différents champs sans changer la valeur du registre.

Adressage indirect avec post-incrémentation ou pré-décrémentation

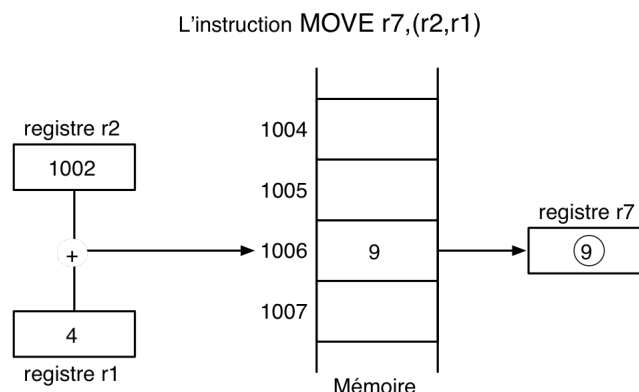
L'adressage indirect est tellement utile pour parcourir un tableau que certains processeurs le déclinent sous deux formes :

- Avec post-incrémentation, il permet, en une seule instruction, non seulement d'accéder à une donnée en mémoire dont l'adresse est dans un registre (mode indirect), mais en plus d'incrémenter le registre de la taille de l'élément pointé (souvent 1, 2 ou 4 octets). On peut ainsi préparer l'accès à l'élément suivant (en ayant maintenant son adresse dans le registre). On le note $(r0)+$.
- L'adressage symétrique, dit « indirect avec pré-décrémentation » et noté $-(r0)$, retranche la taille de l'élément avant d'accéder à la mémoire. On parcourt ainsi un tableau en sens inverse.

Adressage indirect indexé

L'adressage indirect avec déplacement souffre du même défaut que l'adressage direct : la valeur du déplacement doit être indiquée « en dur » dans l'instruction et ne peut pas être calculée dynamiquement lors de l'exécution. Pour cela, il faudrait qu'elle soit dans un registre. C'est exactement ce que permet l'adressage indirect indexé. L'adresse mémoire de la donnée est la somme du contenu d'un registre de base (mode indirect) et d'un registre jouant le rôle d'un déplacement (voir figure 3.10) : `MOVE r7,(r2,r1)`.

Figure 3.10



Adressage indirect indexé.

Certains processeurs autorisent même un mode indexé avec déplacement : `MOVE r4,4(r0,r3)` somme les deux registres `r0` et `r3` avant d'ajouter la valeur 4 pour trouver l'adresse mémoire de la donnée voulue.

Plus le processeur propose de modes d'adressage différents et complexes, plus le séquenceur a de travail pour décoder les instructions et plus il est difficile de l'optimiser pour améliorer ses performances.

3. UAL ET REGISTRES

L'UAL et les registres font partie des composants internes du processeur qui obéissent aux commandes du séquenceur, envoyées en fonction des instructions exécutées.

3.1 Construction d'une UAL

L'UAL est un circuit logique combinatoire formé de portes logiques prenant deux nombres en entrée et générant un nombre en sortie en fonction de signaux de commande indiquant l'opération, arithmétique ou logique, à effectuer. Une UAL peut être caractérisée par sa taille et ses possibilités.

Sa taille, ou largeur, correspond au nombre maximum de bits que peuvent occuper les entrées, autrement dit à la taille maximale des nombres que l'UAL peut traiter. Elle est évidemment fortement liée à la taille des nombres que les registres peuvent stocker puisque ce sont les mêmes qui passent par l'UAL. Il est toujours possible de travailler avec des nombres plus grands, mais il faut alors découper l'opération : on peut additionner de deux nombres de 64 bits dans une UAL de 32 bits de large en ajoutant d'abord les 32 bits de poids faible puis les 32 bits de poids fort, mais cela nécessite au moins deux instructions. Inversement, calculer sur des nombres de 32 bits dans l'UAL ne présente pas d'intérêt si les registres ne peuvent pas stocker plus de 16 bits.

Les possibilités de l'UAL correspondent simplement aux différentes commandes qu'elle reconnaît : elle peut certainement faire une addition et une soustraction, mais qu'en est-il des multiplications et divisions ? Toutes les fonctions logiques sont-elles possibles ?

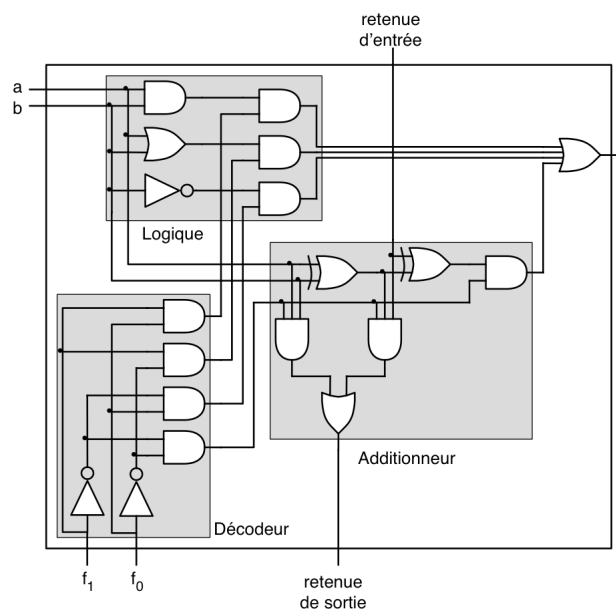
Le constructeur d'un processeur a toujours la possibilité d'étendre les capacités de l'UAL (en autorisant des opérations plus complexes) ou d'augmenter sa taille (pour permettre la manipulation de nombres plus grands). Mais cela implique d'utiliser plus de portes logiques et donc d'occuper plus de place sur la puce, au détriment des autres circuits.

Un dernier paramètre crucial de l'UAL est sa vitesse de fonctionnement. Puisqu'elle est sollicitée à chaque instruction de calcul, sa rapidité influence directement la vitesse d'exécution. Il faut donc en premier lieu optimiser ce critère.

Exemple d'UAL 1 bit

La figure 3.11 montre un exemple d'UAL 1 bit simplifiée. Elle est capable de calculer le ET et le OU de 2 bits, le NON du second bit, et la somme des 2 bits avec une retenue d'entrée. Le choix parmi ces quatre opérations se fait via les deux lignes de commandes f_0 et f_1 . Suivant la valeur de ces 2 bits, un décodeur active une des quatre lignes de sortie, sélectionnant soit une des trois fonctions logiques (ab , $a + b$, \bar{b}), soit la retenue et la sortie de l'additionneur.

Figure 3.11



UAL 1 bit simplifiée.

Pour concevoir une UAL plus large, il suffit de « chaîner » plusieurs UAL 1 bit, reliant la retenue de sortie d'un étage à la retenue d'entrée du suivant.

Plusieurs UAL

L'exemple précédent concernait une UAL effectuant une addition sur un nombre entier. Les circuits nécessaires au calcul sur des nombres flottants sont totalement différents et on ne gagnerait rien à les incorporer au même circuit combinatoire. Les processeurs ont donc maintenant deux UAL distinctes, une pour les opérations logiques et pour les calculs entiers, une autre pour les opérations en virgule flottante. L'avantage est de pouvoir accélérer les traitements en effectuant deux calculs simultanés, un dans chaque UAL.

3.2 Registres

Les registres sont une zone de stockage temporaire des données située dans le processeur. Ils sont construits à l'aide de circuits logiques séquentiels afin de pouvoir mémoriser des bits. Leur intérêt principal est de pouvoir travailler avec des données localisées directement dans le processeur et donc d'un accès beaucoup plus rapide que celles situées en mémoire principale.

Registres généraux

Les registres généraux (regroupés dans un banc de registres) sont à la disposition du programmeur en assembleur (ou du compilateur si l'on code en langage évolué), qui les utilise à sa guise dans les instructions pour manipuler des données.

On a évidemment intérêt à travailler au maximum avec les registres généraux et à n'utiliser le stockage en mémoire qu'en dernier ressort car cela ralentit fortement l'exécution des instructions. Suivant les processeurs, on dispose d'une dizaine à une centaine de registres généraux. Plus ils sont nombreux, moins le programme fait appel à la mémoire, mais plus les circuits prennent de la place sur la puce. Ces registres sont caractérisés, comme l'UAL, par leur taille : combien de bits peuvent-ils stocker ? Depuis plusieurs années, les processeurs sont capables de travailler avec des nombres de 32 bits et les registres pour les nombres entiers sont classiquement de cette largeur, comme l'UAL correspondante. Les processeurs récents possèdent maintenant des registres de 64 bits.

Cependant, deux problèmes se présentent. D'abord il n'y a aucun moyen de distinguer, une fois mis dans un registre, un nombre entier d'un nombre flottant ; or les calculs doivent se faire dans des UAL différentes. Ensuite, il existe une norme de représentation des flottants sur 64 bits et il faut bien pouvoir les mémoriser dans le processeur. Ces deux raisons font qu'il existe deux bancs de registres dans un processeur : l'un pour stocker les nombres entiers, reliés à l'UAL entière, et l'autre pour les nombres flottants, reliés à l'UAL flottante. Chaque banc de registres est situé au plus près de l'UAL correspondante ; les transferts s'en trouvent accélérés.

Il faut alors pouvoir distinguer les registres entiers et flottants dans les instructions et les modes d'adressage. C'est pourquoi on les nomme différemment, par exemple *r0* pour un registre de type entier et *f0* pour un flottant. Les instructions arithmétiques agissant sur chaque type portent également des noms différents de sorte qu'on ne les confonde pas (ADD et FADD par exemple).

Ces bancs de registres sont un peu plus complexes que les circuits séquentiels traditionnels car, comme chaque instruction peut avoir deux registres comme source (par exemple ADD *r1*, *r2*, *r3*), chaque banc autorise deux accès simultanés permettant de récupérer les deux valeurs souhaitées en une seule opération.

En plus des registres généraux, chaque processeur possède des registres spécialisés nécessaires au bon déroulement des instructions.

Registre d'instruction

Lorsqu'une instruction est récupérée en mémoire pour être exécutée dans le processeur, elle est mémorisée dans un registre spécial : le registre d'instruction (RI ou IR, *Instruction Register*, voir figure 3.3). Le séquenceur utilise ce lieu de stockage pour disposer de l'instruction et des bits la composant pendant tout le temps de son exécution. Il gère entièrement l'utilisation de ce registre et le programmeur n'y a jamais accès.

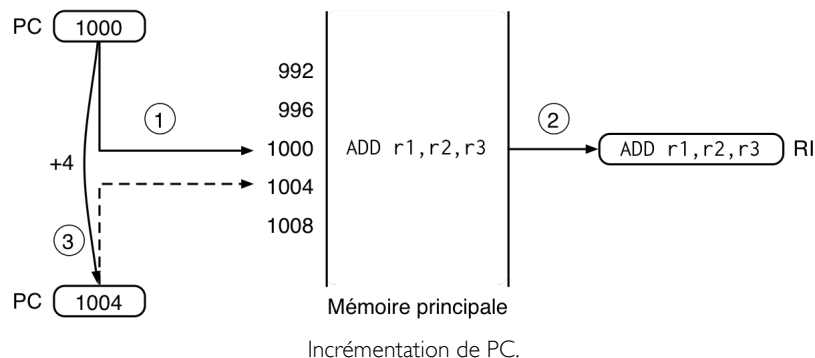
Compteur ordinal

Avant son exécution, un programme est mis en mémoire, ses instructions étant placées les unes à la suite des autres. Chacune est donc à une adresse précise, qu'il faut envoyer au boîtier mémoire lorsque le processeur veut récupérer ladite instruction pour l'exécuter. Il doit donc à tout moment savoir quelle est la prochaine instruction à exécuter et surtout quelle est son adresse. Un registre spécial, appelé « compteur ordinal » (ou

encore, suivant les modèles de processeurs, « PC » pour *Program Counter*, ou « IP » pour *Instruction Pointer*) contient l'adresse en question (voir figure 3.3).

Pour exécuter une instruction, le processeur commence par envoyer un ordre de lecture mémoire associé à la valeur contenue dans PC (d'où le lien entre PC et l'interface avec le bus à la figure 3.3) avant de récupérer l'instruction. Le processeur incrémente alors PC de la taille de l'instruction pour que le registre pointe sur la suivante (rappelons que chaque adresse mémoire correspond à 1 octet et qu'une instruction est longue de plusieurs octets ; PC doit donc être augmenté du nombre d'octets de l'instruction, et non de 1). La figure 3.12 montre le début de l'exécution d'une instruction : sa récupération à l'adresse contenue dans PC, le stockage dans RI et l'incrémement de PC.

Figure 3.12



Le programmeur n'a pas directement accès à ce registre mais peut le modifier via les instructions de branchement. Celles-ci déroutent le processeur de son parcours normal, géré automatiquement par le séquenceur (exécution linéaire des instructions par incrémement de PC), pour lui faire reprendre l'exécution à une autre adresse mémoire. Il suffit, pour ce faire, de remplacer la valeur de PC par l'adresse cible du saut, indiquée dans l'instruction de branchement.

Registre d'état

Le registre d'état (*State Register* ou *Program Status Word*) est tel que ses bits ne forment pas de valeur numérique mais servent d'indicateurs (aussi appelés « drapeaux » ou *flags*) sur l'état du processeur. Certains bits peuvent être positionnés par le programmeur pour demander un comportement particulier (par exemple fixer un niveau de masquage des interruptions, voir chapitre 8), d'autres (appelés « bits conditions ») sont automatiquement mis à jour par le séquenceur à la fin de l'exécution de chaque instruction. Ils sont utilisés pour les sauts conditionnels en liaison avec les instructions de test et de comparaison.

La « condition » liée à un branchement conditionnel consiste toujours en un test d'une valeur particulière d'un ou plusieurs bits conditions, positionnés par l'instruction précédente. Certains de ces bits, tels ceux qui suivent, sont suffisamment universels pour qu'on les retrouve sur tous les modèles de processeurs (mais leur nom peut varier) :

- **Le bit Z (Zero).** Il est mis à 1 si le résultat de l'instruction est nul, sinon il est mis à 0.
- **Le bit C (Carry).** Il est mis à 1 si l'instruction génère une retenue finale, sinon il est mis à 0.
- **Le bit N (Negative).** Il est mis à 1 si le résultat de l'instruction est négatif, sinon il est mis à 0 ; c'est la recopie du bit de poids fort du résultat.
- **Le bit V (oVerflow).** Il est mis à 1 si l'instruction génère un débordement arithmétique, sinon il est mis à 0.

L'utilisation de ces bits conditions se fait toujours de la même manière : d'abord, on effectue un calcul, un test ou une comparaison, positionnant les bits conditions suivant le résultat que l'on veut obtenir, puis l'on effectue un saut conditionnel sur ces bits.

Supposons par exemple que l'on veuille tester si le registre *r1* est nul. On peut écrire le code de la façon suivante :

```
ADD r0,r1,#0
JZS adresse_cible
```

La première instruction additionne *r1* et 0 et met le résultat dans *r0*. Ce résultat ne peut être nul que si *r1* lui-même était nul au départ. Le bit Z n'est donc à 1 après cette instruction que si *r1* était nul. Le branchement n'est effectif que si le bit Z est à 1 (JZS signifie *Jump if bit Z Set*, soit « sauter si le bit Z est à 1 »). On a donc bien deux exécutions différentes suivant la valeur de *r1* : s'il est non nul, le processeur continue avec les instructions situées après le branchement ; s'il est nul, le programme se poursuit à l'adresse cible indiquée dans le branchement emprunté.

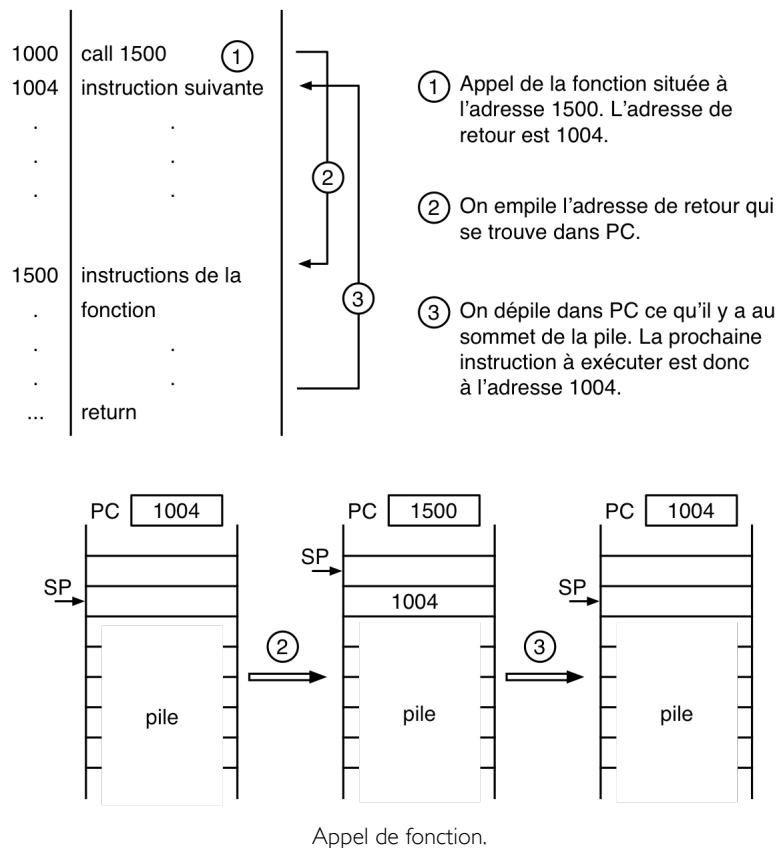
Notez que, dans certains processeurs, de plus en plus nombreux, les tests sont directement réalisés sur le contenu des registres, et non plus sur un registre d'état dont la mise à jour est délicate dans le cas d'une architecture incluant un pipeline, une exécution dans le désordre, etc.

Registre pointeur de pile

Lors de l'exécution de programmes, le processeur doit parfois stocker de l'information en mémoire, non pas à la demande explicite du programmeur, mais afin d'assurer le bon fonctionnement des instructions. Un exemple typique est l'appel de fonction : dans un langage évolué, le programme est dérouté pour que s'exécute ladite fonction puis reprend normalement là où il s'était arrêté (à l'inverse d'un saut qui le dérouté définitivement à une autre adresse). Le même mécanisme existe en assembleur ; il faut alors mémoriser la valeur de PC avant d'exécuter la fonction pour pouvoir reprendre le programme après l'appel.

À cette fin, une structure de pile est implémentée en mémoire et un registre spécial du processeur (appelé « pointeur de pile » ou SP, *Stack Pointer*) pointe sur le haut de la pile, c'est-à-dire sur la première case mémoire libre à son sommet. Là sont empilées ou dépilées des informations, soit automatiquement par le séquenceur, soit à la demande du programmeur. Des instructions spéciales (PUSH ou POP) sont à sa disposition, qui accèdent à la mémoire via le registre SP par un adressage indirect. La figure 3.13 illustre l'usage de SP lors de l'appel d'une fonction pour mémoriser l'adresse de retour. De façon générale, la pile sert également à passer des paramètres à une fonction et à stocker ses variables locales : ces deux types d'informations n'ont pas d'existence en dehors de la fonction en question et, à l'entrée de celle-ci, la pile permet justement de les créer (par empilement), et de les détruire à la sortie (par dépilement).

Figure 3.13



4. SÉQUENCEUR

Le séquenceur est directement lié à la puissance du processeur puisqu'il est responsable du bon déroulement des instructions. Il doit prendre en charge tous les transferts de données et l'envoi de toutes les commandes aux autres composants internes.

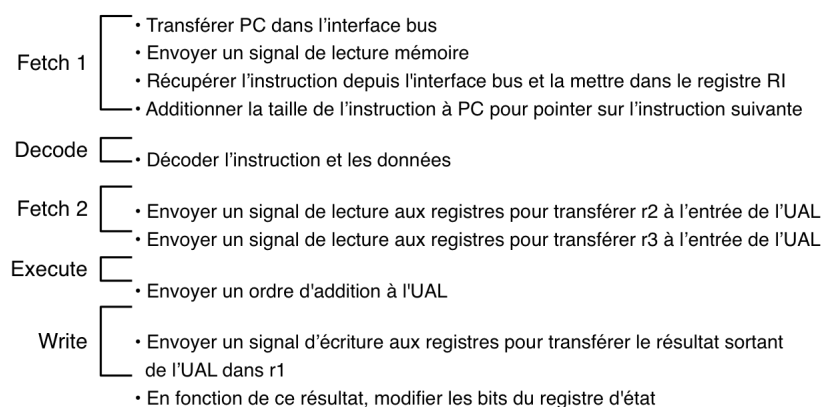
4.1 Cycle d'instruction

L'exécution d'une instruction peut se décomposer en un certain nombre d'étapes composant le cycle d'instruction :

- récupération de l'instruction et mise à jour de PC (*Fetch 1*) ;
- décodage de l'instruction (*Decode*) ;
- récupération des données (*Fetch 2*) ;
- exécution de l'instruction (*Execute*) ;
- écriture du résultat et modification des bits conditions (*Write*).

Chaque étape est gérée par le séquenceur et transformée en ordres internes. La figure 3.14 présente un exemple de l'exécution en interne de l'instruction `ADD r1, r2, r3`.

Figure 3.14



Exécution de `ADD r1, r2, r3`.

Chaque instruction demande ainsi de nombreux ordres et transferts internes, entièrement contrôlés par le séquenceur et rythmés par son horloge. Plus les instructions du processeur sont compliquées, plus elles nécessitent d'étapes internes et plus le séquenceur est complexe.

4.2 Séquenceur câblé

Les premiers microprocesseurs avaient des séquenceurs simples, qui calculaient les commandes internes à envoyer en fonction du code numérique de l'instruction à exécuter. Ces séquenceurs étaient composés de simples circuits logiques combinatoires, qui généraient les ordres à l'aide de fonctions logiques à partir des bits composant le code numérique de l'instruction. Ceux-ci n'étaient donc pas choisis au hasard mais devaient correspondre aux lignes de commande à activer (transfert de données, commande à l'UAL...) ; ils s'agissait de séquenceurs câblés. Ce système de séquenceur est assez simple, ne nécessite pas de circuits compliqués, mais a une puissance limitée. En effet, on ne peut pas obtenir une infinité de fonctions logiques différentes à partir des bits du code de l'instruction et cela restreint les possibilités pour les commandes internes. Une instruction processeur complexe, demandant de nombreuses étapes internes pour son exécution, ne peut être implémentée avec un séquenceur câblé. Pour ce faire, il faut mettre en place un séquenceur plus compliqué, capable d'exécuter des micro-instructions.

4.3 Séquenceur microprogrammé

Une instruction peut se décomposer en plusieurs micro-instructions, chacune correspondant à une action interne du processeur. Pour augmenter les possibilités de ce dernier, on a transformé le séquenceur en processeur miniature, exécutant, pour chaque instruction, un programme formé des micro-instructions. À l'intérieur du séquenceur se trouve une mémoire de microprogramme contenant la traduction en micro-instructions de chaque instruction processeur, ainsi qu'un microcompteur ordinal chargé de gérer l'exécution de celles-ci. Lorsqu'une instruction processeur est récupérée, son code numérique localise l'adresse de début du microprogramme d'exécution dans cette mémoire. Le microcompteur ordinal exécute alors les micro-

instructions dans l'ordre, chacune opérant un transfert interne de données ou envoyant une commande à un composant du processeur.

L'avantage de cette technique est de pouvoir proposer des instructions processeur plus complexes car il n'y a aucune limite au nombre de micro-instructions traduisant une instruction processeur. En revanche, cela complique et ralentit fortement le séquenceur, qui occupe beaucoup plus de place sur la puce (au détriment des autres entités fonctionnelles) car il faut y mettre la mémoire de microprogramme ainsi que les circuits nécessaires à l'exécution des micro-instructions.

La mémoire de microprogramme est bien sûr conçue une bonne fois pour toutes par le constructeur du processeur et n'est pas modifiable. Le programmeur n'y a pas accès, ce mécanisme étant totalement invisible pour lui ; en d'autres termes, le niveau des instructions processeur est à sa portée, mais pas le niveau inférieur.

Les processeurs actuels essaient de combiner les deux techniques de contrôle en exécutant les instructions processeur simples par des circuits logiques rapides et en réservant le découpage en micro-instructions aux instructions processeur les plus complexes.

4.4 RISC contre CISC

Durant la décennie 1970-1980, les processeurs ont progressé en complexité, profitant de l'intégration poussée des transistors pour offrir des jeux d'instructions plus complets et un séquenceur microprogrammé, malheureusement au détriment des autres composants internes, restreints à la portion congrue sur la puce. Entre autres, les données des instructions pouvaient se trouver en registre bien sûr, mais également en mémoire, même pour les opérations arithmétiques (d'où les liens en pointillés à la figure 3.3, entre l'UAL et l'interface avec le bus).

En 1981, deux universitaires ont suggéré de réduire drastiquement le jeu d'instructions des processeurs, par exemple en interdisant les opérandes situés en mémoire, sauf pour les transferts, en limitant les modes d'adressage possibles, et en proscrivant les instructions trop compliquées. Ils avaient constaté que seulement 20 % du jeu d'instructions était utilisé dans 80 % des instructions d'un programme standard.

Ils ont alors proposé de construire un processeur ayant un jeu d'instructions réduit (RISC, *Reduced Instruction Set Computer*) par opposition aux processeurs existants (CISC, *Complex Instruction Set Computer*). Cela devait permettre de simplifier fortement le séquenceur, de revenir à un séquenceur câblé, d'introduire un pipeline (voir section suivante), d'optimiser le tout pour accroître la vitesse d'horloge, et d'utiliser la place libérée sur la puce pour augmenter le nombre de registres (et par là même limiter le nombre d'accès mémoire) et introduire la mémoire cache directement dans le processeur. La réduction du jeu d'instructions entraînant un allongement des programmes (car certaines instructions disponibles sur les processeurs CISC doivent être simulées par plusieurs instructions successives sur les processeurs RISC), les promoteurs de ces processeurs espéraient alors que le ralentissement induit serait plus que compensé par l'accélération des performances des processeurs. En fait, le passage d'un processeur CISC à un processeur RISC transférait la complexité du séquenceur au compilateur, qui devait optimiser le code pour utiliser le pipeline et profiter des nombreux registres.

Pendant longtemps, les partisans des deux types de processeurs se sont affrontés sur le terrain des performances pour aboutir à une synthèse dans les ordinateurs actuels : les techniques des processeurs RISC (séquenceur câblé, pipeline, nombreux registres, cache...) ont été intégrées à tous les processeurs, tandis que les instructions CISC sont toujours présentes via un séquenceur plus compliqué, réservé à leur usage.

5. ARCHITECTURES ÉVOLUÉES

Le modèle de processeur présenté précédemment est fonctionnel mais peu efficace. Depuis une vingtaine d'années, de nombreuses techniques ont été introduites, visant à améliorer les performances et la vitesse d'exécution des processeurs. Ces avancées n'ont été possibles que grâce au progrès technologique, permettant d'intégrer toujours plus de transistors sur la puce et autorisant une complexification progressive des circuits. Parmi toutes ces techniques, nous allons rapidement en présenter quelques-unes que nous avons jugées importantes et de complexité abordable. Nous traiterons donc ici de l'introduction d'instructions spécialisées, du traitement pipeline, des processeurs superscalaires et de l'exécution multithread.

5.1 Opérateurs et instructions spécialisés

Les processeurs ont profité des possibilités offertes par l'adjonction de transistors supplémentaires pour implémenter l'exécution d'instructions arithmétiques plus compliquées. Il y a d'abord eu l'inclusion de l'UAL flottante directement dans le processeur, ce qui a permis d'effectuer rapidement des calculs sur les nombres réels, en commençant par les opérations simples puis en ajoutant progressivement des fonctions mathématiques complexes (racine carrée, fonctions trigonométriques et logarithmiques...). Devant le développement des applications multimédias, tous les fabricants de processeurs ont ensuite inclus des instructions spécifiques à ces applications dans les jeux d'instructions de leurs processeurs. Ces instructions donnent au développeur et au compilateur la possibilité d'effectuer certains calculs spécialisés en une seule opération. Ces extensions sont de deux types : des instructions mathématiques particulières et le calcul vectoriel.

De nombreuses applications de traitement du son et de l'image utilisent des algorithmes mathématiques classiques pouvant se programmer à l'aide des instructions habituelles, mais qui sont en fait généralement composés des mêmes suites d'opérations. Il est donc intéressant d'ajouter dans le processeur les circuits nécessaires pour que ces suites d'opérations, codées maintenant chacune par une instruction, puissent s'effectuer le plus rapidement possible. Prenons par exemple une instruction effectuant une multiplication et une addition ($S = X \times Y + Z$) ; cette opération peut se programmer avec les instructions standard, en deux étapes (une multiplication suivie d'une addition). L'existence d'une instruction particulière implémentant ce calcul permet une amélioration de tous les programmes fondés sur des algorithmes utilisant spécifiquement ladite opération. Ces nouvelles instructions ne sont bien sûr pas choisies au hasard, mais en fonction des algorithmes que l'on souhaite plus performants et qui les intègrent : applications multimédias, traitement d'images, etc.

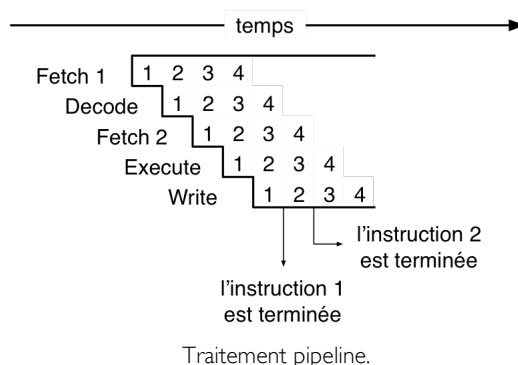
Le calcul vectoriel consiste à effectuer simultanément la même opération sur plusieurs données différentes. Ce type d'exécution est régulièrement mentionné sous le terme SIMD (*Single Instruction Multiple Data*, un seul flot d'instructions et plusieurs flots de données) ou SWP (*SubWord Parallelism*, parallélisme appliqué à des sous-mots). Cela est particulièrement utile lorsque l'algorithme utilisé applique le même traitement à toutes les données d'un tableau. C'est le cas, par exemple, dans de nombreuses applications de traitement d'images où un même calcul est fait sur tous les pixels. Pouvoir traiter ainsi trois ou quatre pixels simultanément permet une importante accélération. Pour ce faire, le processeur intègre les UAL et les registres correspondants. Là encore, l'amélioration ne concerne pas tous les programmes, mais ceux fondés sur ce schéma. Tous les processeurs actuels incluent des instructions de ce type ; il s'agit des extensions appelées MMX, SSE2, AltiVec, 3DNow!, etc.

5.2 Traitement pipeline

Chaque instruction peut être décomposée en plusieurs étapes (la lecture de l'instruction, son décodage, son exécution, etc. ; voir section 4.1), nécessitant chacune des circuits différents. Ces derniers travaillent pendant une partie du temps d'exécution, mais restent inactifs lorsque l'instruction n'est pas à l'étape concernée.

Le principe du traitement pipeline consiste à ne pas attendre la fin de l'exécution d'une instruction pour lancer la suivante, en profitant de l'inactivité des circuits ayant déjà traité l'instruction en cours. C'est l'idée du travail à la chaîne : un circuit effectue une étape d'exécution et enchaîne immédiatement la même étape avec l'instruction suivante pendant que la première avance dans la chaîne de traitement. La figure 3.15 illustre ce concept en décomposant une instruction en cinq étapes. Chaque chiffre correspond à une instruction avançant dans le pipeline et dont l'exécution se termine à sa sortie.

Figure 3.15



Le gain ne se fait pas directement sur la vitesse de traitement d'une instruction (elle doit toujours passer par les cinq étapes du pipeline) mais sur le nombre d'instructions par seconde qui sont exécutées. Dans le meilleur

des cas, le processeur peut traiter cinq fois plus d'instructions qu'un système non pipeliné. De plus, les circuits propres à chaque étape, plus simple, peuvent être optimisés pour fonctionner à une vitesse d'horloge plus rapide et accroître la vitesse du processeur. On a donc tout intérêt à fractionner le plus possible le pipeline pour augmenter le nombre d'instructions en cours d'exécution et optimiser au maximum les circuits correspondant à chaque étape. Certains processeurs ont ainsi des pipelines pouvant contenir une vingtaine voire une trentaine d'étapes.

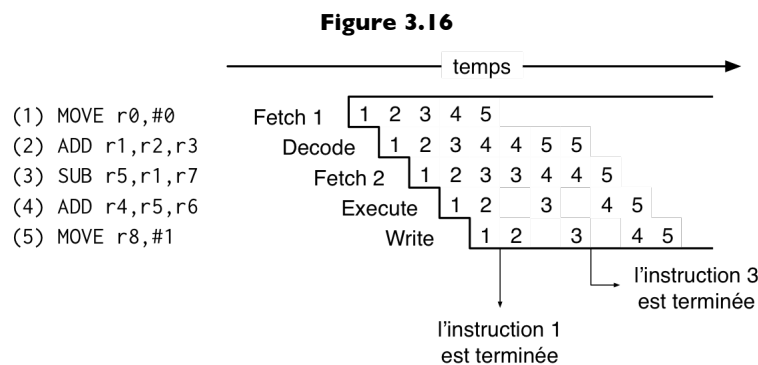
Plusieurs écueils peuvent quand même perturber cette belle organisation et freiner l'exécution des instructions.

Aléa d'étapes différenciées

Pour que le pipeline fonctionne à plein rendement, il faut décaler toutes les instructions simultanément d'une étape, ou autrement dit alimenter le pipeline continuellement (à chaque cycle) en instructions. Celui-ci doit donc être conçu de sorte que, pour toutes les instructions, chaque étape dure aussi longtemps. Malheureusement, les instructions ne sont pas équivalentes : il est plus long de récupérer une donnée en mémoire que dans un registre ; une multiplication prend plus de temps qu'une addition, etc. Des instructions demandant plus de temps pour s'exécuter et introduisant des retards dans le pipeline empêchent celles qui suivent de progresser. Une solution peut être de faire progresser les instructions suivantes en « doublant » l'instruction lente, mais cela complique la gestion du pipeline, qui doit à un moment remettre les instructions dans le bon ordre (voir section suivante).

Dépendances

Les instructions successives dans le pipeline ne sont pas indépendantes et font probablement appel aux mêmes ressources. On peut donc avoir un conflit si l'une d'entre elles a besoin d'une valeur calculée par une autre, qui la précède (pour poursuivre un calcul par exemple). Prenons l'exemple de la figure 3.16.



Dépendances dans un pipeline.

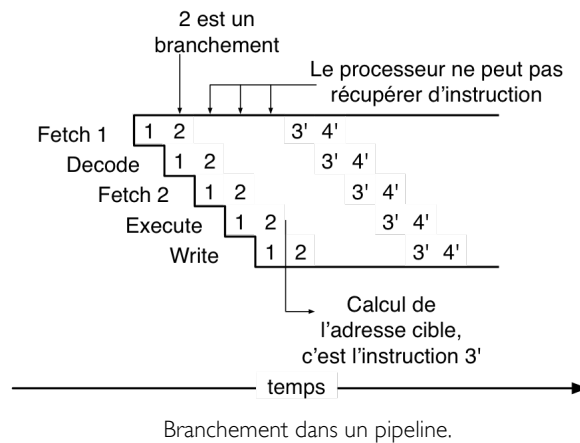
L'instruction 2 modifie le registre *r1*, utilisé (avec sa nouvelle valeur) par l'instruction 3. Celle-ci ne peut donc pas quitter l'étape de récupération des données (*Fetch 2*) avant que *r1* ne soit réécrit par l'instruction 2, c'est-à-dire lorsqu'elle passe l'étape 5 (*Write*). Lorsque le processeur modifie *r1*, il envoie en même temps la valeur à l'étape 3 pour l'instruction 3 (mécanisme de *bypass*), qui peut progresser au prochain top. L'instruction 3 est donc retardée d'un top d'horloge, induisant le retard sur tout le pipeline. Le même phénomène apparaît pour l'instruction 4, qui utilise *r5*, modifié par l'instruction 3.

Pour remédier à ce problème, le processeur peut essayer de réorganiser le flot d'instructions arrivant dans le pipeline en détectant les dépendances et en exécutant d'abord les instructions qui ne dépendent pas de résultats antérieurs. À la figure 3.16, le processeur pourrait avancer l'instruction 5, qui utilise un registre totalement indépendant des autres instructions, et profiter du « trou » créé par la 3 pour exécuter la 5. Cette technique complique fortement le séquenceur, qui doit récupérer plusieurs instructions en même temps, détecter les dépendances, réorganiser l'ordre des instructions et les exécuter, tout cela le plus rapidement possible (voir section suivante).

Branchements

À chaque instant, le processeur doit alimenter le pipeline avec les prochaines instructions à exécuter. C'est évidemment facile lorsque les instructions sont en séquence, mais cela se complique si le processeur tombe sur une rupture de séquence. Dans ce cas, il ne peut pas récupérer les nouvelles instructions avant d'avoir calculé l'adresse cible du branchement, opération effectuée normalement lors de l'étape d'exécution du pipeline. Il faut donc retarder l'exécution des instructions suivant le branchement jusqu'à ce que le processeur connaisse leur adresse (voir figure 3.17).

Figure 3.17



Dans le cas d'un branchement inconditionnel, le processeur peut accélérer le traitement en détectant la présence du saut dès le début du pipeline et en calculant immédiatement l'adresse cible. Cela lui permet de récupérer sans délai les instructions suivantes, au prix là encore de circuits supplémentaires au niveau du séquenceur.

Le problème est plus délicat avec les sauts conditionnels : on peut disposer de l'adresse cible dès la récupération de l'instruction de saut, mais on ne sait pas avant l'étape d'exécution si le saut va être effectif ou pas, car cela dépend des bits conditions qui vont être modifiés par les instructions précédentes. Il est cependant inacceptable de laisser le pipeline vide comme à la figure 3.17, car cela ralentirait fortement le processeur, d'autant plus que le nombre d'étapes est important. La seule solution est alors de faire une hypothèse sur le saut et de récupérer les instructions suivantes en conséquence : soit on suppose que le saut n'est pas réalisé et le processeur commence à exécuter les instructions qui suivent le saut en mémoire, soit on suppose qu'il va dérouter le processeur et celui-ci calcule l'adresse cible par anticipation (comme pour le saut inconditionnel) avant de récupérer les instructions correspondantes. Cette technique, appelée « prédiction de branchement », consiste à prédire au mieux le comportement du branchement avant son exécution effective.

À la fin de l'étape d'exécution du saut conditionnel, le processeur vérifie l'hypothèse, c'est-à-dire si le branchement est pris ou non. Si la prédiction est correcte, le processeur a injecté dans le pipeline les bonnes instructions et aucun retard n'est à déplorer ; si le processeur s'est trompé (prédiction fausse), il doit vider le pipeline, qui ne contient pas la suite effective du programme, pour reprendre les instructions qui sont à exécuter après le saut. Le retard est alors évidemment maximal. L'optimisation des circuits chargés de la prédiction de branchement a un impact prépondérant sur les performances du pipeline et donc du processeur. La qualité du prédicteur de branchement doit augmenter avec la taille du pipeline. En effet, plus le pipeline est long, plus le nombre de cycles perdus est important en cas de mauvaise prédiction. L'objectif est alors de limiter ce nombre de mauvaises prédictions en proposant des techniques de calcul d'hypothèse de branchement (ou de prédiction de branchement) toujours plus efficaces.

Le plus simple est de faire de la prédiction statique : suivant la direction du saut conditionnel (en avant ou en arrière dans le programme), on décide si le branchement est pris ou non. Très souvent, un retour en arrière dans le programme correspond à une fin de boucle ; comme celle-ci sera exécutée plusieurs fois, le saut sera effectué, sauf au dernier passage dans la boucle. La prédiction statique la plus classique consiste donc à dire qu'un saut arrière sera réalisé alors qu'un saut vers l'avant ne le sera pas. Bien sûr, il y a des exceptions, mais cela permet déjà d'améliorer les performances du pipeline.

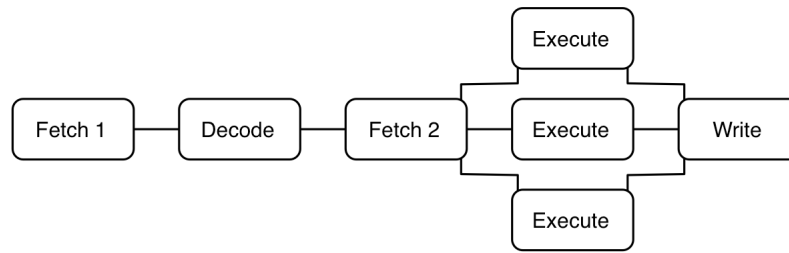
Mieux encore, le processeur peut faire de la prédiction dynamique : un historique est associé à chaque saut et permet de savoir si celui-ci a le plus souvent été effectif ou non. On peut ainsi atteindre un pourcentage élevé de prédictions correctes, permettant de profiter au maximum du pipeline. Les processeurs actuels disposent ainsi de schémas de prédiction de branchement complexes à plusieurs niveaux. L'objectif est d'atteindre des taux de bonnes prédictions allant de 95 % à 99 %.

5.3 Processeur superscalaire

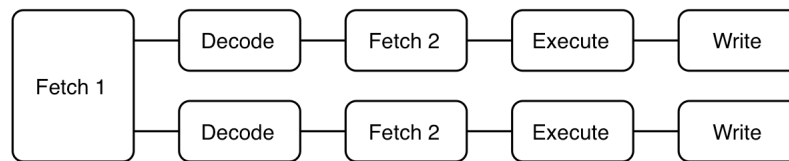
Un pipeline permet de traiter plus d'instructions par cycle mais chacune d'elles se trouve à une étape de traitement différente. Pour pouvoir traiter plusieurs instructions en même temps (on parle alors d'exécution parallèle), les processeurs superscalaires ont été dotés de multiples unités d'exécution dans leur pipeline, voire de plusieurs pipelines complets (voir figure 3.18). L'objectif de ces processeurs consiste alors à alimenter,

à chaque cycle, chaque unité d'exécution. Dans ces conditions, le processeur délivre le maximum de sa puissance puisque aucune unité d'exécution ne reste inoccupée durant les cycles.

Figure 3.18



Plusieurs unités d'exécution



Plusieurs pipelines

Processeur superscalaire.

Avoir plusieurs circuits chargés de l'exécution permet de traiter une instruction compliquée (prenant beaucoup de temps), comme une addition flottante, sans bloquer le reste du pipeline, qui peut passer par les autres unités d'exécution. Il faut pour cela que les instructions successives n'aient pas de liens de dépendance et ne fassent pas appel aux mêmes circuits. On pourra ainsi probablement exécuter en parallèle une opération arithmétique entière et une opération flottante qui n'utilisent pas la même UAL et ne font pas référence aux mêmes registres. De plus, la duplication des unités d'exécution permet de dédier ces dernières au traitement d'instructions du même type (par exemple, une unité d'exécution entière pour toutes les instructions manipulant des données entières, une unité d'exécution flottante pour la manipulation des données flottantes, une unité d'accès mémoire pour les lectures et écritures de données en mémoire...). Cette spécialisation des unités d'exécution permet d'espérer un accroissement des performances et une meilleure utilisation globale du processeur.

Exécution dans le désordre

Si le séquenceur est construit autour de plusieurs pipelines, il peut exécuter complètement plusieurs instructions en parallèle, chacune utilisant un pipeline. Bien sûr, on a toujours les problèmes liés aux dépendances et à l'accès aux mêmes ressources (UAL, registres...), obligeant le processeur à retarder temporairement l'une des instructions ou à intervertir leur ordre d'exécution.

Pour une utilisation optimale des différents pipelines, l'ordonnancement des instructions est déterminé dynamiquement lors de leur chargement. Un buffer de préchargement permet de récupérer plusieurs instructions simultanément dans le cache de niveau 1, laissant un stock d'instructions à disposition de la logique du processeur, à charge pour celle-ci de déterminer l'ordre dans lequel elles vont être exécutées. Cela implique une gestion très lourde de leur exécution : quelles sont celles qui vont effectivement être exécutées (si elles sont derrière un branchement, peut-être celui-ci sera-t-il pris et qu'elles seront ignorées), quels registres vont être utilisés et modifiés, et dans quel ordre (une dépendance, liée à l'utilisation d'un même registre dans deux instructions, peut parfois être éliminée par un renommage du registre dans l'une des instructions, renommage effectué par le séquenceur au moment de l'exécution), quelle instruction peut être lancée à un instant précis (en fonction de la disponibilité des unités d'exécution) ?

Ce mode de fonctionnement, avec le traitement pipeline, est à l'origine de la plus grande partie de l'augmentation des performances des processeurs depuis les années 1990, au prix d'une importante complexité de la logique interne du composant.

Processeur VLIW

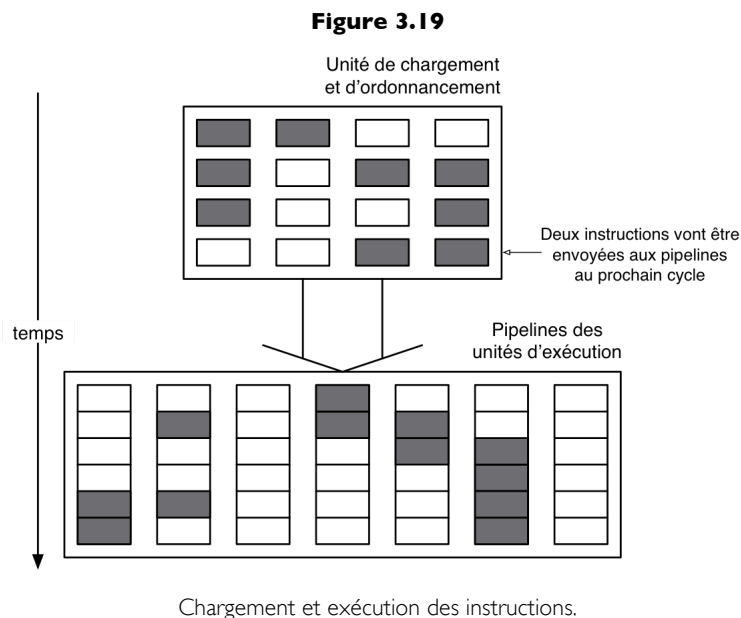
Dans le mode d'exécution précédent, toute la complexité se trouve dans le processeur, qui doit déterminer le parallélisme de l'application (la possibilité d'exécuter simultanément des instructions dans chaque unité fonctionnelle) dynamiquement lors de son exécution. Certains processeurs rejettent le travail sur le

compilateur en l'obligeant à définir lui-même l'ordre d'exécution des instructions. Leur objectif, à l'instar du modèle superscalaire, est de mettre à disposition plusieurs unités d'exécution et de faire en sorte que chacune ait une instruction à exécuter à chaque cycle. En ce sens, les instructions sont regroupées par paquets, qui constituent ce que l'on appelle des instructions VLIW (*Very Long Instruction Word*). On peut prendre l'exemple de processeurs ayant des instructions de 128 bits, composées de quatre instructions élémentaires, chacune de 32 bits. Dans le modèle d'exécution VLIW, le processeur récupère les instructions une par une et dispatche chaque instruction élémentaire dans une unité fonctionnelle différente pour qu'elles soient toutes exécutées simultanément. Le processeur n'a plus à s'interroger sur leur ordre d'exécution, mais laisse le compilateur extraire lui-même statiquement le parallélisme lors de la compilation. Le contrôle du processeur en est donc largement simplifié. Ce modèle est plus largement répandu dans les processeurs spécifiques au domaine du traitement des images et du signal (DSP, *Digital Signal Processor*).

5.4 Exécutions parallèles

Un processeur superscalaire est en quelque sorte composé de deux blocs fonctionnels : l'unité de chargement et d'ordonnancement des instructions, et les unités d'exécution. La première est chargée de récupérer les instructions en mémoire et de les réorganiser pour essayer, à chaque cycle, d'en envoyer le maximum aux unités d'exécution. Elle essaie d'extraire le parallélisme du programme en fonction de la durée d'exécution de chacune des instructions et des dépendances entre elles. Le second bloc est formé des différents pipelines d'exécution, chargés respectivement des calculs sur les entiers, les nombres flottants, ou encore du calcul des adresses lors des accès mémoire, etc.

Les performances optimales sont bien sûr obtenues lorsque toutes les ressources du processeur sont utilisées au maximum. Malheureusement, cela est rarement le cas. En raison du parallélisme limité des programmes, l'unité d'ordonnancement ne peut pas envoyer, à chaque cycle, le maximum d'instructions possible aux unités d'exécution (celles-ci étant de plus en plus nombreuses au sein des processeurs). Elle émet en moyenne, pour des programmes classiques, deux instructions par cycle, alors que les unités d'exécution sont souvent plus nombreuses (il est parfois possible de traiter plus de deux instructions simultanément). De même, les unités d'exécution ne sont pas occupées à 100 % : comme elles sont spécifiques à certains types d'instructions, elles peuvent être vides si ces instructions ne sont pas présentes. Les pipelines peuvent également avoir des retards et des trous si certaines instructions demeurent plus d'un cycle à une étape en raison d'une difficulté d'exécution. Cela est résumé à la figure 3.19, où les cases vides symbolisent les ressources inexploitées lors d'un cycle.

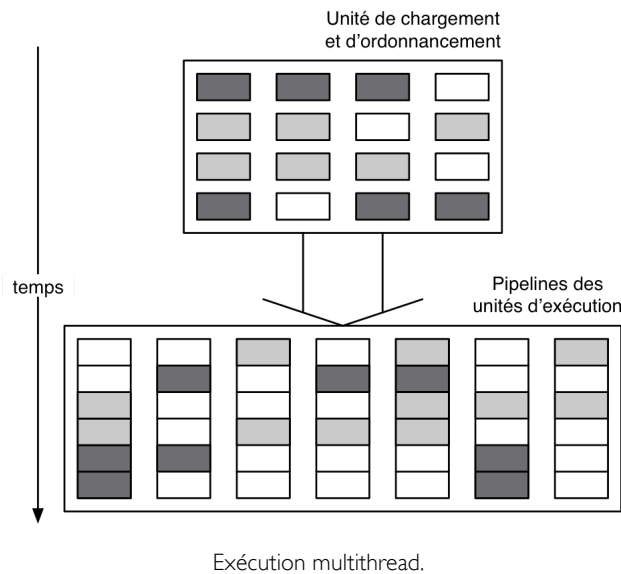


Exécution multithread

Un ordinateur a toujours plusieurs programmes en cours d'exécution, appelés « processus » ou « threads ». Le système d'exploitation est chargé de les ordonner en découpant le temps en tranches (d'une centaine de millisecondes le plus souvent) et en attribuant ces tranches successivement à chacun des threads. Le processeur exécute alors les instructions d'un thread pendant un temps relativement long, puis s'occupe des autres processus avant de revenir au premier.

Pour augmenter l'occupation des ressources internes du processeur (et donc améliorer ses performances en exécutant plus d'instructions par cycle), on a essayé d'exécuter consécutivement plusieurs threads. À chaque cycle, le processeur tente de dédier ses ressources (unité d'ordonnancement et unités d'exécution) aux instructions d'un seul thread. Mais à chaque nouveau cycle (ou, suivant les implémentations, après quelques cycles), le processeur change de thread et, en exécutant les instructions de ce dernier, il est ainsi possible de compenser un retard dans un pipeline. De son côté, l'unité d'ordonnancement envoie le maximum d'instructions du thread traité aux unités d'exécution (voir figure 3.20). Les performances sont améliorées car certaines dépendances disparaissent naturellement à l'exécution (les instructions d'un même thread sont plus « espacées » dans le pipeline) et il en résulte une meilleure utilisation globale des unités d'exécution. De plus, l'impact des retards dus aux accès mémoire (voir chapitres 5 et 6) est diminué (au lieu d'attendre, le processeur exécute quelques instructions d'un autre thread).

Figure 3.20

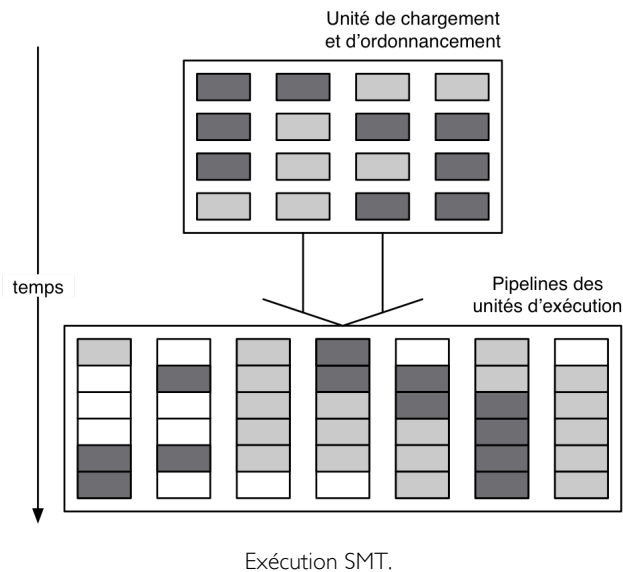


Cependant, il reste encore des trous car le processeur ne peut pas combiner deux threads dans un même cycle (chaque ligne horizontale contient des instructions d'un seul thread). Si les deux ont un faible parallélisme, peu d'instructions sont exécutées à chaque cycle.

Exécution multithread simultanée

En supprimant la contrainte d'un seul thread par cycle, on peut maximiser l'utilisation du processeur : c'est l'exécution SMT (*Simultaneous MultiThreading*) ou hyperthreading (nom commercial donné par le fabricant Intel). À chaque cycle, le processeur essaie d'exécuter le maximum d'instructions possible, indépendamment de leur provenance (voir figure 3.21). Il y a encore moins de trous dans l'unité d'ordonnancement et les pipelines : le processeur exécute encore plus d'instructions par cycle.

Figure 3.21



L'accroissement en complexité du séquenceur (les instructions de threads différents ne doivent pas interférer entre elles, par exemple pour l'utilisation des registres) est largement compensé par l'augmentation des performances : sur les processeurs d'Intel, l'implémentation de l'hyperthreading augmente de 5 % la surface de la puce pour un gain de vitesse d'environ 30 %.

Pour que le processeur puisse lancer deux threads simultanément par hyperthreading, il faut informer le système d'exploitation de cette possibilité, en lui présentant le processeur embarquant cette technologie comme deux (ou plus) processeurs logiques. Le système tentera d'exécuter deux threads (ou plus, un sur chaque « processeur »), qui seront traités simultanément par un seul processeur physique.

Processeurs multicœurs

Les processeurs les plus récents poussent la logique encore plus loin en dupliquant leur cœur : unité d'ordonnancement, unités d'exécution et souvent le cache de niveau 1 (voir chapitre 6). On les dit « dual-core », ou « multicore » s'il y en a plus que deux. On retrouve la possibilité d'exécuter deux threads simultanément, sans la complexité de l'hyperthreading, puisque les deux chemins d'exécution sont physiquement séparés.

RÉSUMÉ

Un ordinateur contient un processeur qui exécute en séquence des instructions de transfert ou de calcul se trouvant en mémoire principale. Il autorise également les ruptures de séquence, ce qui permet d'avoir un comportement différent suivant les données d'un programme, en agissant directement sur le compteur ordinal. Ces instructions sont traitées par un séquenceur, chef d'orchestre interne, qui envoie des commandes à l'unité arithmétique et logique et aux registres du processeur. Ce séquenceur, plus ou moins compliqué, est le composant qui a connu le plus de perfectionnements ces dernières années (introduction d'un pipeline, processeur superscalaire, exécution multithread simultanée), permettant un accroissement sans précédent des performances et surtout de la rapidité d'exécution des instructions.

Problèmes et exercices

Les deux premiers exercices portent sur la structure des programmes et de l'ordinateur. Les suivants abordent les caractéristiques de la conception et de la programmation d'un processeur. Nous allons nous intéresser aux types et formats des instructions (nombre de données, modes d'adressage), voir comment se placent les ruptures de séquence dans le cadre de l'implémentation des structures de contrôle, et illustrer l'utilisation du registre d'état et du registre pointeur de pile. Pour finir, nous étudierons les performances d'un pipeline en lien avec les branchements et la compilation du code.

Exercice 1 : Compilation ou interprétation

Vous écrivez un programme qui peut être soit compilé, soit interprété. Le compilateur peut compiler dix mille lignes de code par seconde, puis il nécessite deux secondes pour l'édition des liens. Votre programme exécutable final contient vingt fois plus de lignes d'instructions processeur que le programme de départ en langage évolué et s'exécute à la vitesse de cent mille instructions par seconde.

Pour l'exécuter, vous avez aussi le choix de passer par un interpréteur travaillant à la vitesse de cent mille instructions par seconde. Malheureusement, l'interprétation est compliquée et nécessite deux cents fois plus d'instructions que votre programme de départ.

- 1) Une fois le programme écrit (mille lignes de code), en combien de temps s'exécute-t-il s'il est compilé (temps de compilation inclus) ? Et s'il est interprété ?
- 2) Quels sont les deux temps d'exécution s'il contient un million de lignes de code ?
- 3) Quand doit-on choisir la compilation plutôt que l'interprétation, ou l'inverse ?

1) Le compilateur met $\frac{1}{10}$ de seconde pour compiler (mille lignes de code à dix mille lignes par seconde), 2 secondes pour l'édition des liens et génère un exécutable de vingt mille instructions. Celui-ci s'exécute en 0,2 seconde, ce qui fait un total de 2,3 secondes.

Si l'on interprète le programme, il faut deux cents mille instructions, ce qui prend 2 secondes d'exécution.

2) Avec un million de lignes de code, le compilateur nécessite 100 secondes, plus 2 secondes pour l'édition des liens. Le programme final fait vingt millions d'instructions (soit 200 secondes d'exécution) et le temps total d'exécution est donc de 302 secondes.

L'interprétation du programme coûte deux cents millions d'instructions et s'exécute en 2 000 secondes.

3) L'exécution d'un programme compilé est beaucoup plus rapide que son interprétation mais il faut tenir compte du temps de compilation, surtout si le code source doit subir de nombreuses modifications.

Ici, pour un programme contenant N lignes de code, le temps total de compilation est de $\frac{N}{10^4} + 2 + \frac{20N}{10^5}$ alors que

le temps d'interprétation est de $\frac{200N}{10^5}$. L'égalité est obtenue lorsque $N = \frac{2 \cdot 10^5}{170} \approx 1176$. Si le programme est plus petit, il est plus rapide de l'interpréter. Sinon, il vaut mieux le compiler.

Exercice 2 : Débit d'informations

- 1) Un processeur exécute une instruction de 4 octets toutes les nanosecondes. De plus, 20 % de ces instructions font référence à une donnée de 4 octets en mémoire. Quel est le débit nécessaire entre le processeur et la mémoire ?

2) Une carte vidéo doit afficher vingt-quatre images par seconde sur un écran de $1\,600 \times 1\,200$ pixels, un pixel étant une combinaison de trois couleurs, chacune codée sur 8 bits. Quel est le débit nécessaire entre la carte vidéo et le moniteur ?

1) Une instruction toutes les nanosecondes équivaut à 1 milliard d'opérations par seconde, soit 4 milliards d'octets à transférer par seconde. La recherche des données de l'instruction ajoute 20 % à ce chiffre, ce qui donne un débit nécessaire de 4,8 Go/s entre la mémoire et le processeur. Ce chiffre énorme est rarement atteint par les systèmes actuels, ce qui explique le recours à la mémoire cache.

2) Il faut bien sûr multiplier tous les chiffres pour avoir le débit total : $24 \times 1\,600 \times 1\,200 \times 3 \times 8 = 1\,105\,920\,000$ bit/s ≈ 138 Mo/s de débit entre la carte vidéo et le moniteur.

Le débit entre le processeur et la carte graphique est variable car il dépend de l'image affichée et de sa complexité (nombres d'images par seconde, deux ou trois dimensions, textures, formes, couleurs...). En effet, ce n'est pas une image brute que le processeur envoie à la carte mais des commandes d'affichage décrivant l'image et ses composants. Plus ceux-ci sont compliqués, plus le processeur doit envoyer d'ordres à la carte graphique et plus le débit d'informations est important.

Exercice 3 : Décalages et multiplication

1) Un décalage à gauche correspond à une multiplication par 2. Comment effectuer une multiplication par 5 avec des décalages et des additions ? Et une multiplication par 7 ?

2) Comment déterminer l'ordre des décalages et des additions à partir de l'écriture binaire du multiplicateur ?

1) Il faut décomposer la multiplication par 5 en multiplications par 2 et en additions :

$$5 \times A = 2 \times 2 \times A + A$$

Il faut donc stocker la valeur initiale dans un registre temporaire, décaler deux fois vers la gauche le registre qui contient la valeur, et ajouter la valeur initiale mémorisée précédemment.

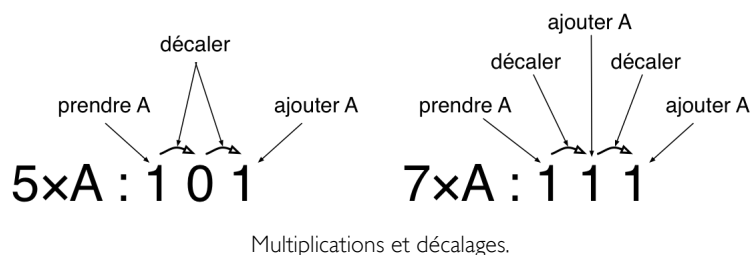
Pour une multiplication par 7, le calcul est similaire :

$$7 \times A = (2 \times A + A) \times 2 + A$$

Là encore, il faut en premier stocker la valeur initiale dans un registre temporaire pour pouvoir la réutiliser, décaler une fois vers la gauche le registre contenant la valeur initiale, ajouter cette valeur, et de nouveau, décaler le registre et ajouter la valeur initiale.

2) Il faut utiliser l'écriture binaire du multiplicateur et parcourir les bits de la gauche vers la droite. Si le bit est à 1, on ajoute la valeur initiale et on décale la valeur courante en passant au bit suivant (voir la figure 3.22 pour les valeurs 5 et 7).

Figure 3.22



On peut ainsi implémenter un algorithme pour simuler la multiplication en utilisant uniquement des instructions de décalage et d'addition, qui s'exécutent rapidement sur un processeur.

Exercice 4 : Sauts et contrôle

Un processeur possède une instruction de branchement conditionnel qui compare une variable à 0 pour décider s'il doit sauter à une autre adresse ou non : Si $A=0$, aller en..., Si $A \neq 0$, aller en..., ou les inégalités (inférieur, inférieur ou égal...) par rapport à zéro.

Mettez sous une forme linéaire (voir figure 3.7) les structures de contrôle classiques en langage évolué :

```

while (i≠0) { ... }
do { ... } while (i=0)
for (i=0; i<max; i++) { ... }
if (i=0) { ... } else { ... }

```

1) Il faut commencer par faire le test et sauter la boucle si la variable i est nulle ; sinon il faut exécuter la boucle et revenir en arrière sur le test.

```

A: Si i=0, aller en B
...
corps de la boucle
...
aller en A
B: ... suite du programme

```

2) Le test se fait maintenant à la fin et, s'il est vrai, on revient en arrière sur la boucle.

```

A: ...
corps de la boucle
...
Si i=0, aller en A
... suite du programme

```

3) La variable i s'initialise et s'incrémente respectivement avant et à la fin de la boucle, et, de son côté, le test s'effectue avant car, s'il est tout de suite faux, il ne faut pas exécuter la boucle. Comme le test consiste uniquement à vérifier si une variable est nulle, il faut utiliser une variable temporaire égale à la différence entre i et max ; quand elle est positive ou nulle, la boucle est finie.

```

i = 0
A: temp = i - max
Si temp>=0, aller en B
...
corps de la boucle
...
i++
aller en A
B: ... suite du programme

```

4) C'est la structure la plus simple : si le test est faux, on saute ; sinon on poursuit l'exécution, sans oublier de sauter le corps du else.

```

Si i≠0, aller en A
...
corps du if
...
aller en B
A: ...
corps du else
...
B: ... suite du programme

```

Exercice 5 : Instructions à une, deux ou trois données

Un processeur possède les registres r0 à r9. On veut calculer l'expression

$$((r1 \times r2 - r3) / (r1 + r4 + r5)) + r6 + r2$$

et mettre le résultat dans r0.

1) Le processeur possède des instructions à trois données qui sont :

```

ADD rX, rY, rZ (rX ← rY+rZ)
SUB rX, rY, rZ (rX ← rY-rZ)
MUL rX, rY, rZ (rX ← rY×rZ)
DIV rX, rY, rZ (rX ← rY/rZ)
MOVE rX, rY (rX ← rY)

```

Écrivez la suite d'instructions correspondant au calcul voulu.

2) Le processeur possède des instructions à deux données qui sont :

```

ADD rX,rY  (rX ← rX+rY)
SUB rX,rY  (rX ← rX-rY)
MUL rX,rY  (rX ← rX×rY)
DIV rX,rY  (rX ← rX/rY)
MOVE rX,rY (rX ← rY)

```

Écrivez la suite d'instructions correspondant au calcul voulu.

3) Le processeur possède un registre accumulateur qui est source et destination de toutes les opérations. Les instructions à une donnée sont :

```

ADD rX    (Acc ← Acc+rX)
SUB rX    (Acc ← Acc-rX)
MUL rX    (Acc ← Acc×rX)
DIV rX    (Acc ← Acc/rX)
LOAD rX   (Acc ← rX)
STORE rX  (rX ← Acc)

```

Écrivez la suite d'instructions correspondant au calcul voulu. On veut toujours avoir le résultat dans *r0*, qui n'est pas l'accumulateur.

1) On construit progressivement l'expression en stockant les calculs intermédiaires dans le registre *r0*.

```

MUL r0,r1,r2 ; r0 ← r1×r2
SUB r0,r0,r3  ; r0 ← (r1×r2)-r3
ADD r1,r1,r4  ; r1 ← r1+r4
ADD r1,r1,r5  ; r1 ← (r1+r4)+r5
DIV r0,r0,r1  ; r0 ← (r1×r2-r3)/(r1+r4+r5)
ADD r0,r0,r6  ; r0 ← ((r1×r2-r3)/(r1+r4+r5))+r6
ADD r0,r0,r2  ; r0 ← ((r1×r2-r3)/(r1+r4+r5)+r6)+r2

```

L'avantage d'une instruction à trois données apparaît dès le début, quand il est possible de multiplier *r1* et *r2* sans modifier ces registres pour pouvoir réutiliser leurs valeurs.

2) Ici, il n'est plus possible de multiplier directement *r1* et *r2* dès le début car cela changerait la valeur d'un des deux registres, valeur indispensable pour la suite du calcul. Il faut donc d'abord transférer l'une des valeurs dans *r0* pour initier le calcul.

```

MOVE r0,r1 ; r0 ← r1
MUL r0,r2 ; r0 ← (r1)×r2
SUB r0,r3 ; r0 ← (r1×r2)-r3
ADD r1,r4 ; r1 ← r1+r4
ADD r1,r5 ; r1 ← (r1+r4)+r5
DIV r0,r1 ; r0 ← (r1×r2-r3)/(r1+r4+r5)
ADD r0,r6 ; r0 ← ((r1×r2-r3)/(r1+r4+r5))+r6
ADD r0,r2 ; r0 ← ((r1×r2-r3)/(r1+r4+r5)+r6)+r2

```

3) Puisque toutes les opérations se font avec l'accumulateur, il faut calculer séparément le dividende et le diviseur. Si l'on commence par le dividende, il faudra le mettre de côté dans un autre registre avant de passer au diviseur, puis le recharger avant d'effectuer la division. Il est plus judicieux de démarrer par le diviseur, de le stocker dans un autre registre et de passer au dividende, qui sera ainsi déjà dans l'accumulateur pour la division.

```

LOAD r1 ; Acc ← r1
ADD r4 ; Acc ← (r1)+r4
ADD r5 ; Acc ← (r1+r4)+r5
STORE r7 ; r7 ← (r1+r4+r5)
LOAD r1 ; Acc ← r1
MUL r2 ; Acc ← (r1)×r2
SUB r3 ; Acc ← (r1×r2)-r3
DIV r7 ; Acc ← (r1×r2-r3)/(r1+r4+r5)
ADD r6 ; Acc ← ((r1×r2-r3)/(r1+r4+r5))+r6
ADD r2 ; Acc ← ((r1×r2-r3)/(r1+r4+r5)+r6)+r2
STORE r0 ; r0 ← ((r1×r2-r3)/(r1+r4+r5)+r6+r2)

```

On voit que disposer de plusieurs données dans les instructions permet de réduire le nombre d'instructions du programme car on peut travailler avec tous les registres à la fois pour le calcul et le stockage. Pour cette raison, les processeurs à accumulateur n'ont pas existé très longtemps.

Exercice 6 : Divers modes d'adressage

Pour accéder à une donnée en mémoire, un processeur ne possède que le mode d'adressage indirect : `MOVE r1,(r2)`, qui récupère la donnée se trouvant à l'adresse mémoire indiquée dans `r2` et la met dans `r1`. On souhaite simuler les autres modes d'adressage accédant à la mémoire. En distinguant à chaque fois deux cas (instruction à deux ou trois données), écrivez les instructions correspondant aux exemples suivants utilisant d'autres modes d'adressage :

- 1) Adressage indirect avec déplacement : `MOVE r1,8(r2)`.
- 2) Adressage indirect avec pré-incrémentation : `MOVE r1,(r2)+`.
- 3) Adressage indirect avec post-décrémentation : `MOVE r1,-(r2)`.
- 4) Adressage indirect indexé : `MOVE r1,(r2,r3)`.
- 5) Adressage indirect indexé avec déplacement : `MOVE r1,8(r2,r3)`.

1) Il faut ajouter 8 à `r2` pour pointer sur la case mémoire correcte. Mais si l'on additionne directement dans `r2`, il faut retrancher 8 pour retomber sur la valeur initiale (car la valeur de `r2` n'est pas modifiée dans l'adressage indirect avec déplacement). Si les instructions ont deux données, cela donne :

```
ADD r2,#8      ; modification de r2
MOVE r1,(r2)
SUB r2,#8      ; remise de r2 à sa valeur initiale
```

Si les instructions ont trois données, on peut prendre un autre registre inutilisé pour pointer en mémoire :

```
ADD r3,r2,#8    ; r3 ← r2+8
MOVE r1,(r3)
```

2) Il suffit d'ajouter 1 à `r2` après l'accès mémoire :

```
MOVE r1,(r2)
ADD r2,r2,#1 ou ADD r2,#1
```

3) Il suffit de retrancher 1 à `r2` avant l'accès mémoire :

```
SUB r2,r2,#1 ou SUB r2,#1
MOVE r1,(r2)
```

4) Comme pour l'adressage indirect avec déplacement, il ne faut pas modifier `r2` et `r3` (ou remettre les valeurs initiales après l'accès mémoire). Pour des instructions à deux données, on a :

```
ADD r2,r3      ; modification de r2
MOVE r1,(r2)
SUB r2,r3      ; remise de r2 à sa valeur initiale
```

Si trois données sont autorisées, on peut passer par un registre intermédiaire (inutilisé) :

```
ADD r4,r2,r3    ; r4 ← r2+r3
MOVE r1,(r4)
```

5) C'est le plus compliqué puisque deux additions sont nécessaires pour calculer l'adresse mémoire de la donnée. Il est là toujours plus intéressant de passer par un registre intermédiaire au lieu de remettre `r2` ou `r3` à sa valeur initiale (car cela nécessiterait deux soustractions après l'accès mémoire). Pour des instructions à deux données, on a :

```
MOVE r4,r2      ; r4 ← r2
ADD r4,r3      ; r4 ← (r2)+r3
ADD r4,#8      ; r4 ← (r2+r3)+8
MOVE r1,(r4)
```

On gagne une instruction avec trois données :

```
ADD r4,r2,r3    ; r4 ← r2+r3
ADD r4,r4,#8    ; r4 ← (r2+r3)+8
MOVE r1,(r4)
```

On voit bien que la multiplication des modes d'adressage différents permet d'économiser des instructions, ce qui était recherché dans les années 1970-1980, à cause de la faible capacité mémoire des ordinateurs. Malheureusement, cela se payait par une forte complication du séquenceur, empêchant d'accélérer le décodage des instructions.

Exercice 7 : Registre d'état et comparaison

Un registre d'état a les bits Z (Zero), C (Carry), N (Negative) et V (oVerflow). On souhaite trouver une expression logique de ces bits indiquant une inégalité. Pour ce faire, on introduit une instruction de comparaison `CMP rX, rY`, qui effectue la soustraction $rX - rY$, positionne les bits du registre d'état (en fonction du résultat), mais ne mémorise pas le résultat de la soustraction.

- 1) Peut-il y avoir débordement (et dans quels cas) lors du calcul $rX - rY$ en fonction des signes de rX et rY ?
- 2) S'il n'y a pas débordement, quel bit du registre d'état permet de savoir, après l'instruction de comparaison, que $rX \geq rY$? Donnez une expression logique vraie s'il n'y a pas débordement et que $rX \geq rY$.
- 3) S'il y a débordement, quel est le signe de $rX - rY$ si $rX < rY$? Quel est le signe de $rX - rY$ si $rX \geq rY$? Donnez une expression logique vraie s'il y a débordement et que $rX \geq rY$.
- 4) Quelle est l'expression logique, combinaison des bits du registre d'état, qui permet de tester si $rX \geq rY$ après une instruction de comparaison ?
- 5) Quelle expression semblable permet de tester si $rX < rY$ après une instruction de comparaison ?

1) Si rX et rY sont de même signe, il ne peut y avoir débordement car :

- si rX et rY sont positifs, on a l'inégalité $-rY \leq rX - rY \leq rX$.
- si rX et rY sont négatifs, on a l'inégalité $-rY \geq rX - rY \geq rX$.

Dans les deux cas, le résultat est représentable.

Si rX et rY sont de signes contraires, il peut y avoir débordement.

2) S'il n'y a pas débordement, cela signifie que le signe du résultat de la soustraction $rX - rY$ est correct, autrement dit qu'il est positif si $rX \geq rY$. Ce signe est donné par le bit N, qui doit donc être à 0. L'expression $\overline{V} \cdot \overline{N}$ est donc vraie s'il n'y a pas débordement et que $rX \geq rY$.

3) Si $rX < rY$, le résultat de la soustraction est forcément négatif. Mais s'il y a débordement, cela signifie que le résultat n'est pas représentable parce qu'il est positif lorsqu'on le lit en complément à 2. L'inverse est également vrai si $rX \geq rY$: le résultat devrait être positif mais se lit comme un nombre négatif (et donc le bit N du registre d'état est à 1). L'expression $V \cdot N$ est donc vraie s'il y a débordement et que $rX \geq rY$.

4) On est soit dans le premier cas, soit dans le deuxième. L'expression finale est donc le OU des deux expressions précédentes : $rX \geq rY \Leftrightarrow N \cdot V + \overline{N} \cdot \overline{V} = 1$

5) $rX < rY$ est l'inverse de $rX \geq rY$. Il faut donc prendre le complémentaire de $N \cdot V + \overline{N} \cdot \overline{V}$. Une simple application des lois de De Morgan donne :

$$\overline{N \cdot V + \overline{N} \cdot \overline{V}} = \overline{N \cdot V} \cdot \overline{\overline{N} \cdot \overline{V}} = (\overline{N} + \overline{V}) \cdot (N + V) = N \cdot \overline{V} + \overline{N} \cdot V$$

Exercice 8 : Pile et paramètre de fonction

Un processeur gère les appels de fonction à l'aide d'une pile en mémoire pour sauvegarder l'adresse de retour (comme à la figure 3.13). Le programmeur peut également opérer lui-même sur la pile à l'aide des instructions `POP rX`, qui met dans rX la valeur au sommet de la pile, et `PUSH rX`, qui empile rX .

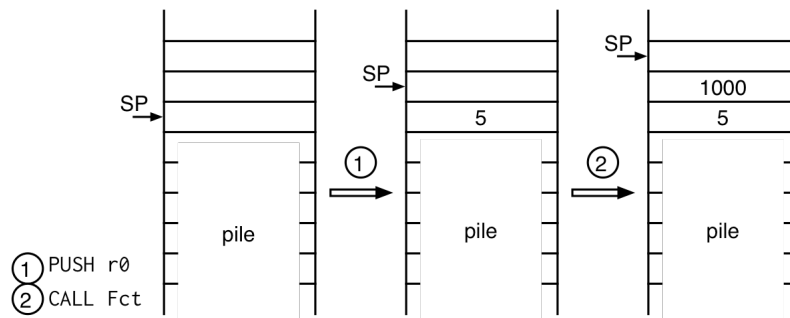
On souhaite utiliser la pile pour passer un paramètre à la fonction. Pour ce faire, on empile la valeur du registre $r0$ (censé contenir le paramètre) avant d'appeler la fonction :

```
PUSH r0
CALL Fct
```

- 1) En supposant que $r0$ vaut 5 et que l'adresse de retour soit 1000, dessinez le contenu de la pile au moment où le processeur entre dans la fonction.
- 2) Quelle séquence d'instructions doit se trouver au début de la fonction pour que l'on puisse récupérer le paramètre dans un registre ?

1) La première instruction empile la valeur de $r0$ (soit 5) et la seconde empile la valeur de PC, c'est-à-dire l'adresse de retour (1000). On a donc ces deux valeurs au sommet de la pile (voir figure 3.23).

Figure 3.23



Appel de fonction avec paramètre.

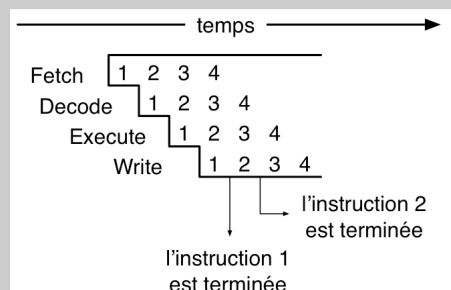
2) Il faut dépiler le paramètre, mais celui-ci ne se trouvera pas au sommet de la pile car c'est la place de l'adresse de retour. Il faut donc d'abord dépiler celle-ci avant de dépiler le paramètre, puis remettre l'adresse de retour sur la pile pour pouvoir revenir dans le programme principal à la fin de la fonction. Les premières instructions de la fonction sont donc les suivantes :

```
POP r9 ; dépiler l'adresse de retour dans un registre (r9 par exemple)
POP r0 ; dépiler le paramètre dans le registre voulu
PUSH r9 ; remettre l'adresse de retour sur la pile
```

Exercice 9 : Pipeline et branchements

Un pipeline interne du processeur est composé de quatre étapes : le chargement de l'instruction (*Fetch*), son décodage (*Decode*), son exécution (*Execute*) et l'écriture du résultat (*Write*). Chaque étape est effectuée en un cycle. À chaque cycle, le processeur charge une nouvelle instruction dans le pipeline en décalant les trois autres encore dedans. Ainsi, la première instruction d'un programme est complètement exécutée en quatre cycles puis l'exécution d'une nouvelle instruction est terminée à chaque cycle (voir l'exemple à la figure 3.24).

Figure 3.24



Exemple de pipeline.

Dans le cas d'une instruction de saut, l'instruction suivante ne peut être chargée qu'une fois son adresse connue. Dans le cas d'un saut inconditionnel, celle-ci est immédiatement disponible (dès l'étape *Fetch*, le processeur reconnaît l'instruction de saut et charge immédiatement l'adresse cible) et n'entraîne pas de retard dans le pipeline. Par contre, dans le cas d'un saut conditionnel, il faut avoir calculé la condition pour savoir si elle est vraie ou fausse et donc si l'adresse de l'instruction suivante est celle qui suit le saut (si la condition est fausse) ou si c'est celle donnée dans l'instruction de saut. Ce calcul de la condition se fait à l'étape *Execute* du pipeline et l'écriture dans le registre PC se fait à l'étape *Write*, ce qui permet au cycle suivant de récupérer la prochaine instruction.

1) On suppose qu'une instruction de saut conditionnel arrive dans le pipeline au cycle 0. Donnez l'état du pipeline pendant quelques cycles après l'arrivée de l'instruction de saut pour les deux cas de la condition (vraie et fausse). Quel est le retard induit par cette instruction de saut ?

2) Pour éviter ce retard, on implémente un mécanisme de prédiction de branchement. Plus précisément, si une instruction de saut conditionnel arrive dans le pipeline au cycle 0, on effectue immédiatement une prédiction de branchement : on fait une hypothèse sur la valeur de la condition et on commence dès le cycle suivant à charger les instructions qui seront exécutées si la prédiction est correcte. Bien sûr, si la prédiction est erronée (on s'en aperçoit lors du calcul effectif de la condition,

c'est-à-dire lorsque l'instruction de saut est à l'étape d'exécution), il faut vider tout de suite le pipeline et attendre que l'adresse de la prochaine instruction soit connue (que PC soit modifié à l'étape Write) pour la charger au cycle suivant. Donnez l'état du pipeline pendant quelques cycles après l'arrivée de l'instruction de saut pour les deux cas de la prédiction (correcte et erronée). Quel est le retard induit par cette instruction de saut dans chacun des cas ?

3) Les instructions C suivantes :

```
for (i=0; i<a; i++)
    t[i]=0;
```

ont été compilées comme suit (chaque ligne correspond à une instruction processeur) :

```
i=0
ici:  comparer i et a
      Si i >= a, aller en suite
      t[i]=0
      i++
      saut inconditionnel à ici
suite: ...
```

Combien de fois l'instruction de saut conditionnel est-elle exécutée ? Combien de fois le branchement est-il pris et combien de fois n'est-il pas pris ? À l'instant 0, le pipeline est vide et on commence à charger l'instruction i=0. En combien de cycles le programme a-t-il fini de s'exécuter si :

- le mécanisme de prédiction de branchement considère que la condition est toujours fausse ?
- le mécanisme de prédiction de branchement considère que la condition est toujours vraie ?

4) Le compilateur optimise le code et produit (pour les mêmes instructions C) :

```
i=0
saut inconditionnel à test
ici:  t[i]=0
      i++
test:  comparer i et a
      Si i < a, aller en ici
suite: ...
```

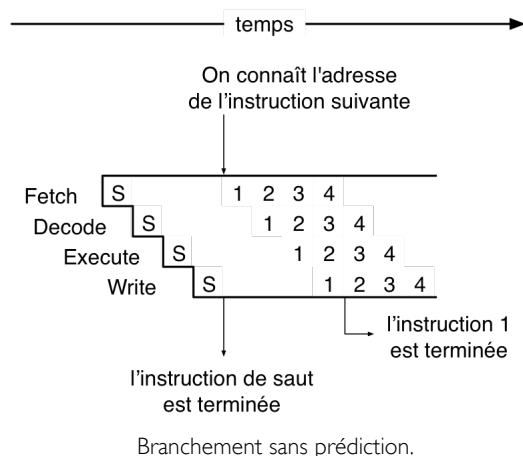
Combien de fois l'instruction de saut conditionnel est-elle exécutée ? Combien de fois le branchement est-il pris et combien de fois n'est-il pas pris ? À l'instant 0, le pipeline est vide et on commence à charger l'instruction i=0. En combien de cycles le programme a-t-il fini de s'exécuter si :

- le mécanisme de prédiction de branchement considère que la condition est toujours fausse ?
- le mécanisme de prédiction de branchement considère que la condition est toujours vraie ?

5) Que conclure ? Proposez un autre algorithme de prédiction de branchement.

1) Il faut attendre que le saut soit à l'étape Write pour connaître l'adresse cible (plus précisément pour savoir si c'est bien l'adresse de la prochaine instruction à exécuter et pour modifier PC). Il y a donc un retard de trois cycles, que la condition soit vraie ou fausse (voir figure 3.25).

Figure 3.25



2) Si la prédiction de branchement est correcte, il n'y a aucun retard ; si la prédiction est fausse, on retombe sur le cas normal avec un retard de trois cycles.

3) La variable i prend toutes les valeurs de 0 à $a - 1$ pour exécuter la boucle, puis prend la valeur a pour sortir de la boucle. Le saut conditionnel est donc exécuté $a + 1$ fois : a fois la condition est fausse (et le processeur poursuit les instructions de la boucle) ; 1 fois la condition est vraie (et le saut est réalisé).

Tableau 3.1

Instruction	Nombre de cycles	Commentaire
$i=0$	4	Car c'est la première instruction.
$t[i]=0$ $i++$ Saut inconditionnel	$3a$	Un cycle supplémentaire par instruction dans la boucle qui s'exécute a fois.
Comparaison	$a + 1$	
Saut (prédiction correcte)	$a \times (1 + 0)$	La condition est fausse a fois, donc la prédiction est correcte a fois et il n'y a pas de retard dans le pipeline.
Saut (avec le retard)	$1 \times (1 + 3)$	La prédiction est incorrecte lors du dernier saut, induisant un retard.

Temps d'exécution si la condition est supposée toujours fausse

Soit un total de $5a + 9$ cycles.

Tableau 3.2

Instruction	Nombre de cycles	Commentaire
$i=0$	4	Car c'est la première instruction.
$t[i]=0$ $i++$ Saut inconditionnel	$3a$	Un cycle supplémentaire par instruction dans la boucle qui s'exécute a fois.
Comparaison	$a + 1$	
Saut (avec le retard)	$a \times (1 + 3)$	La condition est fausse a fois, donc la prédiction est incorrecte a fois, induisant un retard.
Saut (prédiction correcte)	$1 \times (1 + 0)$	La prédiction est correcte lors du dernier saut.

Temps d'exécution si la condition est supposée toujours vraie

Soit un total de $8a + 6$ cycles.

Dans cet exemple, il faut espérer que la condition soit supposée toujours fausse pour profiter de la prédiction de branchement car, si a prend des valeurs importantes, le temps d'exécution est beaucoup plus grand dans le second cas.

4) La variable i prend toutes les valeurs de 0 à $a - 1$ pour exécuter la boucle, puis prend la valeur a pour sortir de la boucle. Le saut conditionnel est donc exécuté $a + 1$ fois : 1 fois la condition est fausse (et le saut n'est pas réalisé) ; a fois la condition est vraie (et le saut est réalisé).

Tableau 3.3

Instruction	Nombre de cycles	Commentaire
$i=0$	4	Car c'est la première instruction.
Saut inconditionnel	1	Le saut ne s'exécute qu'une fois.
Comparaison	$a + 1$	
$t[i]=0$ $i++$	$2a$	Un cycle supplémentaire par instruction dans la boucle qui s'exécute a fois.
Saut (prédiction correcte)	$1 \times (1 + 0)$	La condition est fausse une fois, donc la prédiction est correcte une fois.
Saut (avec le retard)	$a \times (1 + 3)$	La prédiction est incorrecte à chaque saut,

		sauf le dernier, induisant un retard.
--	--	---------------------------------------

Temps d'exécution si la condition est supposée toujours fausse

Soit un total de $7a + 7$ cycles.

Tableau 3.4

Instruction	Nombre de cycles	Commentaire
$i=0$	4	Car c'est la première instruction.
Saut inconditionnel	1	Le saut ne s'exécute qu'une fois.
Comparaison	$a + 1$	
$t[i]=0$ $i++$	$2a$	Un cycle supplémentaire par instruction dans la boucle qui s'exécute a fois.
Saut (avec le retard)	$1 \times (1 + 3)$	La condition est fausse une fois, donc la prédiction est incorrecte une fois, induisant un retard.
Saut (prédiction correcte)	$a \times (1 + 0)$	La prédiction est correcte à chaque saut, sauf le dernier.

Temps d'exécution si la condition est supposée toujours vraie

Soit un total de $4a + 10$ cycles.

Dans cet exemple, il faut espérer que la condition soit supposée toujours vraie pour profiter de la prédiction de branchement.

5) Il est difficile d'avoir une prédiction statique sur la condition car un compilateur peut optimiser le code et inverser la valeur (vraie ou fausse) la plus fréquente de la condition. Dans le premier cas, elle est le plus souvent fausse alors que, dans le second, c'est l'inverse. Or, le processeur ne sait pas quel est le code généré par le compilateur. Une première amélioration de l'algorithme consiste à faire une prédiction statique, non sur la condition, mais sur le sens du saut. L'hypothèse classique consiste à considérer qu'un saut vers l'avant n'est pas réalisé alors qu'un saut vers l'arrière l'est, indépendamment de la condition. Dans les deux exemples de code, cela donnerait le meilleur temps d'exécution car c'est effectivement ce qui se passe.

On peut continuer à améliorer l'algorithme en effectuant une prédiction dynamique, par exemple fondée sur un historique des conditions. Ainsi, quel que soit le code, la prédiction ira le plus souvent dans le sens de la condition, sauf dans des cas spéciaux où la condition est vraie et fausse en alternance.

On peut aussi avoir les moyens et la place pour deux pipelines sur la puce. Il suffit alors de commencer à exécuter les deux branches du saut en parallèle et de garder uniquement la branche correcte une fois la condition évaluée.

Exercice 10 : Exécution multithread

Une application découpée en quatre threads doit s'exécuter sur un processeur. Celui-ci est de type superscalaire, avec quatre unités d'exécution pouvant chacune traiter en un cycle n'importe quelle instruction.

- Le thread T_1 contient mille cinq cents instructions et son degré moyen de parallélisme est de 2 (c'est-à-dire qu'en moyenne, à cause des dépendances, le processeur peut exécuter deux instructions du thread simultanément).
- Le thread T_2 contient deux mille cinq cents instructions et son degré moyen de parallélisme est de 2,5.
- Le thread T_3 contient trois mille cinq cents instructions et son degré moyen de parallélisme est de 3,5.
- Le thread T_4 contient trois mille instructions et son degré moyen de parallélisme est de 1,5.

1) Le processeur exécute les quatre threads successivement. Quel est le temps total nécessaire à leur exécution ?

- 2) Le processeur est maintenant doté d'une capacité d'exécution SMT (*Simultaneous MultiThreading*). Quel est, en théorie, le temps minimum nécessaire pour l'exécution des quatre threads ?
- 3) En pratique, le processeur ne peut exécuter simultanément que deux threads (en équilibrant au maximum l'usage des quatre unités d'exécution entre les deux threads). Il commence par T_1 et T_2 , puis, quand l'un des deux se termine, passe à T_3 puis à T_4 . Quel est le temps total nécessaire à leur exécution ?

1) Le premier thread demande sept cent cinquante cycles ($1\,500/2$). Le deuxième thread demande mille cycles ($2\,500/2,5$). Le troisième thread demande mille cycles ($3\,500/3,5$). Le quatrième thread demande deux mille cycles ($3\,000/1,5$). Soit un temps total de quatre mille sept cent cinquante cycles.

2) En théorie, le processeur pourrait exécuter quatre instructions par cycle, par exemple une de chaque thread (donc plus de problème de dépendances). Il y a dix mille cinq cents instructions au total à exécuter. À raison de quatre par cycle, le temps total d'exécution est de deux mille six cent vingt-cinq cycles. Malheureusement, on peut rarement exécuter autant de threads à chaque cycle car cela demande des ressources supplémentaires (il faut, par exemple, gérer autant de bancs de registres que de threads).

3) On peut découper l'exécution en quatre tranches de temps :

1. Les deux premiers threads s'exécutent pendant sept cent cinquante cycles à raison de deux instructions par thread. Le premier se termine.
2. Il reste mille instructions à T_2 , qui peut donc tourner avec T_3 (en exécutant deux instructions chacun par cycle) pendant cinq cents cycles. T_4 est ensuite lancé.
3. T_4 ne pouvant exécuter qu'une instruction et demi par cycle, T_3 peut occuper deux unités et demi d'exécution et nécessite mille cycles pour se terminer.
4. T_4 ayant exécuté mille cinq cents instructions, les mille cinq cents restantes s'exécutent en de nouveau mille cycles.

Le temps total est donc de trois mille deux cents cinquante cycles. Soit un gain de presque 32 %.

Chapitre 4

Exemple de langage assembleur

On utilise de plus en plus rarement le langage assembleur pour programmer. Néanmoins, il est important de comprendre le fonctionnement du processeur et, pour cela, de comprendre son langage de commande et la façon de l'utiliser.

Après avoir décrit un processeur, nous verrons que les instructions classiques, transferts de données, opérations arithmétiques, ruptures de séquence, sont les principaux outils à notre disposition pour écrire des programmes qui vont être transformés en code machine par un programme d'assemblage, à qui nous allons donner des directives pour faciliter le travail de traduction.

Exemple de langage assembleur

- 1. Description d'un processeur 90
- 2. Instructions 91
- 3. Programme d'assemblage 99
- 4. Extensions possibles 102

Problèmes et exercices

- 1. Trouver la longueur d'une chaîne..... 104
- 2. Parcourir un tableau 104
- 3. Remplacer des caractères dans une chaîne..... 105
- 4. Simuler la multiplication..... 105
- 5. Renverser une chaîne 106
- 6. Transformer une chaîne..... 107
- 7. Additionner deux chaînes de nombres 107
- 8. Compter les caractères 108
- 9. Compter les mots 110
- 10. Compter les sous-chaînes 111
- 11. Permuter des caractères 112
- 12. Additionner sur 64 bits..... 112
- 13. Trier une liste 113
- 14. Trier une liste par appel de fonction..... 114

I. DESCRIPTION D'UN PROCESSEUR

Lors de la conception d'un processeur, le constructeur doit faire des choix concernant son jeu d'instructions et son architecture interne. Ces choix sont conditionnés par un certain nombre de contraintes et de souhaits, par exemple :

- le nombre de transistors disponibles sur la puce ;
- une compatibilité avec les modèles de processeurs précédents ;
- la recherche de performances maximales ;
- un lien avec le système d'exploitation via des instructions spécifiques (gestion de la mémoire, protection du processeur) ;
- une prise en charge des langages évolués (appels de fonction, instructions temps réel) ;
- l'inclusion de nouvelles fonctionnalités (traitement vidéo par exemple) ;
- l'interaction avec d'autres dispositifs matériels (mémoire cache, entrées/sorties).

Concevoir un processeur de A à Z revient souvent à rechercher un équilibre entre les différents composants internes et le jeu d'instructions.

De nos jours, tous les développements se font en langage évolué (Java, C, C++...) et il est assez rare de devoir programmer en langage assembleur. Citons par exemple le développement de pilotes logiciels qui communiquent directement avec le matériel, la programmation de systèmes embarqués (carte à puce, sondes spatiales, électroménager), le besoin de performances maximales (jeux vidéo, traitement d'images en temps réel), qui peuvent être plus efficaces si certaines parties du code sont directement écrites en langage assembleur.

Les processeurs actuels possèdent un jeu d'instructions souvent compliqué, issu d'un mélange de recherche de performances et de besoin de compatibilité, limité par des contraintes techniques. Il est important de connaître la problématique correspondante pour comprendre le fonctionnement du processeur, directement lié à son jeu d'instructions. Par souci de simplification, nous allons ici nous affranchir des contraintes matérielles (coût, performances maximales) pour présenter un processeur imaginaire afin de mieux comprendre l'architecture de son jeu d'instructions et les subtilités de la programmation en langage assembleur.

I.1 Registres et UAL

Notre processeur possède une UAL capable d'effectuer des opérations entières sur 32 bits : addition, soustraction, multiplication, division, ainsi que les opérations logiques et de décalage (voir figure 4.1).

Il possède également trente-deux registres généraux de 32 bits, numérotés de *r0* à *r31*, ainsi qu'un registre PC sur 32 bits pour stocker l'adresse de la prochaine instruction à exécuter. Le registre d'état contient les 4 bits Z, C, N et V, décrits au chapitre 3 :

- **Le bit Z (Zero)**. Il est mis à 1 si le résultat de l'instruction est nul, sinon il est mis à 0.
- **Le bit C (Carry)**. Il est mis à 1 si l'instruction génère une retenue finale, sinon il est mis à 0.
- **Le bit N (Negative)**. Il est mis à 1 si le résultat de l'instruction est négatif, sinon il est mis à 0 ; les nombres étant signés, c'est la recopie du bit de poids fort du résultat.
- **Le bit V (overflow)**. Il est mis à 1 si l'instruction génère un débordement arithmétique, sinon il est mis à 0.

I.2 Jeu d'instructions

On retrouve dans notre processeur le jeu d'instructions classique vu au chapitre 3. Il est fondé sur ce que l'on appelle une architecture *Load/Store* (modèle RISC). Cela signifie que les seules instructions faisant référence à une case mémoire sont les transferts de données depuis ou vers un registre. Les opérations arithmétiques ne peuvent s'effectuer que sur des données se trouvant en registre.

Toutes les instructions ont un code numérique sur 32 bits, décrit à la section suivante.

Figure 4.1

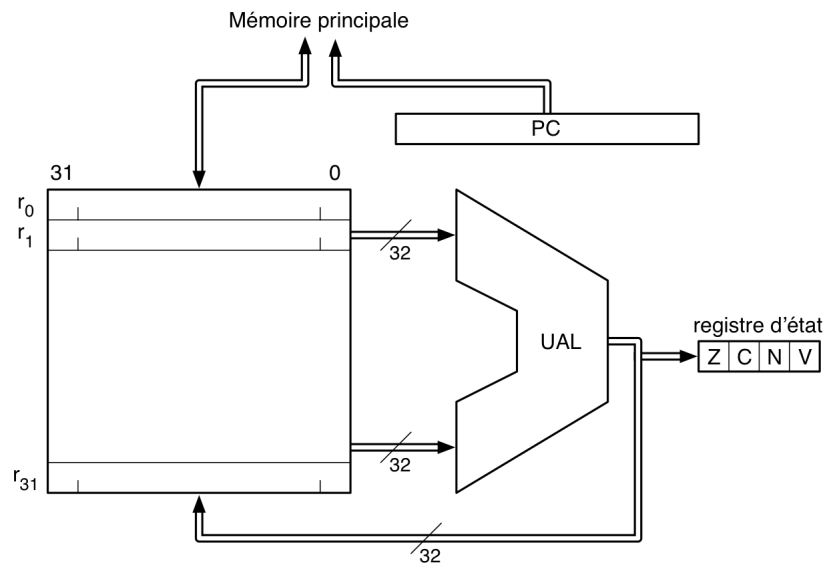


Schéma du cœur du processeur.

1.3 Mémoire

On associe une mémoire au processeur afin de stocker les instructions et les données. Les références à la mémoire se font *via* une adresse contenue dans un registre (adressage indirect). Les registres sont de 32 bits et permettent donc en théorie d'adresser 2^{32} cases mémoire. Cependant, on a la possibilité d'exprimer dans les instructions une valeur immédiate sur 21 bits au maximum (voir section suivante). Il est donc plus simple de limiter les adresses à cette taille et d'avoir un espace mémoire sur 2^{21} octets.

2. INSTRUCTIONS

Tous les processeurs possèdent les mêmes types d'instructions (voir chapitre 3) et le nôtre ne déroge pas à cette règle. Afin de ne pas compliquer le propos, nous nous contenterons de présenter ici les instructions standard du processeur, en traitant les extensions possibles à la dernière section.

2.1 Transfert de données

Les données peuvent être stockées en mémoire ou dans les trente-deux registres entiers. Il faut donc prévoir les possibilités de transfert entre ces différents emplacements. Pour limiter les accès mémoire liés à une instruction (pénalisant en termes de performances, voir chapitre 5), le processeur interdit les transferts directs mémoire-mémoire et se contente des transferts mémoire-registre ou entre registres.

Tableau 4.1

Mnémonique	Opération	Commentaire
MOV rX, rY	$rX \leftarrow rY$	Le registre rX reçoit la valeur sur 32 bits contenue dans le registre rY .
MVI $rX, \#i$	$rX \leftarrow \text{valeur } i$	Le registre rX reçoit sur 32 bits la valeur immédiate i indiquée dans l'instruction (MVI, <i>MoVe Immediate</i>).
LDB $rX, (rY)$	$rX \leftarrow \text{Mem}[rY]_8$	Le registre rX reçoit l'octet mémoire dont l'adresse est dans rY (<i>LoaD Byte</i>).
LDH $rX, (rY)$	$rX \leftarrow \text{Mem}[rY]_{16}$	Le registre rX reçoit les 2 octets mémoire dont l'adresse est dans rY (<i>LoaD Half-word</i>).
LDW $rX, (rY)$	$rX \leftarrow \text{Mem}[rY]_{32}$	Le registre rX reçoit les 4 octets mémoire dont l'adresse

		est dans rY (<i>Load Word</i>).
STB (rX), rY	$\text{Mem}[rX]_8 \leftarrow rY$	L'octet de poids faible de rY est stocké en mémoire à l'adresse contenue dans rX (<i>Store Byte</i>).
STH (rX), rY	$\text{Mem}[rX]_{16} \leftarrow rY$	Les 2 octets de poids faible de rY sont stockés en mémoire à l'adresse contenue dans rX (<i>Store Half-word</i>).
STW (rX), rY	$\text{Mem}[rX]_{32} \leftarrow rY$	Les 4 octets de rY sont stockés en mémoire à l'adresse contenue dans rX (<i>Store Word</i>).

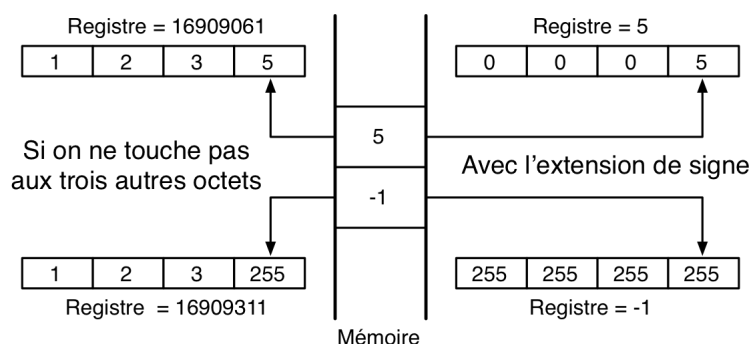
Instructions de transfert de données

La première instruction recopie dans le registre destination (rX) la valeur contenue dans le registre source (rY). Bien sûr, celui-ci n'est en aucun cas affecté par cette opération, qui porte sur tous les bits du registre destination.

La deuxième instruction permet de mettre dans le registre destination (rX) une valeur constante dont le codage est immédiatement donné dans l'instruction. On parle alors du codage immédiat de la valeur ou encore de valeur immédiate. Cette valeur est, par défaut, écrite en décimal. Si elle est négative, elle est codée dans la représentation en complément à 2 dans les 32 bits du registre (voir chapitre 1). Il est parfois plus simple d'exprimer la constante en hexadécimal ; dans ce cas, il faut lui adjoindre le préfixe $0x$: $0xffff$ est ainsi équivalent à $\#-1$. On peut aussi mettre un caractère entre apostrophes, qui sera remplacé par sa valeur en code ASCII : $\#'A'$ s'utilise comme $\#65$. Ces règles seront d'ailleurs valables pour toutes les instructions dans lesquelles est exprimée une valeur immédiate.

Les instructions *Load* permettent de récupérer 1, 2 ou 4 octets en mémoire. Pour ce faire, il faut avoir précédemment mis l'adresse en question dans un autre registre, qui servira alors de pointeur. Lorsque l'on charge plus de 1 octet, l'adresse mémoire indiquée dans le registre correspond à la case du premier octet récupéré, le deuxième ou les trois autres octets pris étant dans les cases mémoire suivantes. Si l'on chargeait 1 ou 2 octets seulement, on pourrait ne pas toucher aux autres bits du registre, mais la valeur numérique de ce dernier ne correspondrait plus à celle récupérée en mémoire, le registre étant toujours lu sur 32 bits. Les nombres étant signés par défaut, il faut donc effectuer une extension de signe (voir chapitre 1) pour être sûr que la valeur lue en mémoire est bien répercutée sur les 4 octets du registre (voir figure 4.2).

Figure 4.2



Chargement en registre de 1 octet en mémoire.

Les instructions de stockage effectuent l'opération similaire en prenant 1, 2 ou les 4 octets du registre (en commençant par les poids faibles) pour les écrire en mémoire à partir de l'adresse indiquée dans le registre de pointage. Il n'y a pas d'extension de signe car la mémoire travaille octet par octet, et non sur 32 bits comme les registres.

Toutes les instructions de transfert de données modifient les bits du registre d'état : les bits Z et N sont positionnés conformément au résultat, tandis que les bits C et V sont toujours mis à 0.

L'espace mémoire disponible est limité à 2^{21} octets. Que se passe-t-il si l'adresse contenue dans le registre de pointage excède cette valeur ? Le processeur peut réagir de différentes façons : soit arrêter le programme en générant une erreur d'exécution, soit ramener l'adresse à l'intervalle $[0, 2^{21} - 1]$ en prenant son modulo 2^{21} . Nos exemples de programmes étant purement théoriques et le propos n'étant pas de construire un simulateur du processeur, nous ne nous appesantirons pas sur cette question.

2.2 Opérations arithmétiques et logiques

La finalité de tout processeur est de réaliser des calculs. Pour cela, il est doté de plusieurs instructions arithmétiques et logiques, qui opèrent sur les registres. Il est possible de réaliser des additions et soustractions sur 32 bits, des multiplications sur 16 bits (sur 16 bits uniquement, afin de pouvoir coder le résultat sur 32 bits), des divisions de deux nombres et des opérations de complément logique ou arithmétique. En outre, notre processeur disposera des opérateurs ET, OU et OU-exclusif, avec deux registres sources. Pour toutes ces opérations, on pourra remplacer le second registre source (ou la source pour les opérations de complément) par une valeur immédiate codée sur 21 bits.

Tableau 4.2

Mnémonique	Opération	Commentaire
ADD rX, rY, rZ ADD $rX, rY, \#i$	$rX \leftarrow rY + rZ$ $rX \leftarrow rY + \text{valeur } i$	Le registre rX reçoit la somme sur 32 bits des valeurs contenues dans les registres rY et rZ (ou la somme de rY et d'une valeur immédiate i).
SUB rX, rY, rZ SUB $rX, rY, \#i$	$rX \leftarrow rY - rZ$ $rX \leftarrow rY - \text{valeur } i$	Le registre rX reçoit la différence sur 32 bits des valeurs contenues dans les registres rY et rZ (ou la différence de rY et d'une valeur i).
MUL rX, rY, rZ MUL $rX, rY, \#i$	$rX \leftarrow rY \times rZ$ $rX \leftarrow rY \times \text{valeur } i$	Le registre rX reçoit sur 32 bits le produit des 16 bits de poids faible des registres rY et rZ (ou le produit des 16 bits de poids faible de rY et d'une valeur i).
DIV rX, rY, rZ DIV $rX, rY, \#i$	$rX \leftarrow rY / rZ$ $rX \leftarrow rY / \text{valeur } i$	Le registre rX reçoit sur 32 bits le quotient de la division entière des registres rY et rZ (ou le quotient de la division de rY et d'une valeur i).
AND rX, rY, rZ AND $rX, rY, \#i$	$rX \leftarrow rY \text{ ET } rZ$ $rX \leftarrow rY \text{ ET valeur } i$	Le registre rX reçoit le ET logique sur 32 bits des valeurs contenues dans les registres rY et rZ (ou le ET logique de rY et d'une valeur i).
OR rX, rY, rZ OR $rX, rY, \#i$	$rX \leftarrow rY \text{ OU } rZ$ $rX \leftarrow rY \text{ OU valeur } i$	Le registre rX reçoit le OU logique sur 32 bits des valeurs contenues dans les registres rY et rZ (ou le OU logique de rY et d'une valeur i).
XOR rX, rY, rZ XOR $rX, rY, \#i$	$rX \leftarrow rY \oplus rZ$ $rX \leftarrow rY \oplus \text{valeur } i$	Le registre rX reçoit le OU-exclusif sur 32 bits des valeurs contenues dans les registres rY et rZ (ou le OU-exclusif de rY et d'une valeur i).
NOT rX, rZ NOT $rX, \#i$	$rX \leftarrow \text{NON}(rZ)$ $rX \leftarrow \text{NON}(\text{valeur } i)$	Le registre rX reçoit le complémentaire sur 32 bits de la valeur contenue dans le registre rZ (ou le complémentaire d'une valeur i).
NEG rX, rZ NEG $rX, \#i$	$rX \leftarrow -rZ$ $rX \leftarrow -\text{valeur } i$	Le registre rX reçoit l'opposé arithmétique sur 32 bits de la valeur contenue dans le registre rZ (ou de la valeur i), en complément à 2.

Instructions arithmétiques et logiques

Toutes ces instructions modifient les 32 bits du registre destination et utilisent les 32 bits des sources (sauf la multiplication, qui concerne les 16 bits de poids faible).

On a besoin de deux instructions de complément, NOT et NEG, qui se distinguent par le fait que, en complément à 2, on calcule l'opposé arithmétique d'un nombre en effectuant le complément logique (NOT) suivi d'une incrémentation.

Toutes ces instructions positionnent les bits Z et N du registre d'état en fonction du résultat. L'addition, la soustraction et la multiplication affectent les bits C et V, conformément à leur définition (retenue et débordement, voir chapitre 1), alors que les opérations logiques ainsi que la division les mettent à 0 (il n'y a jamais de retenue ou de débordement dans ces cas). Pour ce qui est de la négation, le seul cas de débordement possible concerne la tentative de calcul de l'opposé de -2^{31} , qui n'est pas représentable en complément à 2 ; la retenue est toujours fixée à 0, sauf si l'on prend l'opposé de 0 : il n'y a pas débordement mais l'opération

consiste à ajouter 1 au complémentaire logique de 0 qui est 11...11. Cela provoque la remise à 0 de tous les bits (et l'on retombe bien sur la valeur 0), mais génère une retenue finale.

Exemple d'utilisation des instructions arithmétiques

L'opération de division ne fournit que le quotient entier. En supposant que les deux nombres sont compris entre -2^{15} et $2^{15} - 1$, on peut également récupérer le reste de la division via un petit calcul. Puisque la division entière s'écrit $A = qB + r$, le reste se trouve avec l'équation $r = A - qB$. Il faut donc effectuer la division puis soustraire du dividende le produit du quotient et du diviseur.

Si l'on suppose que les registres $r0$ et $r1$ contiennent respectivement le dividende et le diviseur, la suite d'instructions du listing 4.1 permet de récupérer le quotient dans le registre $r2$ et le reste dans $r3$.

Listing 4.1

```
DIV r2,r0,r1    ; r2 ← quotient de r0/r1
MUL r4,r2,r1    ; r4 ← quotient × r1
SUB r3,r0,r4    ; r3 ← r0 - quotient × r1
```

Calcul du reste d'une division entière

Le choix du registre qui sert à stocker temporairement le produit du quotient et du diviseur, ici $r4$, est totalement arbitraire.

Il est important de limiter les deux nombres à une écriture sur 16 bits car la multiplication ne s'effectue que sur 16 bits. Si le diviseur était de plus grande taille, on risquerait d'avoir un quotient également de taille plus importante (plus de 16 bits significatifs), empêchant un calcul correct lors de la multiplication (il aurait fallu effectuer deux multiplications, l'une sur les 16 bits de poids faible du quotient et l'autre sur les 16 bits de poids fort).

2.3 Décalages et rotations

Les instructions de décalage et de rotation sont importantes car elles permettent d'effectuer rapidement des multiplications ou divisions par 2. Par ailleurs, elles sont utiles pour récupérer un par un les bits d'un registre dans le bit C du registre d'état.

Tableau 4.3

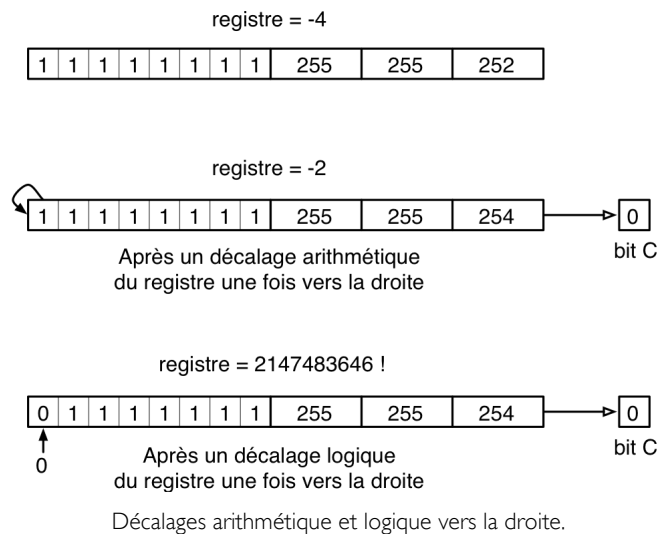
Mnémonique	Opération	Commentaire
LRT rX, rY, rZ LRT $rX, rY, \#i$	$rX \leftarrow rY$ décalé (par rotation) vers la gauche rZ fois (ou i fois)	Le registre rX reçoit la valeur de rY décalée (par rotation) rZ fois (ou i fois) vers la gauche (LRT, <i>Left RotaTe</i>).
LLS rX, rY, rZ LLS $rX, rY, \#i$	$rX \leftarrow rY$ décalé vers la gauche rZ fois (ou i fois)	Le registre rX reçoit la valeur de rY décalée rZ fois (ou i fois) vers la gauche ; les nouveaux bits sont remplacés par des zéros (LLS, <i>Logical Left Shift</i>).
ALS rX, rY, rZ ALS $rX, rY, \#i$	$rX \leftarrow rY$ décalé vers la gauche rZ fois (ou i fois)	Le registre rX reçoit la valeur de rY décalée rZ fois (ou i fois) vers la gauche ; les nouveaux bits sont remplacés par des zéros lors d'un décalage vers la gauche ou par le bit de poids fort lors d'un décalage à droite (ALS, <i>Arithmetical Left Shift</i>).

Instructions de décalage et de rotation

Les instructions de décalage et de rotation fonctionnent toutes de la même façon : le registre rY est décalé ou tourné d'autant de bits que la valeur du registre rZ (ou la valeur immédiate i) et le résultat est mis dans le registre rX . Si rZ (ou la valeur i) est positif, le décalage ou la rotation s'effectue vers la gauche (c'est-à-dire des bits de poids faible vers les bits de poids fort) ; si rZ (ou la valeur i) est négatif, le décalage ou la rotation s'effectue vers la droite.

Dans le cadre d'un décalage vers la gauche (rZ ou i positif), les bits de poids faible sont remplacés par des zéros : cela correspond à une multiplication par 2, les deux instructions LLS et ALS étant alors équivalentes. Les décalages logiques vers la droite (LLS avec rZ ou i négatif) remplacent de la même façon les bits de poids fort par des zéros ; mais il n'y a alors plus de correspondance avec une division par 2 à cause des nombres signés. Pour garder le signe du nombre, il faut conserver le bit de poids fort lors d'un décalage arithmétique vers la droite (instruction ALS) comme à la figure 4.3.

Figure 4.3



Les bits Z et N se positionnent comme d'habitude tandis que le bit V est toujours mis à 0. Le bit C est égal au dernier bit décalé hors du registre rY : lors d'un décalage, c'est le dernier bit qui disparaît, lors d'une rotation, c'est le dernier bit à passer vers le bit opposé (si le nombre de décalages est nul, C est mis à 0).

2.4 Ruptures de séquence

Les sauts conditionnels permettent d'avoir une exécution différenciée (saut effectué ou non) en fonction d'une condition. Dans les instructions que l'on définit, la condition se résume au test de la valeur d'un registre, et suivant les sauts, elle sera vraie (et donc le saut réalisé) si le registre choisi est nul, non nul, positif, négatif, etc. Cette façon d'exprimer la condition facilite les tests sur le résultat d'un calcul, mais ne convient pas si l'on veut tester un bit du registre d'état. On a donc ajouté des instructions de transfert d'un bit de ce registre vers un registre général, permettant ainsi, en deux instructions (transfert puis saut), d'effectuer un branchement en fonction de la valeur de ce bit.

Note

Les deux autres possibilités pour définir la condition de saut (non choisies ici) sont soit de tester un bit du registre d'état (voir un exemple au chapitre 3), soit d'opérer la comparaison complète de deux registres. La première possibilité est plus limitée et oblige à effectuer précédemment un test pour positionner le bit condition en fonction de la valeur souhaitée d'un registre. La seconde est plus puissante puisqu'elle permet de comparer des registres entre eux au lieu d'en comparer un à 0, mais est plus compliquée à implémenter sur un processeur. Par souci de simplicité, nous avons choisi la solution intermédiaire.

Tableau 4.4

Mnémonique	Opération	Commentaire
JMP Adr	$PC \leftarrow Adr$	Saut inconditionnel à l'adresse Adr (JMP, JuMP).
JZ rX,Adr	$PC \leftarrow Adr$ si $rX = 0$	Saut conditionnel à l'adresse Adr si le registre rX est nul (JZ, Jump if Zero).
JNZ rX,Adr	$PC \leftarrow Adr$ si $rX \neq 0$	Saut conditionnel à l'adresse Adr si le registre rX est non nul (JNZ, Jump if Not Zero).
JGT rX,Adr	$PC \leftarrow Adr$ si $rX > 0$	Saut conditionnel à l'adresse Adr si le registre rX est strictement positif (JGT, Jump if Greater Than zero).
JLT rX,Adr	$PC \leftarrow Adr$ si $rX < 0$	Saut conditionnel à l'adresse Adr si le registre rX est strictement négatif (JLT, Jump if Less Than zero).
JGE rX,Adr	$PC \leftarrow Adr$ si $rX \geq 0$	Saut conditionnel à l'adresse Adr si le registre rX est positif ou nul (JGE, Jump if Greater or Equal to zero).
JLE rX,Adr	$PC \leftarrow Adr$ si $rX \leq 0$	Saut conditionnel à l'adresse Adr si le registre rX est

		négatif ou nul (JLE, <i>Jump if Less or Equal to zero</i>).
CCR rX	$rX \leftarrow \text{bit C}$	Le bit C du registre d'état est mis dans le bit de poids faible de rX et ses autres bits sont mis à 0 (CCR, <i>Copy bit C into Register</i>).
CZR rX	$rX \leftarrow \text{bit Z}$	Le bit Z du registre d'état est mis dans le bit de poids faible de rX et ses autres bits sont mis à 0 (CZR, <i>Copy bit Z into Register</i>).
CNR rX	$rX \leftarrow \text{bit N}$	Le bit N du registre d'état est mis dans le bit de poids faible de rX et ses autres bits sont mis à 0 (CNR, <i>Copy bit N into Register</i>).
CVR rX	$rX \leftarrow \text{bit V}$	Le bit V du registre d'état est mis dans le bit de poids faible de rX et ses autres bits sont mis à 0 (CVR, <i>Copy bit V into Register</i>).

Instructions de rupture de séquence et de transfert d'un bit condition

Les branchements sont conformes au schéma classique : un registre est comparé à 0 et, suivant le résultat, soit le processeur effectue le saut en poursuivant l'exécution à l'adresse indiquée dans l'instruction, soit il exécute l'instruction qui suit le saut si la condition est fausse. L'adresse peut être une valeur numérique immédiate, mais elle sera plus probablement donnée par une étiquette liée à une instruction ailleurs dans le programme et calculée par le programme d'assemblage au moment de la génération du code exécutable.

Les instructions de recopie d'un bit condition dans un registre permettent de tester ce bit à l'aide d'un saut. Voici par exemple le test d'un débordement après un calcul :

```
... instructions de calcul
CVR r0
JNZ r0, etiq_debordement
... on continue s'il n'y a pas débordement
```

Ces instructions de recopie mettent à 0 tous les autres bits du registre. Ce dernier ne peut donc valoir que 0 ou 1 suivant la valeur du bit recopié. Par ailleurs, aucune instruction de saut ou de transfert ne modifie les bits du registre d'état.

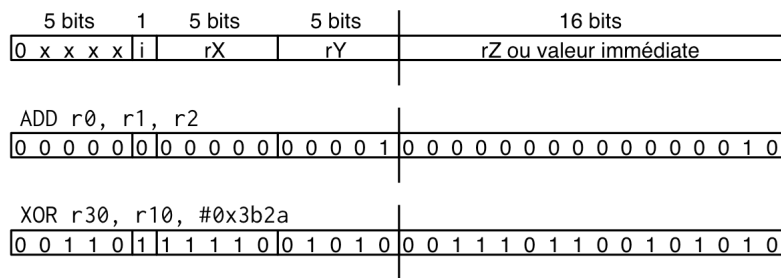
2.5 Format des instructions

Les instructions précédentes vont permettre d'écrire des bouts de programme pour le processeur fictif, mais l'on peut aller plus loin et les coder numériquement en vue de concevoir les circuits internes du processeur. Chacune d'entre elles étant écrite sur 32 bits, on doit séparer les différents champs pour exprimer le code opération et les données.

Quatre familles d'instructions émergent des tableaux 4.1 à 4.4 : les instructions à trois données, à deux registres, à un registre et à une valeur immédiate, et à un seul registre.

Instructions à trois données

Les instructions arithmétiques et logiques ainsi que celles de décalage nécessitent l'expression de trois données dans le code numérique. Comme il y a 32 registres, il faut 5 bits pour en spécifier 1 parmi 32. Le troisième registre pouvant être remplacé par une valeur immédiate, il faut prévoir un bit supplémentaire qui indiquera si la donnée dans l'instruction est un numéro ou une valeur immédiate. Enfin, il serait dommage de limiter cette valeur à 5 bits ; 16 bits sont donc réservés pour le troisième argument de l'instruction (voir figure 4.4).

Figure 4.4

Format des instructions à trois données.

Les 5 premiers bits donnent le code opération de l'instruction suivant le tableau ci-après (qui liste les 4 bits restant à déterminer). Le premier bit est à 0 pour indiquer au processeur une opération de ce type. Le bit *i* est à 0 si les 16 derniers bits donnent un numéro de registre, à 1 si c'est une valeur immédiate. Les deux champs de 5 bits qui suivent correspondent au numéro des registres destination (*rX*) et première source (*rY*).

Comme elles appartiennent à la même famille d'opérations arithmétiques, et bien qu'elles n'aient que deux données, on a inclus les deux instructions NEG et NOT ; dans ce cas, le processeur ignore le champ *rY*.

Tableau 4.5

Instruction	Code opération	Instruction	Code opération
ADD	0000	XOR	0110
SUB	0001	NOT	0111
MUL	0010	NEG	1000
DIV	0011	LRT	1001
AND	0100	LLS	1010
OR	0101	ALS	1011

Code opération des instructions à trois données

La figure 4.4 donne deux exemples de codage d'instruction de ce type. Le premier correspond à ADD *r0*,*r1*,*r2* : on remplace le premier champ par les 4 bits donnant le codage de ADD (0000), puis chacun des trois champs de données contient le numéro en binaire du registre correspondant. Pour finir, on met un 0 dans le bit indiquant si la dernière valeur est immédiate (car ici c'est un registre). Le second exemple est similaire : 0110 est le code de XOR, le bit suivant est à 1 pour indiquer une valeur immédiate comme troisième donnée, et les trois champs suivants précisent les données (registre *r30*, codé 11110 ; registre *r10*, codé 01010 ; valeur hexadécimale 3b2a en binaire).

Instructions à deux registres

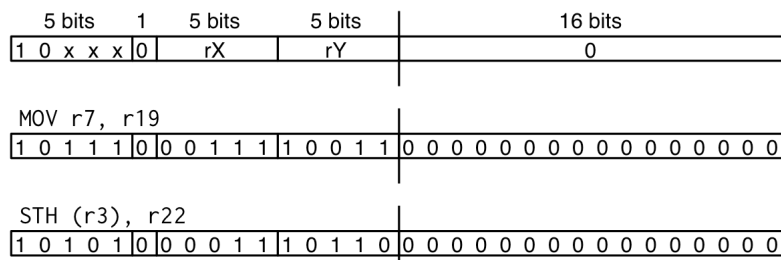
Les instructions de transfert entre registres ou depuis/vers la mémoire utilisent au plus deux registres comme données. On peut donc prendre le même format en laissant à 0 les 16 derniers bits de l'instruction. Les codes opération sont donnés dans le tableau ci-après et le format (ainsi que deux exemples) à la figure 4.5. Pour différencier ces instructions, on fixe les deux premiers bits à 10 et il reste trois bits pour spécifier une instruction parmi sept.

Tableau 4.6

Instruction	Code opération	Instruction	Code opération
MOV	111	STB	100
LDB	000	STH	101
LDH	001	STW	110
LDW	010		

Code opération des instructions à deux registres

Figure 4.5



Format des instructions à deux registres.

Instructions à un registre et à une valeur immédiate

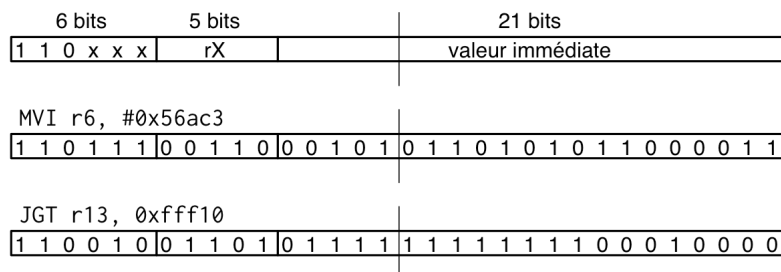
Figurent dans cette catégorie les sauts ainsi que le chargement immédiat d'un registre (MVI). Comme il n'y a qu'un registre à spécifier (sauf avec le saut incondionnel, pour lequel le champ de registre est ignoré), il est ici plus intéressant d'élargir la zone contenant la constante immédiate, en y incluant le champ du second registre (ce qui l'amène à 21 bits), pour augmenter sa valeur maximale (voir tableau ci-après et figure 4.6).

Tableau 4.7

Instruction	Code opération	Instruction	Code opération
MVI	111	JMP	000
JZ	001	JNZ	110
JGT	010	JLE	101
JLT	100	JGE	011

Code opération des instructions à un registre et à une valeur immédiate

Figure 4.6



Format des instructions à un registre et à une valeur.

Instructions à un seul registre

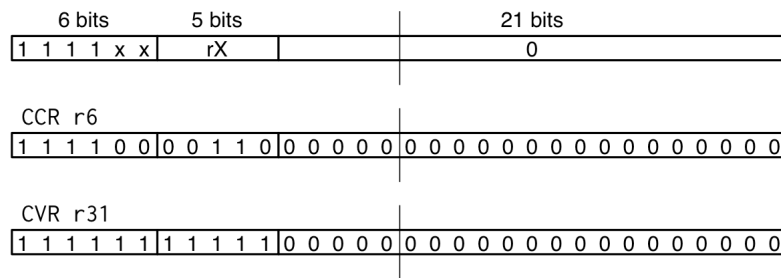
La dernière catégorie concerne les transferts du registre d'état vers un registre général. Il n'y a qu'un registre à préciser, dans le champ du registre destination. On met à 0 les deux champs des registres sources. Par ailleurs, comme il n'y a que quatre instructions, on peut fixer 4 des 6 bits du code opération, 2 bits étant suffisants pour distinguer les instructions (voir tableau ci-après et figure 4.7).

Tableau 4.8

Instruction	Code opération	Instruction	Code opération
CCR	00	CNR	10
CZR	01	CVR	11

Code opération des instructions à un registre

Figure 4.7



Format des instructions à un registre.

3. PROGRAMME D'ASSEMBLAGE

Un programme écrit en langage assembleur, en d'autres termes le code source, est mis dans un fichier texte qui va être soumis à un programme d'assemblage. Celui-ci est chargé de transformer chaque mnémonique en son équivalent numérique et de générer un fichier exécutable, aidé par des directives d'assemblage.

3.1 Génération du code exécutable

Le programme d'assemblage prend comme entrée un fichier code source écrit en langage assembleur et effectue une série de tâches :

- vérification générale de la syntaxe (instructions, données) ;
- mémorisation des correspondances entre les étiquettes et leur valeur ;
- génération du code exécutable.

Une étiquette peut être définie après son utilisation : dans un saut vers l'avant, l'adresse cible du branchement est donnée par une étiquette liée à une instruction (et donc à une adresse mémoire) non encore traduite ; on ne peut donc pas encore remplacer l'étiquette par sa valeur. Pour résoudre ce problème, les programmes d'assemblage fonctionnent souvent en deux passes. La première lecture du code source permet de mémoriser toutes les étiquettes ainsi que leur valeur. Au passage, les vérifications d'usage sont effectuées : chaque étiquette doit se conformer à une syntaxe précise (symboles, longueur) et être définie quelque part dans le programme, une fois et une seule. La seconde passe est l'assemblage proprement dit : les instructions sont réécrites en langage machine, toutes les étiquettes sont remplacées par leur valeur calculée lors de la première passe. Le programme d'assemblage contrôle évidemment la conformité des instructions aux spécifications du constructeur : noms des mnémoniques, noms des opérandes, nombre de données, respect des modes d'adressage autorisés, etc.

Si cette seconde étape se déroule correctement, sont produits un fichier exécutable, traduction des instructions du programme, et un fichier texte source mis en forme avec, en regard de chaque instruction, son adresse mémoire et son code numérique (voir un exemple au listing 4.2).

Listing 4.2

; Adresse	Code num.	étiquette	mnémonique
0x0000	0xc4200014		JZ r1,erreur
0x0004	0x18400001		DIV r2,r0,r1
0x0008	0x10820001		MUL r4,r2,r1
0x000c	0x08600004		SUB r3,r0,r4
0x0010	0xc0000018		JMP suite
0x0014	0xdc600000	erreur:	MVI r3,#0
0x0018	0xdc800002	suite:	MVI r4,#2

Un exemple de fichier source mis en forme

Cross-assembleur

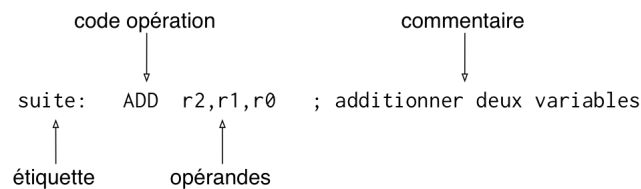
On suppose que le programme d'assemblage utilisé génère un fichier exécutable pour la machine sur laquelle il s'exécute. Mais on peut prévoir de lui faire produire des codes numériques correspondant à un autre processeur (appelé « processeur cible »). On appelle cela un cross-assembleur.

Le processeur cible peut être, non pas dans un ordinateur, mais dans un système embarqué incapable d'exécuter le programme d'assemblage directement. Il faut alors travailler sur un ordinateur séparé, assembler le code source pour le processeur cible, puis transférer l'exécutable sur ce dernier avant de le tester.

3.2 Fichier code source

La structure du fichier code source respecte un format précis pour que le programme d'assemblage puisse faire son travail. Chaque ligne est divisée en quatre champs : étiquette, code opération ou directive, opérandes et commentaire (voir figure 4.8).

Figure 4.8



Format d'une ligne du code source.

Les règles de syntaxe des instructions et des données sont édictées par le constructeur du processeur et publiées dans sa documentation. Toutes les autres caractéristiques du fichier (structure des étiquettes, symboles spéciaux...) sont définies par le programme d'assemblage. On peut donc, en théorie, avoir des syntaxes différentes sur des programmes d'assemblage différents, même s'ils sont destinés à un même processeur. En pratique, le fabricant du processeur donne des indications et produit souvent lui-même un programme d'assemblage qui définit le cadre général. En revanche, rien n'oblige des processeurs différents à avoir une syntaxe d'assembleur identique. Les exemples qui suivent ne prétendent donc pas à l'exhaustivité.

Étiquette

Une étiquette est simplement un nom que l'on associe à une case mémoire, à une instruction ou à une valeur. Ce nom, choisi par le programmeur, peut être composé de lettres (non accentuées pour des questions de compatibilité de jeux de caractères), de chiffres (sauf le premier symbole de l'étiquette) et du caractère `_`, à l'exclusion de tout autre symbole spécial. Pour des raisons évidentes de lisibilité, la plupart des programmes d'assemblage interdisent d'utiliser comme étiquette un nom semblable à un mnémonique du processeur. Souvent, ils imposent de terminer la déclaration d'une étiquette par `:` pour éviter toute confusion. Contrairement à la taille d'un identificateur dans un langage évolué, la longueur d'une étiquette peut se voir limitée par le programme d'assemblage à une dizaine de caractères.

L'étiquette se place en début de ligne et est associée à l'objet suivant se trouvant sur la même ligne. Il est même possible, pour améliorer la lisibilité du code, de ne rien mettre d'autre que l'étiquette sur la ligne : elle est dans ce cas liée à l'objet situé à la ligne suivante.

La grande majorité des lignes de code est dénuée d'étiquette. Pour une meilleure visualisation, on remplace souvent le champ d'étiquette par des espaces ou une tabulation afin d'aligner verticalement toutes les instructions.

Code opération, directives et opérandes

Les deux champs suivants peuvent contenir une instruction, formée du code opération et des opérandes, mais également une directive d'assemblage (aussi appelée « pseudo-opération »), qui n'est pas traduite dans le code exécutable, mais qui constitue une commande pour le programme d'assemblage.

Nous donnons ci-après quelques directives possibles pour notre assembleur fictif. Leurs noms ne sont pas standardisés dans tous les programmes d'assemblage, même si, bien sûr, les concepts sont communs à la plupart d'entre eux.

Définition de valeur

La première directive sert à associer une valeur numérique à une étiquette. On ne stocke rien en mémoire, mais cela permet simplement de remplacer une constante utilisée dans le code par une étiquette pour améliorer la compréhension et simplifier les modifications. On la note `.DEF` (il existe des directives équivalentes `IS` ou `EQU` dans d'autres programmes d'assemblage).

```
valeur: .DEF 10
...
MVI r6,#valeur ; équivalent à MVI r6,#10
```

Le programme d'assemblage est aussi capable d'effectuer des calculs simples sur ces valeurs constantes :

```
valeur: .DEF 10
numero: .DEF 4xvaleur+20 ; numero va valoir 60
```

Début du programme

Le fichier exécutable doit se placer en mémoire pour que le processeur puisse traiter les instructions. Il est possible d'indiquer dans le fichier source l'adresse de début du programme. Plus précisément, la directive `.ORG` indique l'adresse mémoire où il faudra placer l'instruction suivante. Cela permet au programme d'assemblage de calculer l'adresse de chaque instruction et donc de faire la correspondance entre une étiquette et son adresse mémoire pour expliciter les adresses cibles des branchements.

```
.ORG 0x1000
MOV r6,r5 ; 0x1000
JNZ r6,suite ; 0x1004
MVI r0,#1 ; 0x1008
suite: ADD r2,r1,r0 ; 0x100c
```

Dans ce listing, l'adresse de chaque instruction est indiquée en commentaire (chaque instruction est sur 4 octets). Le programme d'assemblage sait donc après la première passe que l'étiquette `suite` est associée à l'adresse `0x100c` ; il peut donc générer le code numérique associé au branchement.

Réservation d'espace mémoire

Le programmeur essaie de faire tenir le maximum de données dans les registres, mais il peut être amené à utiliser la mémoire comme zone de stockage. Il lui suffit d'indiquer une adresse dans un registre, puis d'utiliser le mode d'adressage indirect pour faire référence à une case mémoire. Mais comment la choisir ?

La directive `.RES` permet de réserver un espace mémoire (et d'associer son adresse à une étiquette) de la taille indiquée par le paramètre. Voici un exemple :

```
.ORG 0x1000
var: .RES 20 ; 0x1000 à 0x1013
MVI r0,#var ; 0x1014
MVI r1,#32 ; 0x1018
STB (r0),r1 ; 0x101c
```

Cette suite d'instructions effectue les opérations suivantes :

- définir le début du programme comme étant à l'adresse `0x1000` ;
- réserver 20 octets (donc les adresses hexadécimales `0x1000` à `0x1013`) et associer l'étiquette `var` à l'adresse `0x1000` ;
- mettre cette adresse dans `r0` ;
- mettre la valeur numérique immédiate 32 dans `r1` ;
- stocker cette valeur contenue dans `r1` à l'adresse mémoire contenue dans `r0`, donc à l'adresse `0x1000`.

Initialisation d'espace mémoire

On peut également définir directement des valeurs en mémoire, c'est-à-dire réserver l'espace nécessaire et initialiser une valeur. La directive `.BYTE` réserve autant d'octets qu'il y a de paramètres et initialise chaque octet à la valeur du paramètre correspondant.

`tab: .BYTE 0,1,2,3` réserve 4 octets et leur affecte respectivement les valeurs 0, 1, 2 et 3 ; l'adresse du premier octet est liée à l'étiquette `tab`.

Les directives `.HALF` et `.WORD` font le même travail en réservant respectivement 2 octets (un demi-mot) et 4 octets (un mot) pour chaque paramètre.

Comme il est courant d'utiliser des chaînes de caractères, on peut aussi définir la directive `.STRG`, qui prend en paramètre une chaîne de caractères qu'elle va mettre en mémoire (un caractère par octet). `.STRG "Luc"` est ainsi équivalent à `.BYTE 'L','u','c',0`.

Commentaire

Le dernier champ de chaque ligne, lui aussi facultatif, est le champ commentaire. Un caractère spécial, souvent `;`, délimite le début de cette zone, dans laquelle le programmeur peut mettre ce qu'il veut afin d'expliquer à un lecteur du code source le sens d'une instruction. Il est vivement recommandé de remplir ce champ sans restriction car la lecture d'un programme est encore plus difficile en langage assembleur qu'en langage évolué. Pour ne pas compliquer la tâche du programme d'assemblage, on limite habituellement les commentaires à une ligne. En d'autres termes, ils ne peuvent pas se poursuivre sur plusieurs lignes successives.

4. EXTENSIONS POSSIBLES

Les instructions listées précédemment permettent d'écrire des petits programmes en langage assembleur, mais l'on peut étendre le jeu d'instructions pour le rendre plus puissant : opérations sur des nombres non signés ou flottants, gestion de la pile et appels de fonction, prise en charge des interruptions, etc. Le tableau ci-après propose quelques instructions intéressantes supplémentaires, qui ne sont pas nécessaires en première lecture pour comprendre le langage assembleur.

Tableau 4.9

Mnémonique	Commentaire
LDBU rX, (rY) LDHU rX, (rY)	Le chargement de 1 ou 2 octets depuis la mémoire dans un registre provoque une extension de signe. Il est parfois important de l'éviter et de remplacer les octets de poids fort (non chargés) du registre par 0 ; c'est le cas lorsque l'on travaille avec des caractères ou des entiers non signés (LDHU, <i>LoaD Byte/Half Unsigned</i>).
CALL Adr RET	L'instruction CALL provoque un appel de fonction à l'adresse Adr en empilant le registre PC pour sauvegarder l'adresse de retour. La dernière instruction de la fonction, RET (<i>RETurn</i>), récupère la valeur en haut de la pile et la met dans le registre PC pour reprendre le programme après l'appel.
PUSH rX POP rX	PUSH empile les 32 bits du registre rX. POP récupère les 32 bits au sommet de la pile et les met dans le registre rX.
TRAP RETI	TRAP provoque une interruption logicielle (voir chapitre 8) qui déroute l'exécution à une adresse précise indiquée dans le processeur. RETI (<i>RETurn from Interrupt</i>) termine le traitement de l'interruption et reprend le cours normal du programme.
RST	RST (<i>ReSeT</i>) provoque un redémarrage du processeur, qui arrête tous les traitements en cours.
FMOV fX, fY FMVI fX, #f LDF fX, (rY) STF (rX), fY	Ces instructions organisent les transferts de données flottantes entre registres (FMOV), depuis ou vers la mémoire (LDF/STF), d'une donnée flottante immédiate (FMVI).
FADD fX, fY, fZ FSUB fX, fY, fZ FMUL fX, fY, fZ FDIV fX, fY, fZ	Ces instructions effectuent une opération flottante simple sur des valeurs situées dans les registres.

Instructions supplémentaires possibles

La gestion de la pile doit se faire via un registre pointeur de pile réservé à cet usage, éventuellement l'un des registres généraux, ce qui évite d'avoir en plus des instructions spécifiques pour l'initialiser.

Les opérations en virgule flottante se font via des registres flottants, appelés fX, séparés des registres généraux. Si l'on souhaite implémenter ces instructions, il faut préciser la nature des nombres flottants autorisés (simple ou double précision) ainsi que le nombre de registres flottants, et prévoir des instructions mathématiques plus complexes.

Remarque

Nous avons utilisé jusqu'à présent les termes de « langage assembleur » et de « programme d'assemblage ». Mais en pratique, on emploie de façon générique le terme d'**assembleur**. On programme donc en assembleur, en exécutant un assembleur pour transformer un fichier code source en code exécutable.

RÉSUMÉ

Les différentes instructions assembleur permettent de donner des ordres au processeur et composent tous les programmes qui s'exécutent sur la machine. Elles font appel aux différents registres de la machine et utilisent les modes d'adressage du processeur. Écrit dans un fichier texte, le code source est donné à un programme d'assemblage, dont la tâche est de traduire ces instructions en codes numériques. Pour ce faire, le programme d'assemblage utilise des directives situées dans le code source, qui lui indiquent où placer le programme, quels emplacements mémoire réserver pour les variables, et qui remplacent des valeurs numériques par des étiquettes pour améliorer la lisibilité.

Problèmes et exercices

Le seul moyen d'apprendre à programmer, quel que soit le langage utilisé, en particulier en langage assembleur, est de passer à la pratique en écrivant des programmes. Les exercices de ce chapitre ont pour but de vous faire manipuler les instructions assembleur présentées dans la partie théorique.

Exercice 1 : Trouver la longueur d'une chaîne

Dans *r0* se trouve l'adresse d'une chaîne de caractères terminée par l'octet nul. Écrivez une suite d'instructions assembleur qui, lorsqu'elle se termine, permet de récupérer le nombre de caractères de la chaîne dans *r1*.

Il faut :

- initialiser le compteur (c'est-à-dire *r1*) à 0 ;
- récupérer un caractère et tester s'il est nul ;
- si c'est le cas, c'est fini, sinon il faut incrémenter le compteur et le pointeur sur la chaîne, et recommencer.

Voici la suite d'instructions :

Listing 4.3

```
ici:  MVI    r1,#0      ; compteur égal à 0
      LDB    r2,(r0)   ; récupérer prochain caractère
      JZ     r2,fin    ; fin de la chaîne ?
      ADD    r1,r1,#1  ; sinon incrémenter le compteur
      ADD    r0,r0,#1  ; et passer au caractère suivant
      JMP    ici       ; retour dans la boucle
fin:
```

Longueur d'une chaîne

Exercice 2 : Parcourir un tableau

1) Un tableau d'entiers est stocké en mémoire. Chaque élément a une valeur de 1 à 127 et est donc stocké sur 1 octet. L'octet qui suit le dernier élément du tableau est nul. L'adresse du premier élément du tableau se trouve dans le registre *r0*. Écrivez une suite d'instructions assembleur qui, lorsqu'elle se termine, permet de récupérer le plus grand élément du tableau dans le registre *r1*.

2) Ajoutez au programme précédent le calcul de la somme de tous les éléments du tableau dans le registre *r2*.

1) Il faut parcourir le tableau comme dans l'exercice précédent, mais en ajoutant la comparaison de chaque entier avec la plus grande valeur déjà trouvée.

Listing 4.4

```
ici:  MVI    r1,#0      ; initialiser meilleur à 0
      LDB    r3,(r0)   ; charger prochain caractère
      ADD    r0,r0,#1  ; avancer pointeur sur suivant
      JZ     r3,fin    ; fin de la chaîne ?
      SUB    r31,r3,r1  ; comparer r3 et r1 dans r31
      JLE    r31,ici    ; revenir si caractère ≤ meilleur
      MOV    r1,r3      ; sinon meilleur = caractère
      JMP    ici       ; retour dans la boucle
fin:
```

Plus grand élément d'un tableau

2) Il suffit d'ajouter l'initialisation du registre *r2* à 0 et d'additionner à chaque fois le caractère chargé à la somme.

Listing 4.5

```

MVI    r1,#0      ; meilleur égal à 0
MVI    r2,#0      ; somme égale à 0
ici:   LDB    r3,(r0) ; charger prochain caractère
      ADD    r0,r0,#1 ; avancer pointeur sur suivant
      JZ     r3,fin  ; fin de la chaîne ?
      ADD    r2,r2,r3 ; additionner le caractère à la somme
      SUB    r31,r3,r1 ; comparer r3 et r1 dans r31
      JLE    r31,ici ; revenir si caractère ≤ meilleur
      MOV    r1,r3   ; sinon meilleur = caractère
      JMP    ici     ; retour dans la boucle
fin:
```

Plus grand élément d'un tableau et calcul de la somme

Il est important de limiter les entiers à 127 car, s'ils pouvaient prendre une valeur entre 128 et 255, ils seraient non signés et il faudrait utiliser l'instruction LDBU (voir section 4). En revanche, il pourrait y avoir une erreur car on ne vérifie pas que la somme de tous les éléments ne dépasse pas 2^{31} (ce qui créerait un débordement arithmétique).

La seule difficulté du code est de comparer les deux registres *r1* et *r3*. L'astuce classique est alors d'effectuer une soustraction entre les deux dans un registre inutilisé et de comparer ce dernier à 0.

Exercice 3 : Remplacer des caractères dans une chaîne

Une chaîne de caractères est stockée en mémoire. Chaque élément occupe 1 octet et aucun n'est nul. L'octet qui suit le dernier élément de la chaîne est nul. L'adresse du premier élément de la chaîne se trouve dans le registre *r0*. On souhaite effectuer une opération de remplacement de certains caractères de la chaîne. Le registre *r1* contient le caractère à remplacer. Il s'agit de substituer à chaque occurrence de ce dernier dans la chaîne, le caractère se trouvant dans le registre *r2*.

Écrivez une suite d'instructions assembleur qui, lorsqu'elle se termine, assure que toutes les occurrences (dans la chaîne) du caractère contenu dans *r1* sont remplacées par le caractère contenu dans *r2*. On souhaite également avoir à la fin, dans le registre *r3*, le nombre de remplacements effectués.

Listing 4.6

```

MVI    r3,#0      ; nombre de remplacements égal à 0
ici:   LDB    r4,(r0) ; charger prochain caractère
      JZ     r4,fin  ; fin de la chaîne ?
      SUB    r4,r4,r1 ; comparer r4 et r1
      JNZ    r4,next ; sauter remplacement si différent
      STB    (r0),r2 ; si égal, remplacer par r2
      ADD    r3,r3,#1 ; nombre de remplacements + 1
next:  ADD    r0,r0,#1 ; avancer pointeur sur suivant
      JMP    ici     ; retour dans la boucle
fin:
```

Remplacement d'un caractère d'une chaîne

Là encore, on soustrait *r1* au caractère chargé pour tester l'égalité des deux : si le résultat est nul, les deux caractères sont égaux. L'autre point digne d'attention est que l'on a besoin de remplacer le caractère en mémoire (d'où l'instruction STB) ; il ne faut donc pas incrémenter le pointeur sur la chaîne (le registre *r0*) tout de suite après avoir chargé le caractère dans *r1*, mais à la fin de la boucle.

Exercice 4 : Simuler la multiplication

Dans les 16 bits de poids faible de *r1* se trouve un nombre entier non signé, les 16 autres bits de *r1* étant nuls. Dans les 16 bits de poids faible de *r2* se trouve un nombre entier non signé, les 16 autres bits de *r2* étant nuls. Écrivez de deux manières différentes une suite d'instructions assembleur qui, lorsqu'elle se termine, permet de récupérer dans les 32 bits de *r0* le produit des deux nombres (sans bien sûr utiliser directement une instruction de multiplication).

- 1) La suite d'instructions est une simple boucle effectuant autant d'additions que nécessaire.
- 2) La suite d'instructions implémente la procédure classique de multiplication comme suite d'additions des produits partiels (voir chapitre 1).

1) Il suffit de faire une boucle qui additionne le multiplicande autant de fois que la valeur du multiplicateur. On décrémente donc ce dernier à chaque passage dans la boucle jusqu'à ce qu'il atteigne la valeur 0.

Listing 4.7

```

ici:   MVI    r0,#0      ; résultat égal à 0
      JZ     r1,fin     ; c'est fini ?
      ADD    r0,r0,r2   ; additionner une fois
      SUB    r1,r1,#1   ; décrémente le multiplicateur
      JMP    ici        ; retour dans la boucle
fin:

```

Une multiplication simple

2) Le désavantage majeur de l'algorithme précédent est son inefficacité si le multiplicateur prend une grande valeur. Il vaut mieux implémenter l'algorithme classique de la multiplication, qui teste chaque bit du multiplicateur et additionne le multiplicande (décalé) si le bit est à 1.

Listing 4.8

```

ici:   MVI    r0,#0      ; résultat égal à 0
      MVI    r4,#16     ; 16 bits à tester
      LLS    r1,r1,#-1  ; décaler r1 de 1 bit vers la droite
      CCR    r31         ; récupérer le bit C dans r31
      JZ     r31,decal   ; si égal à 0, passer au bit suivant
      ADD    r0,r0,r2    ; sinon additionner
decal: LLS    r2,r2,#1   ; décaler le multiplicande vers la gauche
      SUB    r4,r4,#1   ; décrémente le compteur de bits
      JNZ    r4,ici     ; encore des bits à tester ?

```

Une multiplication plus efficace

Le registre *r0* contient les additions successives des produits partiels. Le registre *r4* sert de compteur de boucle pour effectuer seize passages afin de tester les 16 bits de *r1*. Pour tester un bit de *r1*, on décale le registre d'un bit vers la droite à chaque passage dans la boucle. Le bit qui disparaît est récupéré dans le bit C du registre d'état puis transféré dans *r31*. On teste alors ce dernier. S'il est nul, il n'y a pas d'addition à faire ; s'il est à 1, on additionne le multiplicande. Ensuite on décale le multiplicande pour le bit suivant et on effectue la boucle seize fois.

Par rapport à l'algorithme précédent, celui-ci est beaucoup plus efficace puisque l'on est sûr de limiter à seize le nombre de passages, alors que le premier algorithme peut produire jusqu'à 65 535 additions.

Exercice 5 : Renverser une chaîne

Une chaîne de caractères est stockée en mémoire. Chaque caractère occupe 1 octet et l'octet qui suit le dernier caractère de la chaîne est nul. L'adresse du premier élément de la chaîne se trouve dans le registre *r0*. On souhaite renverser la chaîne, c'est-à-dire l'écrire en commençant par le dernier caractère et stocker cette nouvelle chaîne en mémoire à une adresse contenue dans le registre *r1*. (Ainsi, si la première chaîne est *truc*, on souhaite écrire la chaîne *curt* en mémoire avec le premier élément à l'adresse qui se trouve dans *r1* et avec l'octet nul final.)

Écrivez une suite d'instructions assembleur qui renverse la chaîne pointée par *r0* et la stocke à l'adresse pointée par *r1*.

Il faut d'abord parcourir toute la première chaîne pour arriver à la fin et commencer à recopier à partir du dernier caractère (donc un caractère avant le zéro final de la première chaîne). Mais il faut également savoir quand s'arrêter. Pour cela, on profite du parcours pour compter au passage le nombre de caractères de la chaîne.

Listing 4.9

```

comp:  MVI    r2,#0      ; compte le nombre de caractères
      LDB    r3,(r0)    ; charge le caractère suivant
      JZ     r3,suite   ; fin de la chaîne ?
      ADD    r2,r2,#1   ; sinon incrémenter le compteur

```

```

        ADD    r0,r0,#1 ; et avancer le pointeur
        JMP    comp    ; reboucler
suite:  SUB    r0,r0,#1 ; reculer pointeur chaîne 1
        JZ     r2,fin   ; on a tout renversé ?
        LDB    r3,(r0)  ; charger caractère
        STB    (r1),r3  ; et le recopier
        ADD    r1,r1,#1 ; avancer pointeur chaîne 2
        SUB    r2,r2,#1 ; décrémenter le compteur
        JMP    suite1   ; reboucler
fin:    STB    (r1),r2  ; et le zéro final à stocker (r2 vaut zéro !)

```

Renversement d'une chaîne

Le registre *r2* sert de compteur pour le nombre de caractères de la chaîne. La seule petite subtilité est qu'il ne faut pas oublier le cas où la première chaîne serait vide. C'est pourquoi on teste tout de suite le compteur, avant de commencer à recopier. De même, il ne faut pas oublier d'ajouter à la « main » le zéro final à la deuxième chaîne.

Exercice 6 : Transformer une chaîne

Une chaîne de caractères est stockée en mémoire. Chaque élément occupe 1 octet et aucun n'est nul. L'octet qui suit le dernier élément de la chaîne est nul. L'adresse du premier élément de la chaîne se trouve dans le registre *r0*. On souhaite générer des acronymes dans cette chaîne, c'est-à-dire ponctuer d'un point (.) chaque caractère d'un mot, par exemple passer de IBM à I.B.M. On placera la chaîne modifiée à l'adresse contenue dans *r1*.

Les mots à transformer sont placés dans la chaîne initiale entre des dièses (#), qui doivent disparaître dans la chaîne transformée. On considère qu'il n'y a pas de blancs ni de caractères bizarroïdes entre les dièses. Par exemple, je suis chez #IBM# pour toujours deviendra je suis chez I.B.M. pour toujours. La chaîne peut commencer par le caractère #: #RATP#, #SNCF# : même combat deviendra R.A.T.P., S.N.C.F. : même combat. Il peut y avoir aucun ou plusieurs dièses. Si c'est le dernier caractère de la chaîne, il peut être omis.

Écrivez une suite d'instructions assembleur qui, lorsqu'elle se termine, laisse à l'adresse pointée par *r1* la chaîne modifiée.

Il y a deux boucles imbriquées à faire pour parcourir, d'un côté, toute la chaîne et, de l'autre, la chaîne pendant la transformation. Il faut évidemment tester la fin de la chaîne ou le caractère # dès le début de la boucle. De même, pendant la transformation, il faut vérifier que l'on n'arrive pas à la fin de la chaîne.

Listing 4.10

```

        MVI    r3,#'.'   ; le caractère à ajouter
debut:  LDB    r2,(r0)    ; récupérer prochain caractère
        JZ     r2,fin    ; fin de la chaîne ?
        SUB    r31,r2,#'#' ; comparer à '#'
        JZ     r31,start ; début de la transformation ?
        STB    (r1),r2   ; sinon recopier le caractère
        ADD    r0,r0,#1  ; avancer pointeur chaîne 1
        ADD    r1,r1,#1  ; avancer pointeur chaîne 2
        JMP    debut     ; passer au caractère suivant
start:  ADD    r0,r0,#1  ; avancer pointeur chaîne 1
        LDB    r2,(r0)  ; récupérer le caractère
        JZ     r2,fin    ; fin de la chaîne ?
        SUB    r31,r2,#'#' ; comparer à '#'
        JZ     r31,end   ; fin de la transformation ?
        STB    (r1),r2   ; sinon recopier caractère
        ADD    r1,r1,#1  ; avancer pointeur chaîne 2
        STB    (r1),r3   ; mettre un '.'
        ADD    r1,r1,#1  ; avancer pointeur chaîne 2
        JMP    start     ; continuer la recopie
end:    ADD    r0,r0,#1  ; avancer pointeur 1 (sauter le '#')
        JMP    debut     ; passer au caractère suivant
fin:

```

Transformation d'une chaîne

Exercice 7 : Additionner deux chaînes de nombres

On désire faire l'addition de deux nombres entiers positifs stockés sous la forme de deux chaînes de caractères. Chaque caractère d'une chaîne représente un chiffre de 0 à 9 stocké sur 1 octet en code ASCII de valeur décimale 48 (pour 0) à 57 (pour 9). Les deux chaînes ont le même nombre de caractères et se terminent chacune par un octet nul. L'adresse du premier caractère de la première chaîne et de la deuxième se trouvent respectivement dans les registres *r0* et *r1*. On désire stocker l'addition des deux nombres sous la forme d'une chaîne de caractères en mémoire terminée par un octet nul. On mettra son premier caractère à l'adresse contenue dans le registre *r2*. Les deux chaînes sont stockées à l'envers de leur affichage : le premier caractère est le chiffre le moins significatif et le dernier le plus significatif. On peut donc effectuer l'addition en parcourant les chaînes du premier au dernier caractère.

Écrivez une suite d'instructions assembleur qui, lorsqu'elle se termine, laisse en mémoire à l'adresse initialement pointée par *r2* la chaîne de caractères ASCII correspondant à la somme des deux nombres.

Pour additionner deux caractères codant des chiffres, il suffit d'ajouter les deux valeurs ASCII et de retrancher le code ASCII de 0 afin de retomber sur un caractère, sauf si la somme dépasse le code ASCII du caractère 9. Dans ce cas, on retranche 10 et on mémorise une retenue pour la somme des deux caractères suivants. À la fin, il ne faut pas oublier une éventuelle dernière retenue, qui se matérialise par le caractère 1 en plus.

Listing 4.11

```
ici:  MVI    r3,#0      ; retenue égale à 0
      LDB    r4,(r0)   ; charger les deux caractères
      LDB    r5,(r1)   ;
      ADD    r0,r0,#1   ; et pointer sur les
      ADD    r1,r1,#1   ; caractères suivants
      JZ     r4,fin     ; fin de la chaîne ?
      ADD    r6,r4,r5   ; additionner les 2 caractères
      SUB    r6,r6,#'0' ; revenir à un caractère ASCII
      ADD    r6,r6,r3   ; ajouter la précédente retenue
      MVI    r3,#0     ; et mettre la retenue à 0
      SUB    r31,r6,#'9' ; le caractère est-il
      JLE    r31,inf    ; plus petit ou égal à 9 ?
      SUB    r6,r6,#10  ; si non, retirer 10
      MVI    r3,#1     ; et mettre la retenue à 1
inf:  LDB    (r2),r6     ; stocker le caractère somme
      ADD    r2,r2,#1   ; et avancer le pointeur
      JMP    ici        ; passer aux caractères suivants
fin:  JZ     r3,fin2     ; y a-t-il une retenue finale ?
      MVI    r7,#'1'   ; si oui mettre un 1
      STB    (r2),r7   ;
      ADD    r2,r2,#1   ; et avancer le pointeur
fin2: MVI    r7,#0     ; mettre le 0 final
      STB    (r2),r7   ; à la fin de la chaîne
```

Addition de deux chaînes

Le registre *r3* sert à mémoriser la retenue éventuelle entre deux caractères, le registre *r7* à mettre l'éventuel caractère 1 final et le caractère nul après.

Exercice 8 : Compter les caractères

1) Une chaîne de caractères est stockée en mémoire. Chaque caractère occupe 1 octet et est soit un *x*, soit un *y*. L'octet qui suit le dernier caractère de la chaîne est nul. L'adresse du premier élément de la chaîne se trouve dans le registre *r0*. On souhaite comparer le nombre de groupes de caractères *x* et le nombre de groupes de caractères *y* de la chaîne. Un groupe de caractères *x* (respectivement *y*) est simplement un ou plusieurs caractères *x* (respectivement *y*) qui se suivent. Plus précisément, on souhaite avoir dans *r1*, à la fin des instructions, 0 s'il y a plus de groupes de *x* que de groupes de *y*, et 1 s'il y a autant ou plus de groupes de *y* que de groupes de *x*.

Écrivez une suite d'instructions assembleur qui, lorsqu'elle se termine, assure que le registre *r1* contient 0 s'il y a strictement plus de groupes de *x* que de groupes de *y*, et 1 sinon.

2) Reprendre la question en supposant la présence éventuelle d'autres caractères dans la chaîne.

1) Le premier algorithme consiste simplement à parcourir la chaîne en bouclant sur chaque groupe et en incrémentant un compteur dans un registre à la fin de chaque groupe. Quand la chaîne est terminée (ne pas oublier de le vérifier à chaque caractère chargé), on compare le nombre de groupes de x au nombre de groupes de y. Cela donne le code suivant :

Listing 4.12

```

MVI    r2,#0      ; nombre de groupes de x
MVI    r3,#0      ; nombre de groupes de y
LDB    r4,(r0)     ; charger le premier caractère
ADD    r0,r0,#1    ; pointeur sur caractère suivant
JZ     r4,fin      ; fin de la chaîne ?
SUB    r31,r4,#'y' ; caractère égal à 'y' ?
JZ     r31,CARy    ; sinon c'est un 'x' ?
CARx:  LDB    r4,(r0) ; charger le caractère suivant
ADD    r0,r0,#1    ; pointeur sur caractère suivant
SUB    r31,r4,#'x' ; est-ce toujours un 'x' ?
JZ     r31,CARx    ; si oui, reboucler sur le groupe
ADD    r2,r2,#1    ; sinon fin d'un groupe de x
JZ     r4,fin      ; est-ce la fin de la chaîne ?
CARy:  LDB    r4,(r0) ; charger caractère suivant
ADD    r0,r0,#1    ; pointeur sur caractère suivant
SUB    r31,r4,#'y' ; est-ce toujours un 'y' ?
JZ     r31,CARy    ; si oui, reboucler sur le groupe
ADD    r3,r3,#1    ; sinon fin d'un groupe de y
SUB    r31,r4,#'x' ; est-ce un 'x' ?
JZ     r31,CARx    ; (sinon fin de chaîne)
fin:   MVI    r1,#1 ; résultat égal à 1
SUB    r31,r3,r2    ; comparer groupes de y et groupes de x
JGE    r31,fin2     ; groupes de y ≥ groupes de x ?
MVI    r1,#0      ; sinon résultat égal à 0
fin2:

```

Compter les caractères (version 1)

Mais une analyse un peu plus fine permet de s'apercevoir qu'il y a au maximum une différence de 1 entre le nombre de groupes de x et de y. En effet, chaque groupe de l'un est forcément suivi d'un groupe de l'autre, sauf éventuellement le dernier. Plus précisément :

- Si le premier caractère est un y, il y a autant de groupes de y que de groupes de x (si le dernier caractère est un x) ou un de plus (si le dernier caractère est un y). Dans ces deux cas, il faut renvoyer 1.
- Si le premier caractère est un x, il y a autant de groupes de x que de groupes de y (si le dernier caractère est un y) ou un de plus (si le dernier caractère est un x). Dans le premier cas on renvoie 1, et 0 dans le deuxième.

D'où le code suivant :

Listing 4.13

```

MVI    r1,#1      ; le cas par défaut
LDB    r2,(r0)     ; charger le premier caractère
JZ     r2,fin      ; fin de la chaîne ?
SUB    r31,r2,#'y' ; le premier caractère égal à 'y' ?
JZ     r31,fin      ; si oui c'est fini
ici:   MOV    r3,r2 ; sauvegarder le dernier caractère
ADD    r0,r0,#1    ; pointeur sur caractère suivant
LDB    r2,(r0)     ; charger caractère suivant
JNZ    r2,ici      ; fin de la chaîne ?
SUB    r31,r3,#'y' ; le dernier caractère était-il un 'y' ?
JZ     r31,fin      ; sinon c'est le dernier cas
MVI    r1,#0      ; 'x' en tête et en fin
fin:

```

Compter les caractères (version 2)

On commence par tester le premier caractère. Si c'est un y, c'est terminé : on renvoie un 1. Sinon, on parcourt toute la chaîne en mémorisant le dernier caractère lu (pour pouvoir le tester après avoir récupéré le zéro final) ; s'il vaut y, on renvoie encore 1, sinon 0.

2) Lorsque l'on introduit d'autres caractères que x et y dans la chaîne, cela force à la parcourir en comptabilisant les deux groupes avec le code suivant :

Listing 4.14

```

MVI    r2,#0      ; nombre de groupes de x
MVI    r3,#0      ; nombre de groupes de y
ici:   LDB    r4,(r0) ; charger le premier caractère
      ADD    r0,r0,#1 ; pointeur sur caractère suivant
      JZ     r4,fin   ; fin de la chaîne ?
      SUB    r31,r4,#'y' ; caractère égal à 'y' ?
      JZ     r31,CARy ; si oui, aller au groupe de y
      SUB    r31,r4,#'x' ; caractère égal à 'x' ?
      JNZ    r31,ici  ; sinon reboucler
CARx:  LDB    r4,(r0) ; charger le caractère suivant
      ADD    r0,r0,#1 ; pointeur sur caractère suivant
      SUB    r31,r4,#'x' ; est-ce toujours un 'x' ?
      JZ     r31,CARx ; si oui, reboucler sur le groupe
      ADD    r2,r2,#1 ; sinon fin d'un groupe de x
      JZ     r4,fin   ; fin de la chaîne ?
      SUB    r31,r4,#'y' ; caractère égal à 'y' ?
      JNZ    r31,ici  ; reboucler si ce n'est pas le cas
CARY:  LDB    r4,(r0) ; charger le caractère suivant
      ADD    r0,r0,#1 ; pointeur sur caractère suivant
      SUB    r31,r4,#'y' ; est-ce toujours un 'y' ?
      JZ     r31,CARY ; si oui, reboucler sur le groupe
      ADD    r3,r3,#1 ; sinon fin d'un groupe de y
      SUB    r31,r4,#'x' ; est-ce un 'x' ?
      JZ     r31,CARx ; si oui, passer au groupe de x
      JNZ    r4,ici   ; reboucler si ce n'est pas la fin de la chaîne
fin:   MVI    r1,#1   ; résultat égal à 1
      SUB    r31,r3,r2 ; comparer groupes de y et groupes de x
      JGE    r31,fin2 ; groupes de y ≥ groupes de x ?
      MVI    r1,#0   ; sinon résultat égal à 0
fin2:

```

Compter les caractères (version 3)

On peut encore simplifier l'algorithme car il suffit en fait de tester les changements d'un caractère par rapport au caractère précédent pour détecter les groupes :

- Si le caractère est identique au précédent, on passe au suivant.
- Sinon, s'il est égal à x, c'est le début d'un groupe de x.
- Sinon, s'il est égal à y, c'est le début d'un groupe de y.
- On passe au caractère suivant.

D'où le code suivant :

Listing 4.15

```

MVI    r2,#0      ; nombre de groupes de x
MVI    r3,#0      ; nombre de groupes de y
MVI    r4,#0      ; caractère précédent vide
ici:   MOV    r5,r4  ; sauvegarde du caractère précédent
      LDB    r4,(r0) ; charger caractère suivant
      ADD    r0,r0,#1 ; pointeur sur caractère suivant
      JZ     r4,fin   ; fin de la chaîne ?
      SUB    r31,r5,r4 ; identique au caractère précédent ?
      JZ     r31,ici  ; si oui, passer au prochain
      SUB    r31,r4,#'x' ; est-il différent de 'x' ?
      JNZ    r31,testY ; si oui, tester si c'est 'y'
      ADD    r2,r2,#1 ; sinon début d'un groupe de x
      JMP    ici      ; boucler sur le caractère suivant
testY: SUB    r31,r4,#'y' ; comparer le caractère à 'y'
      JNZ    r31,ici  ; sinon reboucler
      ADD    r3,r3,#1 ; si oui, début d'un groupe de y
      JMP    ici      ; boucler sur le caractère suivant
fin:   MVI    r1,#1   ; résultat égal à 1
      SUB    r31,r3,r2 ; comparer groupes de y et groupes de x
      JGE    r31,fin2 ; groupes de y ≥ groupes de x ?
      MVI    r1,#0   ; sinon résultat égal à 0
fin2:

```

Compter les caractères (version 4)

Exercice 9 : Compter les mots

Dans *r0* se trouve l'adresse d'une chaîne de caractères terminée par l'octet nul. Écrivez une suite d'instructions assembleur qui, lorsqu'elle se termine, permet de récupérer le nombre de mots de la chaîne dans *r1*. Un mot est un ensemble de caractères ne comprenant pas le caractère espace (32 en décimal). Attention, tous les cas sont possibles : la chaîne peut être vide, il peut n'y avoir que des espaces dans la chaîne, il peut y avoir plusieurs espaces entre deux mots et il n'y a pas forcément d'espace avant le premier mot ni après le dernier.

Il suffit de faire deux boucles en les alternant : sur toutes les espaces successives puis sur tous les autres caractères qui se suivent. Il faut tester chaque caractère pour voir s'il n'est pas nul (fin de la chaîne).

Listing 4.16

```

    MVI    r1,#0      ; compteur égal à 0
loop1: LDB    r2,(r0)   ; lire un caractère
        ADD    r0,r0,#1 ; avancer le pointeur
        JZ     r2,fin   ; fin de la chaîne ?
        SUB    r31,r2,#32 ; une espace ?
        JZ     r31,loop1 ; dans ce cas, reboucler
        ADD    r1,r1,#1 ; sinon début d'un mot
loop2: LDB    r2,(r0)   ; lire un caractère
        ADD    r0,r0,#1 ; avancer le pointeur
        JZ     r2,fin   ; fin de la chaîne ?
        SUB    r31,r2,#32 ; une espace ?
        JNZ    r31,loop2 ; sinon reboucler sur le mot
        JMP    loop1    ; sinon revenir au début
fin:
```

Compter les mots

Exercice 10 : Compter les sous-chaînes

Une première chaîne de caractères est stockée en mémoire. Chaque élément occupe 1 octet et aucun n'est nul. L'octet qui suit le dernier élément de la chaîne est nul. L'adresse du premier élément de la chaîne se trouve dans le registre *r0*. Une seconde chaîne de caractères de caractéristiques identiques est stockée en mémoire et l'adresse du premier élément est dans *r1*. On souhaite compter le nombre d'occurrences de la seconde chaîne dans la première (c'est-à-dire le nombre de fois que cette chaîne apparaît dans la première) et récupérer ce résultat dans *r2*. Ainsi, dans la chaîne *abcabcbdbce*, on trouvera trois occurrences de la chaîne *bc*. On ne comptera que les occurrences distinctes : dans les chaînes *baabaaaab* et *baabaabaab*, on ne trouvera que trois occurrences de *aa*.

Écrivez une suite d'instructions assembleur qui, lorsqu'elle se termine, permet de récupérer dans *r2* le nombre d'occurrences de la chaîne pointée par *r1* dans la chaîne pointée par *r0*.

La difficulté de l'exercice est que l'on peut trouver le début de la seconde chaîne dans la première sans l'avoir en entier. Il faut donc mémoriser la position dans la première chaîne, chercher la seconde, et si on ne la trouve pas en entier, passer simplement au caractère suivant.

Listing 4.17

```

    MVI    r2,#0      ; compteur égal à 0
loop:  LDB    r3,(r0)   ; charger un caractère chaîne 1
        JZ     r3,fin   ; fin de la chaîne ?
        LDB    r4,(r1)   ; charger un caractère chaîne 2
        SUB    r31,r3,r4 ; comparer les deux caractères
        JZ     r31,match ; sont-ils identiques ?
        ADD    r0,r0,#1 ; sinon avancer le pointeur chaîne 1
        JMP    loop     ; et passer au caractère suivant
match: MOV    r5,r0     ; pointeur temporaire chaîne 1
        MOV    r6,r1     ; pointeur temporaire chaîne 2
suite: ADD    r5,r5,#1   ; avancer pointeur temporaire 1
        LDB    r3,(r5)   ; et charger le caractère
        ADD    r6,r6,#1 ; avancer pointeur temporaire 2
        LDB    r4,(r6)   ; et charger le caractère
        JZ     r4,trouve ; fin de la chaîne 2 ?
        SUB    r31,r3,r4 ; comparer les deux caractères
```

```

        JZ     r31,suite    ; sont-ils toujours identiques ?
        ADD    r0,r0,#1     ; sinon passer au caractère suivant
        JMP    loop        ; dans chaîne 1 et recommencer
trouve:  ADD    r2,r2,#1     ; une occurrence de plus
        MOV    r0,r5        ; continuer dans chaîne 1
        JMP    loop        ; après la chaîne trouvée
fin:

```

Compter les sous-chaînes

Lorsque l'on a trouvé un début de correspondance entre les deux chaînes, on utilise deux pointeurs temporaires *r5* et *r6* pour tester l'égalité exacte. Cela permet de ne pas perdre le début de la seconde chaîne (stockée dans *r1*) ni l'endroit où l'on est dans la chaîne 1 (car la seconde chaîne pourrait commencer au caractère suivant, comme lorsque l'on cherche aab dans la chaîne aaab).

Exercice 11 : Permuter des caractères

Une chaîne de caractères est stockée en mémoire. Chaque caractère occupe 1 octet et l'octet qui suit le dernier caractère de la chaîne est nul. L'adresse du premier élément de la chaîne se trouve dans le registre *r0*. Un tableau de nombres positifs est également stocké en mémoire, chaque nombre sur 1 octet, la valeur de l'octet étant la valeur numérique du nombre (et pas son caractère codé en ASCII) et l'octet qui suit le dernier nombre est nul. L'adresse du premier nombre du tableau se trouve dans le registre *r1*. Le tableau de nombres et la chaîne ont la même longueur.

On souhaite permuter les caractères de la première chaîne suivant les valeurs trouvées dans le tableau. Plus précisément, les éléments du tableau donnent successivement l'indice du caractère à prendre dans la chaîne initiale pour constituer la chaîne finale. Par exemple, si la chaîne est abcde et si le tableau de nombres contient les valeurs 3 1 5 4 2, on veut obtenir comme résultat caedb. On stockera la nouvelle chaîne à partir de l'adresse contenue dans *r2*.

Écrivez une suite d'instructions assembleur qui, lorsqu'elle se termine, assure que la chaîne pointée par *r2* est le résultat de la permutation.

Listing 4.18

```

loop:   LDB     r3,(r1)      ; charger la valeur numérique
        JZ      r3,fin      ; est-ce fini ?
        ADD     r4,r0,r3     ; sinon pointer sur le caractère voulu
        SUB     r4,r4,#1     ; en retranchant 1
        LDB     r5,(r4)      ; on le récupère
        STB     (r2),r5      ; et on le stocke
        ADD     r1,r1,#1     ; passer au nombre suivant
        ADD     r2,r2,#1     ; et avancer le pointeur
        JMP     loop        ; passer au caractère suivant
fin:    STB     (r2),r3      ; le zéro final est déjà dans r3

```

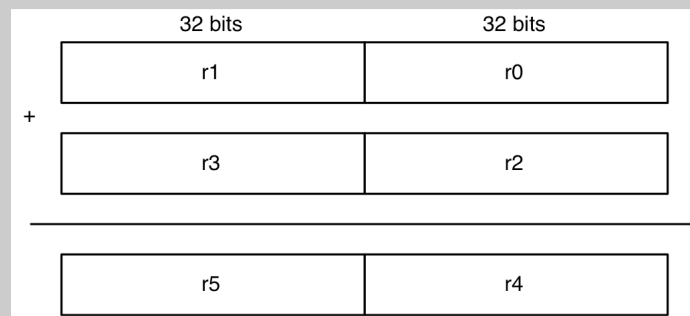
Permuter des caractères

Il suffit de récupérer le nombre et de l'additionner à *r0*, qui pointe toujours sur le début de la chaîne (en retranchant 1 car le nombre 1 correspond au premier caractère dont l'adresse est justement dans *r0*, le nombre 2 au caractère situé un cran plus loin...). On avance nombre par nombre dans le tableau et caractère par caractère dans la chaîne en construction, jusqu'à arriver à l'octet nul à la fin du tableau. Il faut alors le recopier dans la deuxième chaîne pour la terminer.

Exercice 12 : Additionner sur 64 bits

Un premier nombre entier de 64 bits est mémorisé dans les registres *r0* et *r1* (les 32 bits de poids faible dans *r0* et les 32 de poids fort dans *r1*). Un second nombre de 64 bits est mémorisé de la même façon dans *r2* et *r3* (*r2* ayant les 32 bits de poids faible).

Écrivez une suite d'instructions assembleur qui additionne les deux nombres et met le résultat sur 64 bits (voir figure 4.9) dans les registres *r4* (poids faible) et *r5* (poids fort).

Figure 4.9

Addition sur 64 bits.

Listing 4.19

```

ADD  r4,r0,r2    ; addition des 32 bits de poids faible
CCR  r31         ; sauvegarde de la retenue
ADD  r5,r1,r3    ; addition des 32 bits de poids fort
ADD  r5,r5,r31   ; et addition de la retenue intermédiaire

```

Addition sur 64 bits

On travaille sur 32 bits à la fois en effectuant les additions entre les registres. Il faut juste penser à la retenue intermédiaire, obtenue après l'addition des 32 bits de poids faible, qui doit s'ajouter à l'addition des 32 bits de poids fort.

Exercice 13 : Trier une liste

On souhaite trier par ordre décroissant une liste de six nombres entiers strictement positifs (à vous de les choisir), occupant 16 bits chacun et stockés en mémoire. La liste se termine par un 0. On prévoit un emplacement où la mettre une fois triée.

Écrivez une suite d'instructions assembleur qui trie la liste par ordre décroissant (et qui termine la liste résultante par 0).

Il faut définir la liste avec la directive `.HALF` (car chaque nombre est sur 2 octets) et réserver l'emplacement de la liste triée avec `.RES`.

Pour trier la liste, on imbrique deux boucles : la première travaille tant qu'il reste des nombres dans la liste, la seconde recherche le plus grand élément non encore trié.

Listing 4.20

```

liste: .HALF 10,5,3,11,4,7,0    ; voici une liste de nombres
triee: .RES 14                 ; 2 octets pour 7 nombres
      MVI r10,#triee          ; récupérer adresse liste triée
      MVI r30,#-1             ; pour remplacer l'élément trié
boucle: MVI r0,#liste          ; récupérer adresse liste
      MVI r1,#0               ; plus grand nombre égal à 0
loop:  LDH r2,(r0)             ; récupérer nombre suivant
      JZ  r2,next             ; fin de la liste
      SUB r31,r2,r1           ; comparer r2 et r1
      JLE r31,petit           ; saut si r2 ≤ r1
      MOV r1,r2               ; sinon meilleur = nombre
      MOV r3,r0               ; mémoriser son adresse
petit: ADD r0,r0,#2            ; passer au nombre suivant
      JMP loop                ; reboucler dans le tableau
next:  JZ  r1,fin              ; on a le plus grand ; si 0, c'est fini
      STH (r3),r30            ; on le remplace par -1
      STH (r10),r1            ; on le met dans la liste triée
      ADD r10,r10,#2          ; nombre suivant dans la liste triée
      JMP boucle              ; passer au nombre suivant
fin:   STH (r10),r1           ; mettre le zéro final

```

Trier une liste

La boucle interne cherche le plus grand nombre de la liste. Quand il est trouvé, il est mémorisé dans *r1* et son adresse dans *r3*. À la fin de la boucle, on recopie ce nombre dans la liste triée (pointée par *r10*, que l'on incrémente après) et on le remplace dans la première liste par -1. On réitère tant qu'il y a des nombres dans la première liste : si le plus grand nombre est égal à 0, cela signifie que l'on a tout trié.

Exercice 14 : Trier une liste par appel de fonction

On reprend l'exercice précédent en le scindant en deux :

- écriture d'une fonction indépendante qui renvoie le plus grand élément d'une liste ;
- tri de la liste par appels successifs à la fonction précédente.

1) Écrivez une fonction (appelée par CALL et donc se terminant par RET, voir section 4) qui récupère sur la pile (instructions PUSH et POP, voir section 4) l'adresse d'une liste (composée d'entiers strictement positifs sur 16 bits et se terminant par 0) et renvoie son plus grand élément (qui sera remplacé par -1 dans la liste) en l'empilant avant la fin de la fonction.

2) Écrivez une suite d'instructions assembleur qui trie par ordre décroissant la liste définie dans l'exercice 13 (et qui termine la liste résultante par 0). Utilisez la fonction précédente pour trouver le plus grand élément restant.

1) Le code correspond à la boucle intérieure de l'exercice précédent, avec l'ajout de la gestion de la pile. L'adresse de retour est la dernière valeur empilée. Il faut donc la dépiler pour récupérer l'adresse de la liste, et empiler le plus grand élément avant de remettre l'adresse de retour sur la pile.

Listing 4.21

```
Grand: POP    r29      ; sauvegarder l'adresse de retour
      POP    r0       ; récupérer l'adresse de la liste
      MVI    r30,#-1  ; pour remplacer l'élément trié
boucle: MVI    r1,#0   ; plus grand nombre égal à 0
loop:  LDH    r2,(r0)  ; récupérer nombre suivant
      JZ     r2,next   ; fin de la liste
      SUB    r31,r2,r1 ; comparer r2 et r1
      JLE    r31,petit ; saut si r2 ≤ r1
      MOV    r1,r2     ; sinon meilleur = nombre
      MOV    r3,r0     ; mémoriser son adresse
petit: ADD    r0,r0,#2  ; passer au nombre suivant
      JMP    loop      ; reboucler dans le tableau
next:  JZ     r1,fin    ; on a le plus grand ; si 0, c'est fini
      STH    (r3),r30  ; on le remplace par -1
fin:   PUSH   r1       ; empiler le plus grand élément
      PUSH   r29      ; remettre l'adresse de retour
      RET                ; fin de la fonction
```

Trouver le plus grand élément avec une fonction

2) On reprend là encore l'exercice précédent en remplaçant la boucle par l'appel de la fonction.

Listing 4.22

```
liste: .HALF 10,5,3,11,4,7,0 ; voici une liste de nombres
triee: .RES 14                ; 2 octets pour 7 nombres
      MVI    r10,#triee ; récupérer adresse liste triée
boucle: MVI    r0,#liste  ; récupérer adresse liste
      PUSH   r0          ; empiler l'adresse
      CALL   Grand       ; appeler la fonction
      POP    r1          ; dépiler le plus grand élément
      STH    (r10),r1    ; on le met dans la liste triée
      ADD    r10,r10,#2  ; nombre suivant dans la liste triée
      JNZ    r1,boucle  ; si ce n'est pas zéro, ce n'est pas fini
```

Trier une liste par appels de fonction

La fonction modifie certains registres. Si on voulait être perfectionniste, il faudrait sauvegarder les valeurs de ces registres avant toute modification et les recharger à la fin de la fonction.

Chapitre 5

Mémoire

Composant indispensable de l'ordinateur, la mémoire se résume souvent à une simple fonction de stockage. On distingue plusieurs catégories de mémoires, différenciées par leurs caractéristiques (adressage, performances, accès...) : mémoire principale, secondaire, etc.

La mémoire principale est elle-même composée de types différents (ROM, SRAM, DRAM, Flash...). Elle sert à stocker le code et les données lors de l'exécution d'un programme. Ces données peuvent avoir un statut de variables globales ou locales, modifiant la façon dont elles vont être mémorisées et donc utilisées par le programmeur.

Mémoire

- 1. Caractéristiques..... 116
- 2. Mémoire à semi-conducteurs..... 119
- 3. Programmation et stockage en mémoire..... 123

Problèmes et exercices

- 1. Calculer le temps de propagation..... 127
- 2. Transférer des données..... 127
- 3. Calculer les performances d'un disque dur..... 127
- 4. Renvoyer une adresse..... 128
- 5. Comprendre le ramasse-miettes 131
- 6. Choisir l'ordre des octets 132
- 7. Incrémenter une variable..... 133

I. CARACTÉRISTIQUES

Il existe différentes zones de stockage de l'information au sein de l'ordinateur : registres, mémoire principale, mémoires secondaires, etc. Chacun de ces espaces se différencie par des caractéristiques propres (capacité, adressage, méthode d'accès, performances...), qui déterminent leur utilisation.

I.1 Localisation

Les localisations géographique et logique des mémoires influent directement sur leurs performances. Plus une mémoire est loin du processeur, plus les signaux électriques mettent du temps à l'atteindre. À l'échelle du processeur, les distances de l'ordre de la dizaine de centimètres ne sont pas négligeables en matière de propagation de courant, d'autant plus que ces composants ne lui sont pas reliés directement mais *via* des circuits intermédiaires induisant des retards supplémentaires.

On distingue quatre groupes de mémoires en fonction de leur localisation :

- Au cœur du processeur sur la puce en silicium figurent les registres ainsi que la mémoire cache de niveau 1. Cet emplacement permet un accès rapide aux données.
- Reliée directement au processeur par un bus spécial, la mémoire cache de niveau 2 permet ainsi de maximiser la vitesse des transferts. Anciennement placée à côté du circuit intégré formant le processeur, elle lui est maintenant intégrée. On trouve parfois un cache de niveau 3 extérieur à la puce ou même intégré (voir chapitre 6).
- Reliée *via* un contrôleur mémoire chargé de répartir les mouvements de données entre les différents composants (voir chapitre 3), la mémoire principale est aussi parfois, suivant les implémentations, reliée directement au processeur.
- Reliées par le biais d'un bus d'entrées/sorties, plutôt lent, toutes les mémoires secondaires (disque dur, CD/DVD, bandes, clés amovibles...) utilisent en outre un ou plusieurs contrôleurs systèmes pour communiquer avec le processeur. Évidemment, cela ralentit fortement les échanges. Mais de toute façon, les périphériques ont un mode de fonctionnement beaucoup plus lent que celui du processeur.

I.2 Adressage et capacité

Chaque espace mémoire possède une capacité mesurée en octets. On a classiquement les valeurs suivantes :

Tableau 5.1

Registre	64 octets à 1 Ko
Cache de niveau 1	32 Ko à 128 Ko
Cache de niveau 2	512 Ko à 8 Mo
Cache de niveau 3	3 à 10 Mo
Mémoire principale	512 Mo à 8 Go
Disque dur	250 Go à 2 To
CD	700 Mo
DVD, DVD blu-ray	4,7 Go à 50 Go
Clé amovible	1 Go à 32 Go
Bande	10 Go

Capacités standard des mémoires

Chaque information est stockée dans une case mémoire que l'on référence *via* son adresse. Pour un registre, on précise son nom dans l'instruction. Pour un octet stocké en mémoire principale, on donne son adresse numérique. La localisation sur un disque se fait *via* un numéro de piste, de secteur et de décalage (voir chapitre 8). La mémoire cache accélère les transferts depuis la mémoire principale (voir chapitre 6) ; il ne faut

donc pas la considérer comme un espace de stockage supplémentaire, mais comme un simple dispositif matériel géré automatiquement, auquel le programmeur n'a pas accès.

Note

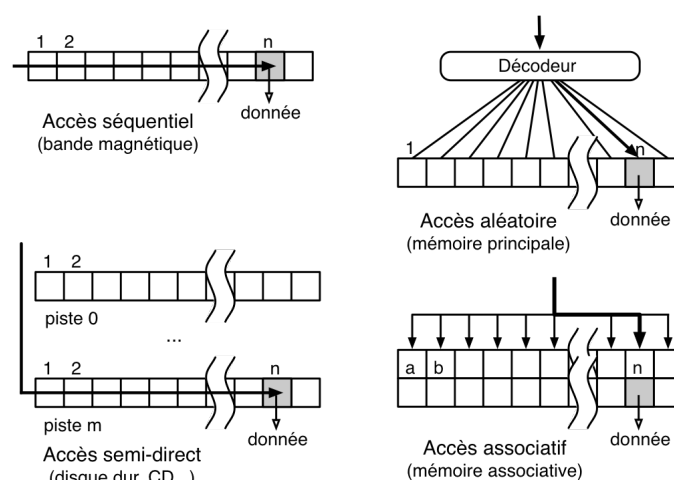
La taille d'une case mémoire est l'octet. Celle d'un registre est de 32 ou 64 bits, soit 4 ou 8 octets (voire 128 bits, soit 16 octets, pour certains calculs mathématiques). Celle des instructions est variable (souvent de 2 à 10 octets environ) ou fixe (souvent 4 octets). Les mémoires cache et principale sont reliées au processeur de sorte à pouvoir envoyer plusieurs octets successifs à la fois (16 ou 32) afin de maximiser le débit et de ne pas laisser le processeur en attente de données.

1.3 Méthode d'accès

Même si l'on connaît son adresse, l'accès à une case mémoire se fait de façon différente suivant les supports, en raison de leur construction distincte (voir figure 5.1) :

- **Accès séquentiel.** Cet accès, le plus lent, est celui des bandes magnétiques. Il oblige à parcourir l'intégralité des adresses avant d'arriver à celle voulue : il faut dérouler toute la bande depuis le début (éventuellement en accéléré) pour aboutir à la case souhaitée.
- **Accès semi-direct.** En 1956, IBM invente le disque dur, une révolution dans les systèmes de stockage, qui permet un accès plus rapide par sa structure à deux dimensions. Il est constitué de pistes concentriques, tout comme les autres supports sous forme de disque (CD, DVD). Tous permettent un accès direct à la piste voulue (par le déplacement d'une tête de lecture), avant un accès séquentiel à la donnée via le défilement de la piste (sous la tête de lecture).
- **Accès direct ou aléatoire.** Intel invente la mémoire principale sous forme de semi-conducteurs en 1970. La grande innovation est de permettre un accès direct (et donc rapide) à la donnée via un décodeur d'adresse. À partir de la valeur numérique de l'adresse, ce circuit sélectionne la cellule mémoire (stockant 1 octet ou 1 bit) souhaitée et renvoie sa valeur (voir section 2). Cet accès direct est aussi dit « aléatoire » (ce qui explique le terme de mémoire RAM, *Random Access Memory*). Les registres et la mémoire principale fonctionnent de cette façon.
- **Accès associatif.** Les mémoires à accès associatif enregistrent des couples de valeurs. Au lieu de sélectionner une case mémoire via son adresse, on envoie au circuit mémoire la première valeur d'un couple et, si cette valeur existe au sein d'un couple mémorisé, la mémoire renvoie la seconde valeur dudit couple. Cela permet de mémoriser des correspondances quelconques de valeurs (et pas seulement des correspondances adresse-valeur stockée). Plusieurs mécanismes de mémoire cache ou de mémoire virtuelle (chapitres 6 et 7) font un usage intensif de tables de correspondances stockées dans des mémoires de ce type.

Figure 5.1



Différentes méthodes d'accès.

I.4 Performances

La mesure des performances de la mémoire est compliquée par le fait que l'accès n'est pas uniforme : suivant que l'on récupère un ou plusieurs octets, situés ou non sur le même boîtier dans le cas d'un accès aléatoire, on peut avoir des valeurs complètement différentes.

Mémoire à accès aléatoire

Historiquement, deux valeurs correspondaient aux performances des mémoires à accès aléatoire : le temps d'accès et le temps de cycle. Le premier est le délai entre la présentation de l'adresse et la disponibilité de la donnée. Les registres ont un temps d'accès de l'ordre de la nanoseconde, la mémoire cache de quelques nanosecondes et la mémoire principale de quelques dizaines de nanosecondes.

Après un accès, les boîtiers mémoire ont souvent besoin d'un certain temps avant de pouvoir accepter un nouvel accès, qui permet aux circuits électroniques de revenir à leur état initial. On définit alors le temps de cycle comme étant l'intervalle de temps minimum entre deux accès successifs à un même boîtier. Pour la mémoire principale, ce temps de latence est de quelques nanosecondes.

Le temps d'accès étant assez important par rapport à l'horloge du processeur. Les innovations technologiques se sont multipliées depuis le début des années 1990 pour accélérer la production de données par un boîtier mémoire : répartition des données successives sur des boîtiers différents pour économiser le temps de relaxation ou envoi des données successives en mémoire, après un premier accès, sans attendre la suite d'adresses correspondantes.

Dans tous les cas, ces boîtiers mémoire seront reliés au processeur et l'on indique souvent les performances d'un système en donnant la bande passante du lien. Il s'agit du débit maximum en bits par seconde des informations circulant entre les deux composants ; c'est également le produit de la fréquence de fonctionnement de la mémoire et du nombre d'octets transférés à chaque opération. Ainsi, un lien reliant une mémoire et un processeur, fonctionnant à 500 MHz et transférant 8 octets à la fois (par cycle d'horloge), a une bande passante de 4 Go/s.

Mémoire à accès semi-direct

Trois paramètres peuvent caractériser les performances d'un disque :

- **Le temps de positionnement (ou temps d'accès).** Il s'agit du temps nécessaire pour déplacer la tête de lecture suivant un rayon du disque pour arriver à la piste voulue. L'ordre de grandeur est de 5 à 10 millisecondes pour un disque dur.
- **Le temps de latence.** Une fois la tête arrivée devant la bonne piste, il faut attendre que la piste défile sous la tête jusqu'à l'information voulue. Cela prend en moyenne une demi-rotation, soit un ordre de grandeur de 5 millisecondes. Il est plutôt d'usage de donner la vitesse de rotation du disque en tours par seconde pour mesurer ce paramètre : plus cette vitesse est rapide, plus le temps de latence est faible.
- **Le taux de transfert.** Une fois que l'on commence à transférer l'information, il est facile de continuer à le faire puisque le disque tourne et que les bits défilent sous la tête. On peut alors mesurer un taux de transfert en bits par seconde. Celui-ci va d'une dizaine à une cinquantaine de mégaoctets par seconde.

Suivant l'usage et l'accès au disque, ces paramètres ont plus ou moins d'importance : combien d'octets souhaite-t-on transférer et où se trouvent-ils sur le disque (impliquent-ils de nombreux déplacements de la tête de lecture ou non) ? Si l'on désire transférer des octets situés à des emplacements éloignés sur le disque, la tête de lecture doit effectuer de nombreux déplacements et le temps total est fortement dépendant du temps de positionnement. À l'inverse, si les données se suivent, c'est plutôt le taux de transfert qui est prépondérant.

I.5 Caractéristiques physiques

Dans les premières années de l'informatique, la mémorisation des informations lors de l'exécution des programmes était le problème le plus difficile auquel étaient confrontés les concepteurs de machines : il n'y avait pas de manière simple de stocker le courant électrique. On a utilisé des ondes acoustiques se propageant dans des tubes de mercure, l'illumination de grains de phosphore sur l'écran de tubes cathodiques (dans ces deux cas, la mémorisation était provisoire et il fallait régénérer régulièrement le signal) ou l'aimantation de tores de ferite.

De nos jours, la mémorisation des données fait appel aux propriétés électroniques des matériaux semi-conducteurs pour les registres, les mémoires cache et principale, à des propriétés magnétiques pour les disques durs et les bandes, et à des propriétés optiques pour les CD et les DVD (voir chapitre 8).

Deux autres caractéristiques sont importantes pour différencier les mémoires : la volatilité et le fait d'être modifiable. Une mémoire volatile perd l'information lorsque l'alimentation électrique est interrompue. C'est le cas des registres, de la mémoire cache et d'une importante fraction de la mémoire principale. Toutes les mémoires secondaires (disque dur, CD, DVD, bande, clef) ainsi que certaines parties de la mémoire principale (voir section 2) sont non volatiles. Toutes les mémoires volatiles sont heureusement modifiables, mais certains supports de mémoire non volatile empêchent toute modification postérieurement à l'initialisation : CD-ROM, DVD-ROM et mémoires ROM (voir section 2).

2. MÉMOIRE À SEMI-CONDUCTEURS

Toute la mémoire à accès rapide de l'ordinateur est composée de cellules mémoire fondées sur un support semi-conducteur. En fonction des usages, il existe différents types de mémoires électroniques, qui ont chacun leurs caractéristiques et leur place dans l'ordinateur.

2.1 Différents types de mémoires

Typologie

On distingue plusieurs types de mémoires (résumées tableau 5.2) :

- **RAM** (*Random Access Memory*, mémoire à accès aléatoire). Il s'agit de la mémoire la plus classique de l'ordinateur (aussi appelée « mémoire vive »). Elle constitue la grande majorité de la mémoire principale. Elle est composée de cellules mémoire standard volatiles. Elle sert à mémoriser toutes les instructions et les données nécessaires au bon fonctionnement des programmes. L'initialisation et les modifications de l'information se font grâce au courant électrique.
- **ROM** (*Read Only Memory*, mémoire à lecture seule). À l'inverse, la ROM (appelée « mémoire morte ») a son contenu figé lors de la fabrication du circuit intégré. Chaque cellule mémoire contient ou non une liaison suivant l'information souhaitée, 0 ou 1. Comme cette liaison est créée à la conception, toute modification ultérieure est impossible. En outre, la fabrication d'un circuit ROM étant personnalisée (par la présence de ces liaisons), une diffusion en petite série n'est pas envisageable car la production nécessite un matériel lourd dans de coûteuses usines.
- **PROM** (*Programmable ROM*, mémoire à lecture seule programmable). À l'image d'une ROM, une PROM est dotée lors de sa fabrication d'une liaison dans chaque cellule. Cette liaison est en fait un fusible que l'utilisateur peut détruire. Une PROM est vendue vierge (elle a donc un coût plus faible puisque tous les circuits fabriqués sont identiques) et un appareillage électrique simple permet d'éliminer les fusibles dans certaines cases mémoire suivant les choix de l'utilisateur. Il suffit pour cela de placer le boîtier PROM sur l'appareillage et de le programmer. L'avantage est un coût bien plus intéressant puisque la programmation des informations stockées ne nécessite plus un passage en usine, mais un simple appareil. En revanche, l'inconvénient reste le même : une fois la programmation faite, il est impossible de modifier les données mémorisées car l'on ne peut pas remettre un fusible microscopique dans la cellule.
- **EPROM** (*Erasable PROM*, PROM effaçable). Un bit est maintenant stocké sous la forme de charges électriques localisées dans une cellule mémoire, isolées du reste du circuit. En raison de cette isolation, la mémoire est non volatile : les charges ne disparaissent pas lorsque l'alimentation est coupée. Mais on peut les enlever manuellement, ce qui est impossible sur les ROM et les PROM. Pour ce faire, on expose le boîtier mémoire plusieurs dizaines de minutes à un rayonnement ultraviolet, qui « décharge » les cellules. Ensuite, à l'aide d'un appareil de programmation, on réinjecte des charges dans les cellules souhaitées pour mémoriser à nouveau des informations. On reconnaît ces mémoires à la présence d'une fenêtre en quartz (qui laisse passer les ultraviolets) sur le dessus du boîtier, à travers laquelle on voit la puce électronique.
- **EEPROM** (*Electrically Erasable PROM*, PROM effaçable électriquement). L'inconvénient des EPROM est la nécessité d'effectuer des manipulations pour effacer l'information et la reprogrammer : il faut extraire le boîtier de son support, l'exposer à un rayonnement ultraviolet, le placer sur l'appareil de programmation avant de le réinstaller. Les EEPROM suppriment ces manipulations en permettant un effacement électrique des informations : un certain type de courant permet l'écriture de l'information par stockage de charges tandis qu'un autre type les fait disparaître. L'EEPROM n'a plus besoin d'être sortie de son logement pour subir les modifications souhaitées. Les mémoires dites « flash » fonctionnent également de cette façon. Les EEPROM sont plus lentes que les RAM et ne supportent pas une infinité de cycles de lecture/écriture : chaque effacement de données provoque une oxydation du support, garanti pour

environ cent mille écritures. On ne peut donc pas les utiliser comme mémoire vive principale (qui peut potentiellement changer de données à chaque instruction, donc des milliards de fois par seconde), mais comme support de mémorisation susceptible d'être modifié de temps en temps.

Tableau 5.2

Type	Accès	Modifications	Initialisation	Volatile
RAM	Lecture/écriture	Électrique	Électrique	Oui
ROM	Lecture seule	Non	Masque	Non
PROM	Lecture seule	Non	Électrique	Non
EPROM	Essentiellement lecture	Ultraviolets	Électrique	Non
EEPROM Flash	Essentiellement lecture	Électrique	Électrique	Non

Différents types de mémoires

Usage

La RAM est la mémoire de travail de l'ordinateur : on y stocke les instructions et les données, chaque cellule mémoire pouvant être modifiée des milliards de fois par seconde sans aucun inconvénient.

Les boîtiers ROM et PROM ne peuvent servir qu'à mémoriser de façon permanente des informations figées. Voici quelques exemples d'utilisation de ces mémoires mortes au sein de l'ordinateur :

- Un séquenceur microprogrammé traite chaque instruction en exécutant des micro-instructions qui effectuent chacune un travail élémentaire dans le processeur (voir chapitre 3). Chaque instruction correspond donc à un microprogramme mémorisé dans une mémoire du processeur réalisée lors de sa fabrication. Ces microprogrammes ne doivent évidemment pas disparaître et n'ont pas à être changés : c'est un cas typique d'utilisation de ROM.
- Les opérations d'entrées/sorties nécessitent des instructions processeur spéciales, dépendant du matériel. Il est donc normal qu'un fabricant d'ordinateur propose des fonctions de base déjà programmées permettant d'effectuer des entrées/sorties simples à partir desquelles un logiciel peut élaborer des opérations plus complexes. Une partie de la mémoire principale de l'ordinateur est donc composée de mémoire morte conservant les instructions qui forment ces fonctions de base.
- Lorsque l'ordinateur s'allume, le processeur cherche à exécuter des instructions. Malheureusement, la RAM étant volatile, elle ne contient aucune information viable lors de l'allumage. Là encore, une partie de la mémoire principale est sous forme de mémoire morte contenant un programme de démarrage dont l'unique tâche est, par exemple, de charger depuis un disque dur le reste du système d'exploitation.
- Certaines entrées/sorties sont suffisamment complexes pour nécessiter la présence d'un processeur auxiliaire qui leur est uniquement dédié. Comme tout processeur, celui-ci exécute des instructions. Il ne s'agit pas, dans ce cas, d'un programme utilisateur, mais d'instructions réalisant les entrées/sorties. Le fabricant les programme et les stocke sur une mémoire morte pour qu'elles ne disparaissent pas.

Le premier exemple utilise une ROM pour des questions de rapidité d'exécution des instructions. Les trois autres utilisent des ROM, PROM ou même des EPROM ou EEPROM. L'intérêt de ces deux derniers types de mémoires est de pouvoir changer les programmes mémorisés, lors de la correction de bogues ou de l'introduction de nouvelles fonctionnalités. Introduites antérieurement, les EPROM étaient utilisées dans les années 80 mais ont été supplantées par les EEPROM et les mémoires flash dont la souplesse d'utilisation (et surtout d'effacement) est sans commune mesure.

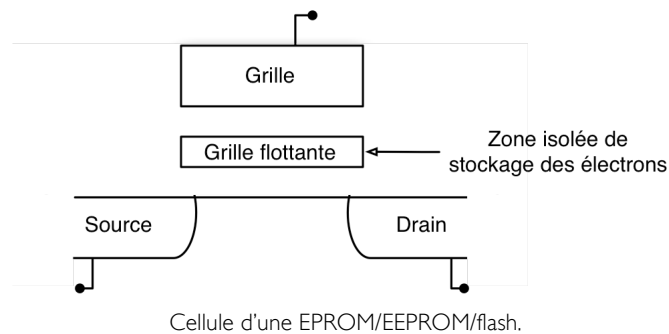
2.2 Organisation interne

L'unité élémentaire de stockage est le bit, mémorisé dans une cellule mémoire. Les cellules mémoire sont ensuite regroupées dans des puces mémoire permettant d'accéder à toutes les informations.

Stockage d'un bit

Chaque cellule mémoire permet le stockage d'un bit mais le procédé utilisé diffère suivant qu'il s'agit d'une ROM, d'une EPROM, etc. Les mémoires ROM et PROM mémorisent l'information par la simple présence ou absence de liaison entre deux connecteurs. Les EPROM, EEPROM et mémoires flash possèdent, dans chaque cellule, un transistor ayant une « grille flottante » isolée (voir figure 5.2).

Figure 5.2



Avec les bonnes différences de potentiel entre les trois électrodes, il est possible d'injecter des électrons dans la grille flottante, qui sont piégés, permettant ainsi de mémoriser un état 0 ou 1. Une autre combinaison de différences de potentiel (ou une exposition aux ultraviolets dans le cas des EPROM) permet de les éjecter de la grille flottante et d'effacer ainsi la cellule.

Dans le cas de la mémoire vive, deux technologies sont complémentaires : la RAM statique (SRAM) et la RAM dynamique (DRAM).

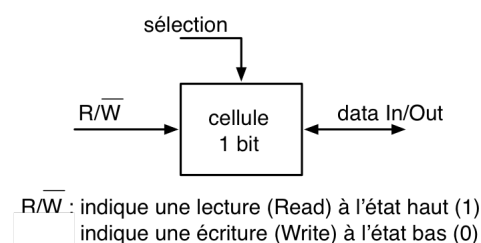
La cellule mémoire des RAM statiques est construite comme une bascule élémentaire (voir chapitre 2) mémorisant un bit de façon permanente tant que l'alimentation électrique est présente. Un avantage est que le temps d'accès est faible car le bit est stocké de manière active dans la cellule par la présence de courant dans la bascule : il y a donc une réserve d'énergie disponible qui peut positionner immédiatement les circuits lorsque l'on veut connaître la valeur stockée. La rapidité de la RAM statique en fait le candidat idéal pour construire les registres et la mémoire cache. L'inconvénient principal est qu'il faut de quatre à six transistors par cellule pour construire la bascule, ce qui explique la capacité plus faible des boîtiers de RAM statique.

La cellule des RAM dynamiques garde un bit sous forme de charges électriques dans un minuscule condensateur associé à un transistor de commande. Cette construction permet de réduire fortement la taille de la cellule (un transistor au lieu de quatre à six) et d'augmenter nettement la capacité maximale des boîtiers mémoire, utilisés alors comme mémoire principale. En revanche, cela amène deux handicaps :

- L'information étant stockée sous forme passive (présence ou absence de charges dans le condensateur), il faut un certain temps (lié à la décharge du condensateur) pour tester la donnée mémorisée. Les RAM dynamiques ont un temps d'accès plus élevé.
- D'inévitables fuites de courant déchargent le condensateur en quelques millisecondes. L'information est alors perdue. Pour éviter cela, il faut régulièrement « rafraîchir » la mémoire en lisant et en réécrivant toutes les données mémorisées, et ce avant qu'elles ne disparaissent (c'est-à-dire environ toutes les millisecondes). Cela peut paraître coûteux en temps mais à l'échelle temporelle du processeur (la nanoseconde), ce n'est pas handicapant.

Quel que soit le type de mémoire RAM, chaque cellule de mémorisation possède une ligne de donnée permettant de lire le bit stocké ou de l'écrire, une ligne de commande indiquant une lecture ou une écriture, une ligne de sélection permettant d'activer la cellule (voir figure 5.3).

Figure 5.3



Commande d'une cellule mémoire RAM.

Pour effectuer une lecture, on envoie le signal correspondant ($R/\overline{W} = 1$), on sélectionne la cellule (*sélection* = 1) et on récupère la donnée en sortie. Pour effectuer une écriture, on met le bit à mémoriser sur la ligne de donnée, on positionne le signal d'écriture ($R/\overline{W} = 0$) et on sélectionne la cellule.

Puce mémoire

Les cellules mémoire sont réunies dans une puce mémoire pour former une partie de l'espace adressable. L'ensemble des cellules d'une puce mémoire a la forme d'une matrice carrée (voir figure 5.4).

Figure 5.4

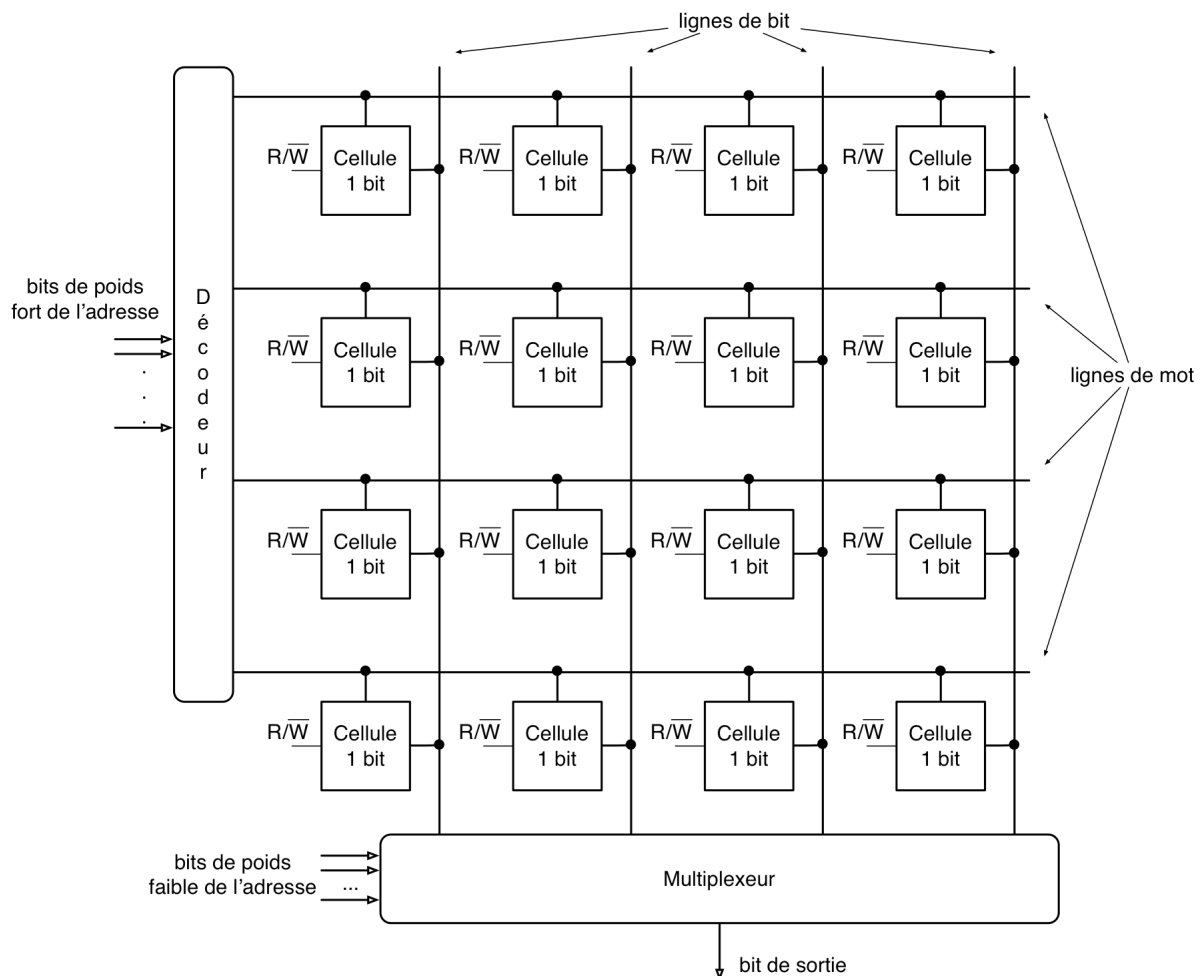


Schéma d'une puce mémoire.

Cette puce mémoire permet de stocker 2^n bits, sélectionnés par une adresse sur n bits. Pour récupérer un bit stocké, on envoie la moitié supérieure de l'adresse (les $n/2$ bits de poids fort) sur un décodeur, qui active alors une ligne de mot parmi $2^{n/2}$. Toutes les cellules situées sur la ligne de mot sont sélectionnées et envoient leur bit sur les lignes de bit correspondantes. Les $n/2$ bits de poids faible de l'adresse sont dirigés sur un multiplexeur, qui laisse passer sur la ligne de sortie le bit voulu uniquement, dont le numéro de colonne est égal à cette moitié d'adresse.

On construit ainsi une mémoire $2^n \times 1$ bits. Mais comme chaque adresse fait référence à 1 octet, il faut associer huit puces mémoire en parallèle, commandées par les mêmes lignes d'adresse, et générant chacune un des 8 bits de sortie (voir figure 5.5).

Figure 5.5

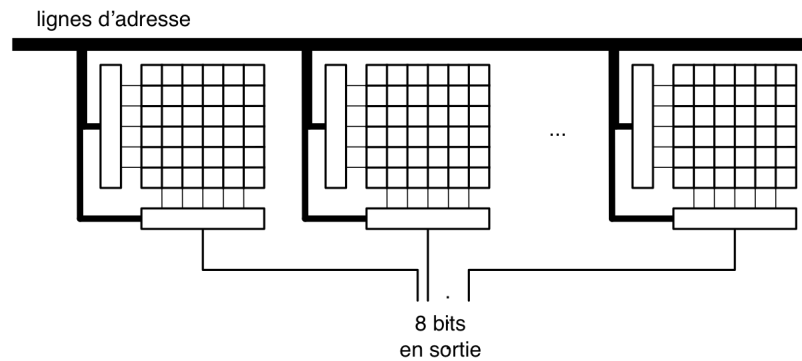
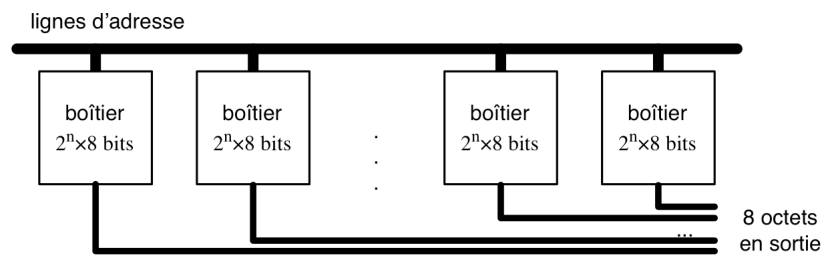


Schéma d'un module mémoire 8 bits.

Historiquement, les barrettes mémoire portaient toujours huit boîtiers mémoire, qui stockaient chacun $2^n \times 1$ bits, ce qui fournissait bien un module mémoire de 2^n octets. Maintenant, on fabrique des boîtiers mémoire intégrant ces huit matrices carrées et qui mémorisent donc $2^n \times 8$ bits. Cependant, on associe toujours plusieurs de ces boîtiers (souvent huit) pour accélérer les transferts entre la mémoire et le processeur : de la sorte, chaque accès mémoire récupère plusieurs octets successifs (voir figure 5.6).

Figure 5.6



Mémoire $2^n \times 8$ octets.

Note

Séparer en deux les bits de l'adresse (voir figure 5.4) peut aussi permettre d'économiser sur les broches de connexion des boîtiers sans perdre en rapidité d'accès. Au lieu d'avoir n connexions, on en garde la moitié. On envoie d'abord les bits de poids fort, ce qui active la ligne de mot et, pendant le temps d'accès de chaque cellule sur la ligne, on envoie sur ces mêmes connexions les bits de poids faible de l'adresse, qui sont dirigés vers le multiplexeur et arrivent à temps pour la sélection du bit voulu.

3. PROGRAMMATION ET STOCKAGE EN MÉMOIRE

Un programme s'exécutant sur un ordinateur utilise la mémoire pour son code exécutable et ses données. Chaque langage a sa propre manière de stocker les informations pendant leur utilisation, mais les découpages de l'espace mémoire sont standard. En revanche, il faut faire attention au stockage des éléments particuliers, qui peut différer suivant les machines.

3.1 Espace mémoire d'un programme

En plus du code exécutable, un programme a deux types de variables à stocker : les globales (qui peuvent exister pendant toute l'exécution) et les locales (qui n'existent que pendant l'appel d'une fonction). Les premières sont dans une zone mémoire appelée « tas » (*heap*), les secondes dans une pile (*stack*). Voyons ce qu'il en est plus précisément pour deux langages classiques.

Langage C

Dans le tas se trouvent les variables globales (définies en dehors de toute fonction) ainsi que les allocations mémoire (créées à l'aide de l'appel à la fonction `malloc()`). Cette zone évolue donc au fur et à mesure que le programme alloue et libère de la mémoire.

La pile contient les variables locales d'une fonction, ses paramètres ainsi que l'adresse de retour. Lors de l'appel d'une fonction, la fonction appelante empile les paramètres de l'appel et l'adresse de retour. Le contrôle est alors donné à la fonction appelée, qui empile ses variables locales en prévoyant pour chacune le nombre d'octets nécessaires en fonction de leur type. À la fin de l'exécution de la fonction, les variables locales, les paramètres et l'adresse de retour sont dépilés, et l'on ne garde que cette dernière pour reprendre l'exécution de la fonction appelante.

Considérons la fonction suivante :

```
void foo(int i) {
    char *ptr = malloc(100);
    ...
}
```

Lors de son appel, en plus de l'adresse de retour, 8 octets sont réservés sur la pile, 4 pour le paramètre `i` et 4 pour le pointeur `ptr`. Ce dernier contient l'adresse d'une zone de 100 octets réservée sur le tas.

Une fois créée sur le tas, une zone d'allocation n'est détruite que par un appel explicite à la fonction `free()`, qu'il ne faut pas oublier de faire sous peine de saturation de la mémoire.

Langage Java

Le tas correspond maintenant à la zone mémoire où sont créés tous les objets avec leurs variables d'instance (déclarées dans une classe, et non dans une méthode). Sur la pile se trouvent le code des méthodes (empilées dans l'ordre des appels) ainsi que leurs variables locales.

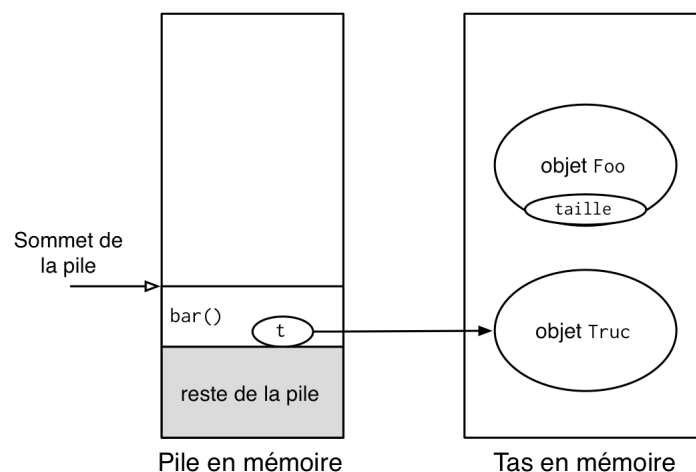
Considérons l'exemple suivant :

```
public class Foo {
    int taille;

    public void bar() {
        Truc t = new Truc(0);
    }
}
```

À sa création, un objet de la classe `Foo` est placé sur le tas avec sa variable d'instance `taille`. Lors de l'appel de la méthode `bar()`, la variable locale `t` est empilée avec le code de `bar()`. `t` est une référence (équivalant à un pointeur C) à un objet de la classe `Truc` créé sur le tas (voir figure 5.7).

Figure 5.7



Exemple de création d'objet Java.

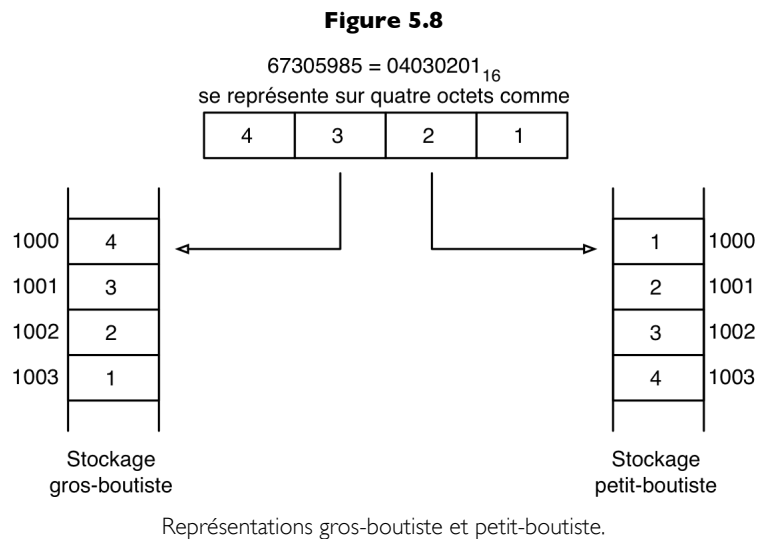
Il n'y a pas en Java de destruction explicite des objets. Le mécanisme de ramasse-miettes (*garbage collector*) supprime régulièrement du tas ceux qui ne sont plus référencés par une variable (et auxquels on ne peut donc plus accéder).

Le langage C++ possède des mécanismes similaires, à l'exception du ramasse-miettes : le programmeur doit explicitement détruire les objets créés, par appel à la fonction `delete()`.

3.2 Mémorisation d'un nombre

Les variables d'un programme sont mises en mémoire sans indication de leur type et cela amène à des résultats aberrants si le programmeur se trompe de type à l'affichage ou, pire, s'il essaie de stocker une valeur numérique plus grande que le maximum autorisé par le type de la variable (voir chapitre 1).

Il y a malheureusement un autre problème lié au stockage des valeurs numériques en mémoire, qui occupent 2, 4 voire 8 octets. Il est légitime de se demander si l'adresse d'une variable correspond à l'octet de poids fort, suivi des autres octets dans les cases suivantes, ou à l'octet de poids faible, les autres se mettant là encore aux adresses mémoire suivantes. Les deux solutions existent et chacune a été implémentée par certains processeurs. Ceux qui stockent les octets de poids fort d'abord sont appelés gros-boutistes (*big-endian*), les autres petit-boutistes (*little-endian*). La première catégorie regroupe les processeurs fabriqués par IBM et Motorola, la seconde ceux d'Intel (voir figure 5.8).



Cette différence de stockage des entiers est-elle importante ? Chaque processeur est cohérent dans son travail et les opérations arithmétiques s'effectuent correctement en commençant par les octets de poids faible quel que soit l'ordre de stockage. Le problème se pose lorsque l'on essaie de transférer des données d'un ordinateur à un autre.

Imaginons que l'on écrive le code suivant en langage C pour sauvegarder une valeur numérique dans un fichier :

```
int i;  
fwrite(&i, sizeof(i), 1, f);
```

La fonction `fwrite()` sauve les 4 octets de `i` tels quels dans le fichier, dans l'ordre où ils sont stockés en mémoire. Pour relire le fichier, on utilise le code complémentaire suivant :

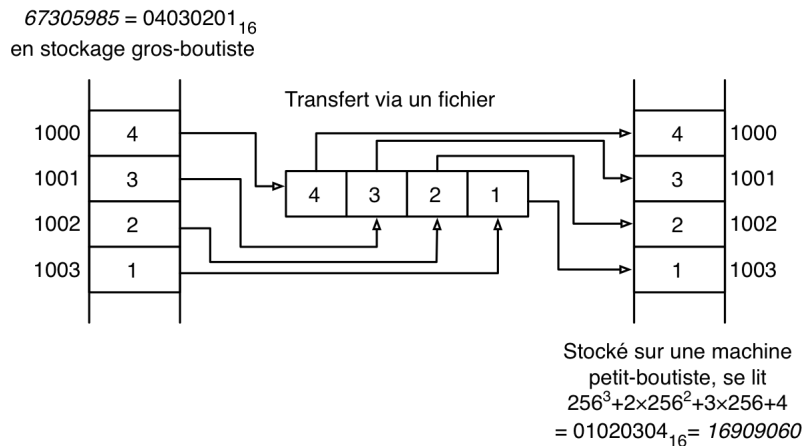
```
int i;  
fread(&i, sizeof(i), 1, f);
```

La fonction `fread()` lit successivement les 4 octets du fichier et les stocke en mémoire dans cet ordre.

Tant que l'on reste sur le même ordinateur ou des ordinateurs de même type (tous petit-boutistes ou tous gros-boutistes), il n'y a aucun problème : la lecture de l'entier se fait de façon identique sur chaque machine.

En revanche, si l'on transfère le fichier entre deux ordinateurs de types différents, le premier sauvegarde les octets dans un certain ordre, conservé par le second lors de la lecture, alors qu'il utilisera le nombre en lisant dans l'ordre inverse (voir figure 5.9).

Figure 5.9



Transfert entre des machines ayant des représentations différentes.

Pour résoudre ce problème, il faut imposer une norme de représentation de chaque type standard en vue des transferts d'informations, ainsi que les règles suivantes :

- Chaque programme doit sauvegarder les données ou les stocker dans sa mémoire en prenant les octets dans le même ordre que celui du fichier si sa représentation est la même que celle de la machine.
- Chaque programme doit inverser l'ordre des octets entre leur lecture et leur stockage (dans les deux sens) si les représentations fichier et machine sont inversées.

On peut aussi restreindre le transfert à des informations textuelles, c'est-à-dire convertir les valeurs numériques en chaînes de caractères lors de la sauvegarde, et faire la conversion inverse lors de la lecture du fichier.

RÉSUMÉ

Il existe différents types de mémoires, se distinguant par leurs performances, leurs accès, leur fabrication. La mémoire électronique rapide (RAM statique) est utilisée dans le processeur et la mémoire cache, tandis que la RAM dynamique est le choix idéal pour la mémoire principale. La mémoire secondaire oblige à faire appel à un support optique ou magnétique, avec des performances bien moindres mais une capacité inégalée.

Les variables d'un programme sont stockées en mémoire principale, sur le tas ou la pile suivant leur caractère global ou local, et leur utilisation n'est bien sûr pas identique. En outre, il existe deux manières de stocker les entiers en mémoire suivant l'ordre des octets et il faut faire attention si l'on souhaite transférer une valeur d'un processeur à un autre.

Problèmes et exercices

Les trois premiers exercices portent sur les caractéristiques des mémoires (performances, bande passante), les suivants sur le stockage en mémoire des données lors de l'exécution d'un programme et le choix des variables par le programmeur. Il sera également question de la différence entre variable locale et allocation mémoire, et des problèmes de stockage des entiers sur plusieurs octets (dans quel ordre sont-ils stockés ?).

Exercice 1 : Calculer le temps de propagation

La distance géographique entre la mémoire et le processeur influence-t-elle le temps d'accès à la mémoire ? Faut-il tenir compte d'un paramètre supplémentaire pour estimer le temps nécessaire au processeur pour récupérer les données ?

Le temps d'accès à une mémoire ne dépend que des circuits internes de son boîtier : temps de propagation des signaux à l'intérieur de ce dernier, temps de commutation des transistors, etc. Il est donc totalement indépendant de la distance entre le processeur et la mémoire. Cela dit, la demande émane du processeur, elle doit arriver à la mémoire (avant que celle-ci ne puisse commencer à travailler) qui, ensuite, produit la ou les données à retourner au processeur. C'est lors de cette propagation aller-retour que la distance entre les deux composants intervient.

Normalement, on pourrait considérer le temps de propagation des informations comme négligeable mais il ne faut pas oublier que le courant électrique se propage à une vitesse finie, en l'occurrence à environ deux tiers de la vitesse de la lumière dans un conducteur, soit 200 000 km/s. Le courant met donc à peu près 1 nanoseconde pour parcourir 20 cm, distance typique entre deux composants à l'intérieur d'un ordinateur. On voit donc que ce temps de propagation est de moins en moins négligeable au fur et à mesure que le temps d'accès de la mémoire baisse et que la fréquence d'horloge du processeur augmente.

Exercice 2 : Transférer des données

Une mémoire RAM, reliée au processeur, a un temps de cycle de 20 ns pour le premier accès et de 10 ns pour les trois accès suivants qui sont accélérés. Chaque accès récupère 8 octets.

Quelle est la bande passante du transfert d'informations entre la mémoire et le processeur ?

On récupère 8 octets en 20 ns puis encore 3×8 octets les 30 ns suivantes. On transfère donc 32 octets toutes les 50 ns.

La bande passante est donc de :

$$\frac{32}{50 \cdot 10^{-9}} = 640 \text{ Mo/s.}$$

Exercice 3 : Calculer les performances d'un disque dur

Un disque dur possède les caractéristiques suivantes :

- temps d'accès de 10 ms ;
- vitesse de rotation de 6 000 tour/mn ;
- taux de transfert de 20 Mo/s.

- 1) On transfère vers la mémoire un fichier de 20 Mo dont les données sont réparties en mille groupes de 20 Ko. Ces groupes sont situés à des endroits différents sur le disque, nécessitant un accès pour chacun. Quel est le temps total de transfert et quelle est la bande passante résultante ?
- 2) Suite à une réorganisation des informations sur le disque, le fichier de 20 Mo est maintenant stocké en cent groupes de 200 Ko. Quel est le temps total de transfert et quelle est la bande passante résultante ?

1) La vitesse de rotation est de 6 000 tour/mn, soit 100 tour/s. Pour que les informations arrivent sous la tête de lecture lorsqu'elle est positionnée au-dessus de la piste voulue, le disque doit pivoter en moyenne d'un demi-tour, soit une attente de 5 millisecondes. Une fois la tête positionnée au-dessus de la donnée, on transfère les informations avec un débit de 20 Mo/s. Il faut donc 1 milliseconde pour transférer un groupe de 20 Ko.

Au total, pour chaque groupe, il faut 10 ms d'accès, 5 ms de latence et 1 ms de transfert, soit 16 ms pour transférer 20 Ko. Il y a mille groupes à envoyer, ce qui fait un temps total de 16 secondes. La bande passante correspond à 20 Mo/16 secondes, soit 1,25 Mo/s.

2) Pour chaque groupe, il faut 10 ms d'accès, 5 ms de latence et 10 ms de transfert, soit 25 ms pour transférer 200 Ko. Le temps total est de 2,5 secondes pour les cent groupes. La bande passante est maintenant de 20 Mo/2,5 secondes, soit 8 Mo/s.

On voit bien l'importance de la répartition des données sur un disque. Le plus pénalisant lors d'un accès à une zone de stockage de la mémoire secondaire est les temps d'accès au support et de latence. Il est donc primordial d'optimiser ces accès. On peut le faire de deux manières :

- **En utilisant des périphériques performants.** Le temps d'accès et la vitesse de rotation d'un disque dur sont les critères principaux dont il faut tenir compte pour accélérer les opérations d'entrées/sorties.
- **En regroupant sur le disque les informations appartenant à un même fichier.** Cela permet de maximiser la quantité de données transférées à chaque déplacement de la tête de lecture. C'est le principe même d'une défragmentation (opération à effectuer régulièrement !).

Exercice 4 : Renvoyer une adresse

On considère le programme suivant :

Listing 5.1

```
char *Lecture(void) {
    char tableau[256];
    scanf("%s", tableau);
    return tableau;
}

main() {
    char *ptr;
    ptr = Lecture();
    printf("%s", ptr);
}
```

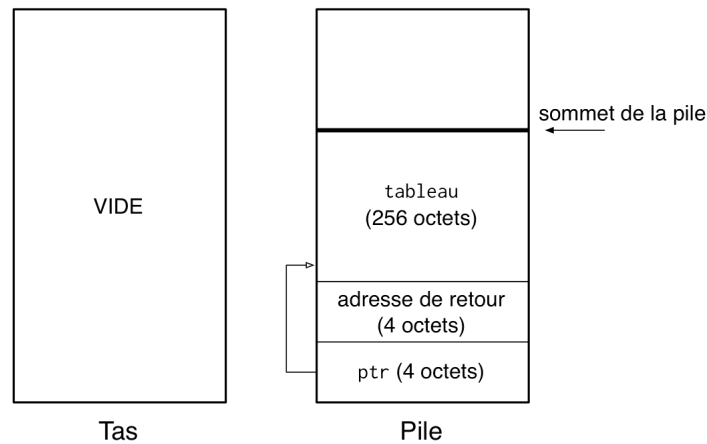
Lecture d'une chaîne (version incorrecte)

- 1) Dessinez l'état de la pile et du tas :
 - après l'appel de `scanf()` ;
 - avant l'appel de `printf()`.
- 2) Ce code vous semble-t-il correct ?
- 3) Que faut-il changer pour être sûr de sa fiabilité ?

1) Il n'y a pas de variables globales, ni d'allocation mémoire suite à un appel de la fonction `malloc()`. Le tas reste donc vide tout au long du déroulement du programme.

Lorsque l'exécution débute par la fonction `main()`, une variable locale `ptr` (un pointeur sur 4 octets) est créée sur la pile. L'appel de `Lecture()` empile l'adresse de retour (il n'y a pas de paramètre) puis, dans cette fonction, on réserve 256 octets, toujours sur la pile (c'est une variable locale), pour stocker le tableau de caractères récupéré par `scanf()` (voir figure 5.10).

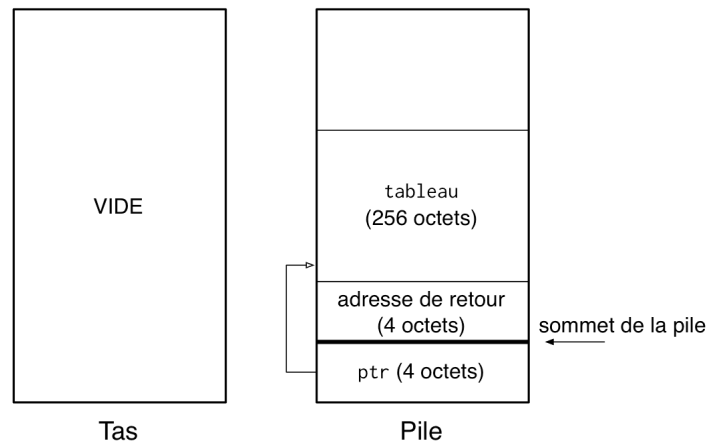
Figure 5.10



Pile et tas.

La fonction `lecture()` renvoie l'adresse de tableau, qui est mise dans `ptr`. Par ailleurs, à la fin de la fonction, la variable locale `tableau` et l'adresse de retour sont dépilées (voir figure 5.11).

Figure 5.11



Pile et tas (après retour).

2) Dépiler une valeur consiste à en récupérer une copie et à déplacer le sommet de la pile ; la valeur elle-même existe toujours en mémoire. Dans cet exemple, les caractères mis dans le tableau sont toujours stockés, et leur adresse stockée dans `ptr` est toujours exacte, **mais** ils sont dans une zone mémoire susceptible à tout moment d'être écrasée par de nouvelles valeurs empilées.

Le programme peut fonctionner si l'on utilise immédiatement la valeur renvoyée et si rien n'est empilé. Mais, selon toute probabilité, `ptr` va pointer rapidement sur des octets n'ayant plus rien à voir avec les caractères entrés car il s'agit d'une zone susceptible d'être utilisée : il suffit d'un appel de fonction (par exemple `printf()`) pour que de nouvelles informations (paramètres, adresse de retour, variables locales) soient empilées et viennent effacer les caractères de tableau.

Il ne faut jamais renvoyer l'adresse d'une variable locale hors de la fonction correspondante, car cette variable n'a plus d'existence légale. Malheureusement, c'est un bogue difficile à détecter car, les données étant toujours en mémoire, il est possible d'observer un fonctionnement correct, jusqu'au jour où l'on ajoute un appel de fonction qui écrase ces données en empilant ses propres variables. Heureusement, certains compilateurs affichent un avertissement si l'on renvoie l'adresse d'une variable locale.

3) On peut toujours renvoyer une valeur hors d'une fonction. Cependant, si cette valeur est un pointeur (comme `tableau` dans le listing précédent), il faut être certain que l'adresse pointée a été allouée sur le tas, et non sur la pile. Pour corriger le listing précédent, il suffit donc d'avoir un pointeur comme variable locale (on renverra sa valeur) et de l'initialiser avec l'adresse d'une zone mémoire allouée sur le tas.

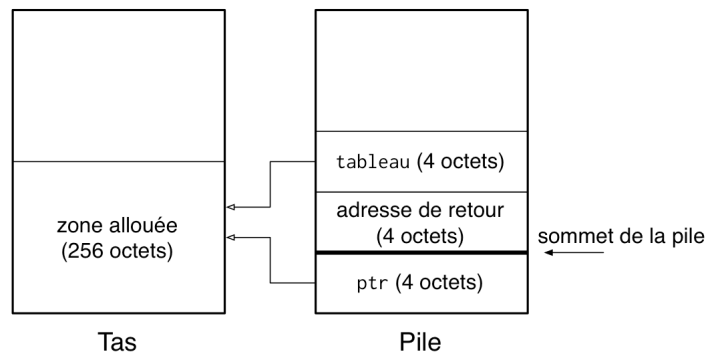
Listing 5.2

```
char *Lecture(void) {  
    char *tableau = malloc(256);  
    scanf("%s", tableau);  
    return tableau;  
}  
  
main() {  
    char *ptr;  
    ptr = Lecture();  
    printf("%s", ptr);  
}
```

Lecture d'une chaîne (version correcte)

Dans le listing 5.2, tableau devient un pointeur sur une zone mémoire de 256 octets allouée sur le tas. À la fin de l'exécution de cette fonction, tableau « disparaît » (puisque c'est une variable locale sur la pile), mais sa valeur (donc l'adresse de la zone mémoire) est toujours valable : elle est renvoyée par la fonction et mise dans ptr (voir figure 5.12). On peut donc bien se servir de la zone mémoire pour stocker des caractères. Si l'on voulait écrire un code parfaitement propre, on devrait libérer la zone après utilisation avec un appel de free(), mais ce n'est pas ici nécessaire car, à la fin du programme, toutes les zones allouées sont automatiquement libérées. C'est en revanche une étape indispensable si le programme est en permanence en cours d'exécution (un serveur tournant 24 heures sur 24 par exemple).

Figure 5.12



Pile et tas (version correcte).

Exercice 5 : Comprendre le ramasse-miettes

On considère le programme Java suivant :

Listing 5.3

```
public class Truc {
    public static Truc Creation() {
        Truc newTruc = new Truc();
        Copie(newTruc);
        return newTruc;
    }

    public static void Copie(Truc copieTruc) {
        Truc localTruc = copieTruc;
    }

    public static void main(String [] args) {
        Truc truc_1;
        Truc truc_2 = new Truc();
        Truc truc_3 = new Truc();
        Truc truc_4 = truc_3;
        truc_1 = Creation();
        // Insérer ici une ligne
    }
}
```

Un exemple

1) Dessinez l'état de la pile et du tas après l'appel à la méthode création(). Y a-t-il des candidats au ramasse-miettes ?

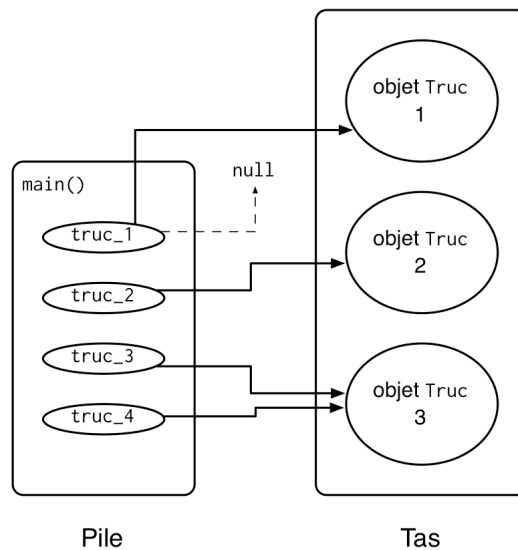
2) Indiquez s'il est licite de remplacer la ligne de commentaire de la méthode main() par une des lignes suivantes. Dans quels cas (si le remplacement est licite) y a-t-il un objet candidat au ramasse-miettes avant que la méthode ne se termine ?

- a) copieTruc = null;
- b) newTruc = null;
- c) newTruc = truc_3;
- d) truc_1 = null;
- e) truc_2 = null;
- f) truc_3 = null;
- g) truc_4 = null;
- h) truc_1 = truc_4;
- i) truc_3 = truc_2;

1) La méthode Creation() construit un nouvel objet de la classe Truc et renvoie sa référence. La méthode copie() ne fait rien d'intéressant : elle recopie la référence passée en paramètre dans une variable locale, qui va disparaître à la fin de l'exécution de la méthode.

Le main() crée quatre références à des objets Truc : la première vaut d'abord null puis vaut la référence renvoyée par Creation() ; la deuxième et la troisième pointent sur des objets créés sur le tas ; la quatrième référence le même objet que la troisième. On a donc le schéma de la figure 5.13.

Figure 5.13



Pile et tas (Java).

Tous les objets sont référencés ; il n'y a donc pas lieu de faire intervenir le ramasse-miettes.

2) On ne peut pas remplacer le commentaire par l'une des trois premières lignes car celles-ci font référence à une variable locale d'une méthode déjà exécutée (copieTruc ou newTruc).

`truc_1 = null;` est licite et supprime la référence au premier objet. Celui-ci n'étant plus référencé, il devient candidat au ramasse-miettes.

Même raisonnement pour `truc_2 = null;`, qui supprime l'unique référence au deuxième objet.

Les deux lignes `truc_3 = null;` et `truc_4 = null;` changent la valeur de la référence concernée, mais l'objet référencé ne peut pas être détruit par le ramasse-miettes car il fait l'objet au départ de deux références et qu'il en reste encore une. Il n'a donc aucune raison de disparaître.

`truc_1 = truc_4;` fait pointer la première variable sur le troisième objet. On perd ainsi la seule référence au premier objet, qui va être nettoyée par le ramasse-miettes. `truc_3 = truc_2;` fait de même entre la troisième variable et le deuxième objet, mais sans autre dégât car le troisième objet est toujours référencé.

Exercice 6 : Choisir l'ordre des octets

Dans le but de transférer un nombre entier sur 4 octets entre deux ordinateurs *via* un fichier, on décide d'une représentation standard : on envoie les octets dans l'ordre, de l'octet de poids faible à celui de poids fort.

Comment les générer indépendamment et les envoyer dans un fichier ? Comment faire l'opération inverse ?

Si l'on ne connaît pas le type de représentation utilisée en interne (petit-boutiste ou gros-boutiste), il faut simplement calculer chaque octet par des divisions et des modulus :

```
int val;
char octets[4];
...
octets[0] = val %256;
octets[1] = (val/256) %256;
octets[2] = (val/65536) %256;
octets[3] = (val/0x01000000) %256;
```

On peut améliorer le code en passant par des décalages et des ET logiques, qui s'exécutent beaucoup plus rapidement :

```
int val;
char octets[4];
...
octets[0] = val & 0xff; /* on garde les 8 bits de poids faible */
octets[1] = (val>>8) & 0xff; /* >> décale de 8 bits à droite */
octets[2] = (val>>16) & 0xff;
```

```
octets[3] = (val>>24) & 0xff;
```

Il est ensuite facile de sauvegarder ce tableau dans un fichier :

```
fwrite(octets, 4, 1, f);
```

L'opération inverse consiste d'abord à lire les 4 octets dans un tableau :

```
char octets[4];  
fread(octets, 4, 1, f);
```

On reconstruit ensuite le nombre par des additions et des multiplications :

```
int val;  
val = octets[3]*256*256*256+octets[2]*65536+octet[1]*256+octets[0];
```

Là encore, on peut utiliser des décalages et des OU logiques :

```
int val;  
val = (octets[3]<<24) | (octets[2]<<16) | (octet[1]<<8) | octets[0];
```

Exercice 7 : Incrémenter une variable

On écrit le code source suivant :

Listing 5.4

```
void Inc(int *var) {  
    (*var)++;  
}  
  
main() {  
    char c = 0;  
    Inc(&c);  
    printf("%d",c);  
}
```

Incrémentation d'une variable

- 1) Ce programme se compile-t-il correctement ? Que faut-il ajouter pour que le compilateur n'affiche pas d'avertissement ?
- 2) Quelle est la valeur affichée à la fin de l'exécution ?

1) La fonction Inc() prend un pointeur sur un entier et incrémente la variable pointée. Dans le main(), on déclare un caractère et on lui affecte la valeur numérique 0 (et non le caractère '0' de code numérique ASCII 48). On l'incrémente avant d'afficher sa valeur numérique sous forme lisible (et non comme un caractère, ce qui nécessiterait d'utiliser %c).

Le seul problème éventuel provient de l'appel de la fonction Inc(). Cette dernière attend comme paramètre un pointeur sur un entier alors que &c est du type pointeur sur un caractère. Suivant les options de compilation, le compilateur peut avoir les réactions suivantes :

- être laxiste et ne rien dire ;
- afficher un avertissement (« pointeurs de types incompatibles ») et continuer son travail en générant un fichier exécutable (comportement le plus courant) ;
- refuser de compiler en considérant cette erreur comme rédhibitoire (c'est le cas si l'on a choisi des options strictes de compilation).

Dans les deux derniers cas, il suffit de forcer le compilateur à accepter le changement de type en effectuant un transtypage (cast) sur la variable :

```
Inc( (int *) &c);
```

On transforme ainsi le pointeur sur un caractère en un pointeur sur un entier si le compilateur ne le fait pas automatiquement (quitte à afficher l'avertissement) et refuse de compiler.

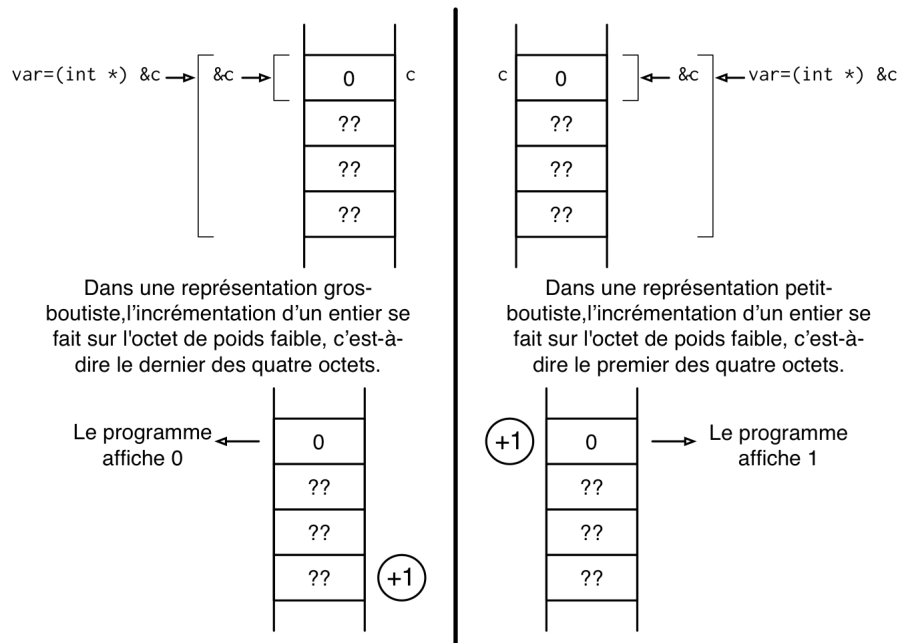
2) Il n'y a pas une bonne réponse mais deux : le programme va afficher 0 ou 1 suivant le type de processeur de l'ordinateur.

Cette différence provient du fait qu'un pointeur sur un caractère désigne bien l'adresse d'un octet en mémoire alors qu'un pointeur sur un entier (qui a la même valeur numérique que le pointeur sur le caractère) désigne un ensemble de 4 octets dont l'adresse peut correspondre à l'octet de poids faible (représentation petit-boutiste des entiers) ou à l'octet de poids fort (représentation gros-boutiste).

Dans une représentation petit-boutiste, le pointeur var possède l'adresse du premier octet de l'entier, qui correspond aussi à l'octet de poids faible. C'est donc bien celui-là qui est incrémenté par la fonction. Dans une représentation gros-boutiste, le pointeur var désigne la même adresse que c, mais c'est celle de l'octet de poids fort de l'entier. Comme le processeur sait qu'il doit incrémenter un entier sur 4 octets (cela lui a été indiqué par le compilateur dans le code exécutable), il additionne 1 à l'octet de poids faible, se trouvant 3 octets plus loin en mémoire. Le premier octet, qui stocke c, n'est *a priori* pas affecté par cette opération (sauf si les 3 octets de poids faible ont la valeur 255 et qu'une retenue se propage ainsi jusqu'à l'octet de poids fort, celui de c).

Lorsque l'on exécute la fonction printf(), le programme récupère la valeur de c, donc dans tous les cas, la valeur contenue dans le premier octet car c désigne bien le caractère stocké à cette adresse. Un processeur petit-boutiste affiche donc 1, et un processeur gros-boutiste 0 (voir figure 5.14).

Figure 5.14



Incrémentation d'un entier.

En plus de cette incohérence (un programme correct devrait s'exécuter de façon identique sur tous les processeurs), il y a un risque de corruption de données : le processeur gros-boutiste incrémente un octet qui est situé plus loin en mémoire que l'adresse de c et qui correspond peut-être au stockage d'une autre variable dont la valeur sera ainsi involontairement modifiée.

Il faut donc faire attention aux changements de type de pointeur en langage C car cela modifie la façon dont le processeur lit les données en mémoire.

Chapitre 6

Mémoire cache

La mémoire principale n'arrive plus à suivre le rythme effréné des fréquences d'horloge des processeurs du marché. Heureusement, entre les deux, il est possible d'intercaler une mémoire cache (appelée aussi « cache »), plus rapide et de faible taille, pour limiter l'impact de cette lenteur par rapport aux temps de cycle des processeurs. Cette mise en place constitue un premier niveau de hiérarchisation de la mémoire. L'objectif est de mémoriser les informations les plus utilisées par le processeur afin d'être en mesure de les lui retransmettre rapidement en cas de besoin. Dans ce cas, il n'est pas nécessaire de solliciter de nouveau la mémoire principale, d'où un gain de temps important. L'efficacité d'un tel système dépend des caractéristiques de la mémoire cache, qui conditionnent également sa complexité. Pour limiter cette complexité, il est possible de hiérarchiser davantage la mémoire en plaçant plusieurs niveaux de cache entre la mémoire principale et le processeur.

Mémoire cache

- 1. Principe 136
- 2. Caractéristiques 138
- 3. Amélioration des caches 145

Problèmes et exercices

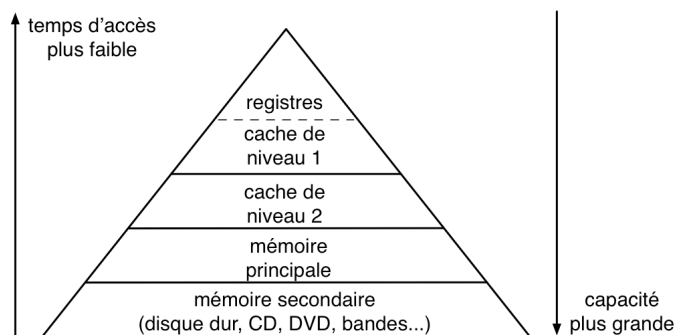
- 1. Calculer des temps d'exécution 149
- 2. Comparer les tailles des lignes 150
- 3. Comparer les adresses mémoire 153
- 4. Optimiser un programme 155
- 5. Réduire la taille des boucles 156
- 6. Un exemple plus important 158
- 7. Influence de la réécriture 159

I. PRINCIPE

De nombreux types de mémoires sont à la disposition des concepteurs d'ordinateurs, mais aucun ne satisfait tous les besoins : elle doit être rapide pour alimenter le processeur en instructions et en données, et de grande capacité pour stocker l'intégralité des informations disponibles. Or, souvent, plus la mémoire offre de cases disponibles, plus son accès est lent.

Il faut donc utiliser tous les types de mémoires en hiérarchisant et en répartissant les informations (instructions et données) en fonction de leur fréquence d'utilisation. Plus celle-ci est importante, plus il faut mettre l'information dans une mémoire rapide près du processeur. Les autres informations se verront reléguées sur des supports plus grands, mais plus lents. On représente cette utilisation par une pyramide couvrant, peu ou prou, tous les besoins de stockage (voir figure 6.1).

Figure 6.1



Hiérarchie mémoire.

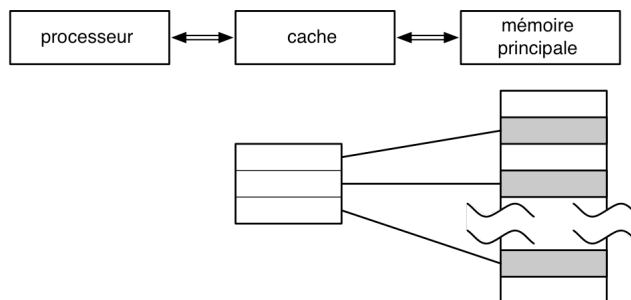
Depuis 30 ans, les performances des processeurs ont connu une accélération phénoménale : augmentation de la fréquence d'horloge et introduction de techniques avancées (pipeline, superscalaire...). Malheureusement, le temps d'accès à la mémoire principale n'a pas diminué dans les mêmes proportions. Alors que le processeur avait, à l'époque, une vitesse de fonctionnement proche de celle des mémoires, de nos jours il doit attendre l'extraction de données et d'instructions en provenance de la mémoire principale. Tandis que lui essaie d'exécuter une instruction (voire plusieurs) par cycle d'horloge, c'est-à-dire à peu près toutes les nanosecondes, elle, de son côté, a un temps d'accès de quelques dizaines de nanosecondes.

Pour résoudre ce handicap, les constructeurs ont intercalé une mémoire rapide, dite « cache » (ou, parfois, appelée « antémémoire ») entre le processeur et la mémoire principale. Elle fournit à celui-ci les données et instructions nécessaires à l'exécution des programmes, en opérant plus rapidement que la mémoire principale car elle utilise la technologie de mémoire statique. En contrepartie, sa taille est limitée. Pour cette raison, elle ne peut constituer l'intégralité de la mémoire principale, ce qui résoudrait le problème de la différence de vitesse.

I.1 Fonctionnement

Le cache contient en permanence un sous-ensemble (une copie) des informations stockées en mémoire principale (voir figure 6.2).

Figure 6.2



Principe de la mémoire cache.

Lorsque le processeur effectue une lecture mémoire, il cherche d'abord si l'information (donnée ou instruction) est dans le cache. Si c'est le cas, il la récupère rapidement sans attendre. Sinon (le cache ne contient qu'une partie des informations), il va la prendre en mémoire principale.

Algorithme de lecture mémoire

À chaque accès mémoire effectuant une lecture, l'algorithme suivant est exécuté :

- Le processeur envoie l'adresse souhaitée au cache et à la mémoire principale.
- Si le cache contient l'information correspondante (c'est-à-dire si l'adresse demandée fait partie du sous-ensemble qu'il mémorise), il l'envoie au processeur, et la requête vers la mémoire principale est abandonnée.
- Si le cache n'a pas l'information, on dit qu'il y a un défaut de cache. On poursuit alors l'accès à la mémoire principale, qui renvoie au processeur l'information stockée dans la case *ad hoc*. Au passage, la ligne contenant l'adresse (un ensemble de quelques cases voisines) est mémorisée dans le cache pour de possibles accès futurs.

Ce mécanisme est entièrement géré par le matériel et totalement transparent du point de vue du programmeur, qui n'a aucun moyen d'action sur le cache.

Comment augmenter la probabilité que le processeur trouve dans le cache l'information voulue ? En plaçant dans cette zone ce que le processeur va utiliser dans un avenir proche ! Malheureusement, on ne peut le savoir à l'avance. On ne peut faire que des hypothèses et espérer qu'elles soient vérifiées. Par chance, le comportement d'un processeur n'est pas complètement aléatoire et peut, dans une certaine mesure, être prédit. C'est au travers de l'exploitation de deux « principes » de localité des accès mémoire que la mise en place des caches va apporter un gain en performance important.

1.2 Localité

Les programmes standard ont souvent des comportements prédictibles soit temporellement, soit spatialement, de sorte que les accès mémoire réalisés par le processeur pour les exécuter ne se font pas au hasard.

Localité temporelle (ou principe de réutilisation)

Le processeur accède à une case mémoire contenant une donnée parce qu'une variable y est stockée. Selon toute probabilité, il y accèdera de nouveau dans un avenir proche. En effet, la variable est peut-être utilisée dans le cadre d'un calcul ou comme indice de boucle. Dans ces deux cas, le code va plusieurs fois d'affilée y faire référence.

En outre, le processeur exécute de temps à autre une même instruction plusieurs fois dans un court intervalle de temps, par exemple lors de l'exécution d'une boucle.

Définition

Le principe de localité temporelle dit simplement que le processeur fait plusieurs fois référence à une même case mémoire à des instants rapprochés.

Ainsi, lorsque le processeur accède à une adresse mémoire et récupère l'information (donnée ou instruction) en mémoire principale, il est intéressant de la stocker dans le cache pour que son prochain accès soit plus rapide, car le processeur va probablement en avoir de nouveau besoin bientôt.

Ce principe n'est pas vrai à 100 % (il arrive qu'on n'utilise qu'une seule fois une variable ou une instruction) mais il est assez souvent vérifié.

Localité spatiale

Après avoir exécuté une instruction, le processeur va chercher celle qui lui succède, qui normalement se situe en mémoire à l'adresse suivante, puis continue de la même manière avec les instructions les unes après les autres. Les seules exceptions à cette utilisation linéaire de la mémoire correspondent aux ruptures de séquence : sauts inconditionnels, sauts conditionnels effectués, appels de fonction. Dans ces cas-là, il n'y a aucun rapprochement entre les deux adresses mémoire auxquelles le processeur accède successivement, celles du saut et de l'adresse cible.

Les accès aux données se font aussi parfois à des adresses successives en mémoire : les éléments d'un tableau sont stockés les uns à la suite des autres et les algorithmes demandent couramment de le parcourir élément par élément.

Définition

Selon le principe de localité spatiale, si le processeur accède à une case mémoire à un instant donné, il accédera probablement à des cases mémoire voisines aux instants suivants.

Partant de là, on propose à chaque accès mémoire de ramener dans le cache, non seulement l'information voulue qui ne s'y trouve pas (principe de réutilisation), mais également un ensemble (appelé « ligne ») de plusieurs cases mémoire voisines de celle à laquelle le processeur a accédé, en espérant que le prochain accès vérifiera le principe de localité spatiale et se fera dans les références rapatriées dans le cache.

Là encore, ce principe n'est pas toujours vérifié (ruptures de séquence, accès à des variables quelconques), mais suffisamment respecté pour être intéressant.

2. CARACTÉRISTIQUES

Si le principe de la mémoire cache est assez facile à comprendre, son implémentation demande de prendre en considération un certain nombre de caractéristiques : quelle taille doit avoir le cache et chaque ligne, comment se fait le rangement et le remplacement des lignes à l'intérieur du cache, que se passe-t-il en cas d'écriture ?

2.1 Tailles d'un cache

Dimension globale

Il semble évident que plus la capacité du cache est importante, plus le processeur a de chance d'y trouver l'information qu'il souhaite. On devrait donc toujours chercher à avoir le cache le plus important possible. Il faut cependant moduler cette affirmation par quelques considérations :

- La mémoire cache a un coût non négligeable en raison de sa technologie.
- Implantée sur une puce, elle occupe de l'espace, d'autant plus grand qu'elle est plus rapide, et ce peut-être au détriment d'autres composants du processeur dont elle prend la place. Il faut donc faire un compromis entre la capacité du cache et les unités formant le processeur.
- En raison de la complexité des circuits utilisés, le temps d'accès d'un cache est souvent proportionnel à sa capacité, ce qui rentre en conflit avec la rapidité que l'on recherche lors de la mise en place d'un cache.
- Le gain obtenu grâce à la présence de la mémoire cache est dégressif. À partir d'une certaine capacité, en ajouter ne sert plus à grand-chose : il y aura toujours des informations qui feront défaut car elles n'auront jamais été chargées avant leur usage.

La dimension globale du cache est donc un paramètre parmi d'autres lors de la conception des circuits électroniques formant le processeur sur la puce.

Taille d'une ligne

La ligne est l'ensemble des informations ramenées en cache lors de l'accès, en mémoire principale, à une adresse. Quelle doit être sa taille ? Autrement dit, combien d'octets doit-on ramener en cache à chaque accès en mémoire principale ?

On peut penser qu'il faut prendre une ligne la plus grande possible. En effet, plus on ramène d'informations dans le cache, plus on peut profiter du principe de localité spatiale : toutes les instructions rapatriées dans la ligne seront exécutées rapidement à partir du cache. Ce raisonnement est d'autant plus exact que le programme s'exécute de manière parfaitement linéaire sans rupture de séquence. Cependant, à cause de telles ruptures, des parties différentes du code s'exécutent à la suite, et de temps à autre plusieurs fois d'affilée. Il est alors intéressant de stocker dans le cache davantage de lignes différentes pour avoir plus de chances que la prochaine instruction y soit. La contrepartie est que cela oblige à avoir plus d'emplacements dans le cache et donc de diminuer la taille d'une ligne. Le même phénomène se reproduit avec les données, qui respectent moins bien le principe de localité spatiale. Il n'y a donc pas de valeur optimale pour la taille d'une ligne. L'optimisation sera plus ou moins bonne suivant chaque programme. Un autre paramètre intervient lorsque l'on cherche à déterminer cette taille optimale : il s'agit de la taille du lien entre la mémoire principale et le cache. Ces deux composants sont reliés par un bus mémoire capable de transférer plusieurs octets, souvent 8 ou 16, en une opération. Cette unité de transfert est un candidat idéal pour la taille d'une ligne : le processeur ne gagne pas de temps à envoyer moins d'octets alors que cela en prend plus d'avoir une ligne plus grande (car il est obligé d'effectuer plus d'un transfert sur le bus).

Note

Les lignes ne sont pas disposées n'importe où en mémoire. Leur adresse (celle de leur premier octet) est toujours un multiple de leur taille. On dit que les lignes sont alignées en mémoire. La ligne 0 commence à l'adresse 0 et la ligne N se trouve à l'adresse TN si T est la taille de la ligne. Cela permet de retrouver facilement, à partir de l'adresse d'un octet, la ligne correspondante et son adresse.

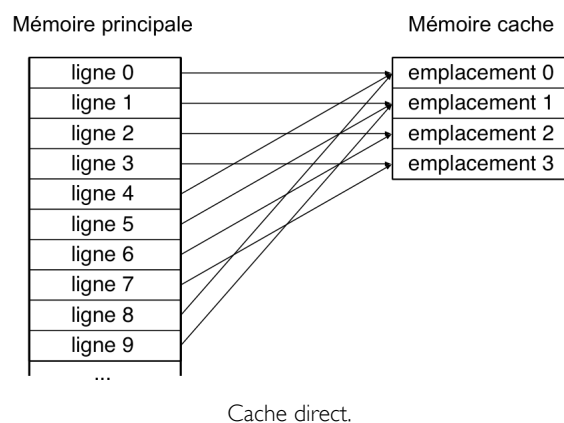
2.2 Adressage

Lorsque la mémoire transfère une ligne dans le cache, il faut décider où celle-ci doit-elle se placer.

Cache à correspondance direct (ou cache direct)

L'adressage le plus simple est celui des caches directs : chaque ligne mémoire y a un emplacement réservé. À partir du numéro de ligne mémoire, on obtient le numéro de l'emplacement dans le cache où elle va se ranger en prenant le numéro de la ligne modulo le nombre de lignes dans le cache (voir figure 6.3).

Figure 6.3



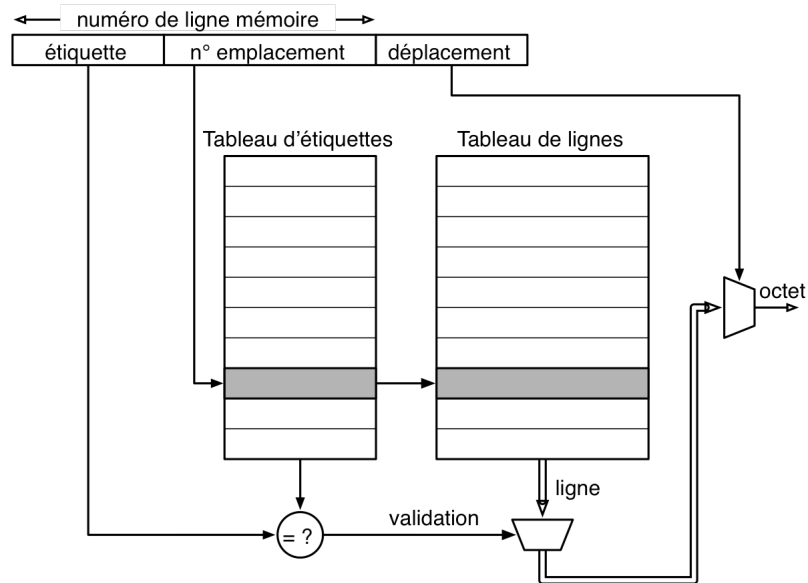
Cette technique est simple à mettre en œuvre car il est facile de calculer la position de la ligne dans le cache lorsqu'elle arrive de la mémoire. Par ailleurs, lorsque l'on cherche si une ligne mémoire est déjà présente dans le cache, il n'y a qu'un emplacement à vérifier, celui où elle peut se ranger.

L'organisation de la mémoire cache est alors la suivante : à la façon d'une mémoire associative, chaque ligne mémorisée est associée à une partie de l'adresse mémoire correspondante. Cette association est réalisée au travers d'une étiquette, dont le rôle consiste à spécifier quelle ligne de la mémoire centrale est stockée dans la ligne de cache correspondante. En effet, comme plusieurs lignes mémoire peuvent occuper le même emplacement dans le cache, il faut savoir laquelle est effectivement présente. En comparant les étiquettes de l'adresse demandée et de celle stockée dans le cache, il est possible de déterminer si l'information présente dans le cache est celle qui est recherchée.

Une adresse mémoire est découpée en trois parties (voir figure 6.4). Les bits de poids faible correspondent au déplacement dans la ligne qu'il faut effectuer pour récupérer l'octet demandé. Sur une ligne de 2^d octets, d bits indiquent le déplacement. Le reste des bits donne le numéro de ligne mémoire, lui-même formé de deux champs : d'abord, le numéro de l'emplacement dans le cache où va la ligne mémoire ; là encore, s'il y a 2^n emplacements, ce champ fait n bits ; ensuite, les bits de poids fort restants forment l'étiquette mémorisée avec la ligne dans le cache.

Lors d'une lecture, le numéro d'emplacement est extrait de l'adresse mémoire demandée par le processeur lors de sa requête mémoire (concernant une donnée ou une instruction). Il permet d'extraire l'étiquette de la ligne stockée dans le cache. En comparant cette étiquette et celle de la requête du processeur, il est possible de déterminer si l'information recherchée est dans le cache ou pas. S'il n'y a pas égalité, une autre ligne mémoire est dans le cache et il y a alors un défaut de cache sur l'adresse demandée (on parle aussi de *cache miss*). Si les deux étiquettes sont égales, la ligne stockée dans le cache est bien celle qui est recherchée et il est possible de l'extraire (on parle alors de *cache hit*). Le champ de déplacement permet de récupérer un octet précis de cette ligne (voir figure 6.4).

Figure 6.4



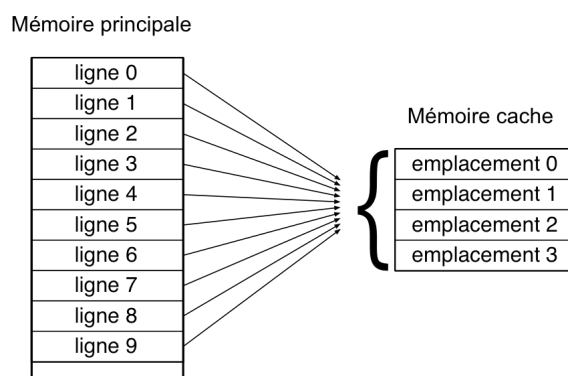
Lecture dans un cache direct.

Le mécanisme de cache direct est facile à implémenter : chaque ligne ne pouvant occuper qu'un emplacement, il n'y a qu'un comparateur à prévoir. Il peut être construit avec des circuits électroniques rapides sans occuper trop de place sur la puce. Malheureusement, cet avantage se paie par une diminution de l'efficacité. Chaque ligne mémoire ayant un emplacement attribué, deux références mémoire successives peuvent se faire sur deux lignes mémoire devant occuper le même emplacement dans le cache. Le deuxième accès va évincer la première ligne pour mettre celle qui est référencée à sa place. Toute référence à la première ligne provoquera donc un défaut de cache, même si d'autres emplacements restent libres dans le cache. On risque donc une sous-utilisation de cette mémoire, certains emplacements n'étant pas occupés alors qu'une concurrence s'exerce pour l'usage de quelques autres.

Cache complètement associatif

Pour permettre un meilleur usage des différents emplacements du cache, il est nécessaire de se libérer de la contrainte sur la localisation unique d'une ligne mémoire dans le cache. Dans un cache associatif, une ligne mémoire peut occuper n'importe quel emplacement (voir figure 6.5).

Figure 6.5



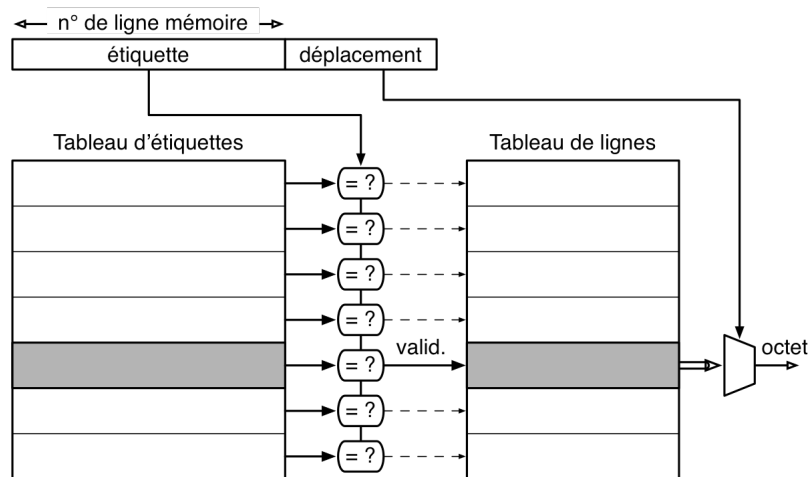
Cache associatif.

De cette façon, on ne gaspille aucun emplacement du cache puisque dès qu'il y en a un de libre, il peut être utilisé par une ligne ramenée depuis la mémoire principale. Un conflit d'accès à une ligne survient uniquement lorsque le cache est entièrement rempli. L'efficacité est donc maximale, et la complexité des circuits également.

Lorsque l'on doit chercher une adresse mémoire dans le cache, il faut examiner tous les emplacements et comparer, pour chacun, l'étiquette stockée et celle de l'adresse mémoire. Celle-ci est formée de deux champs : les bits de poids faible donnent le déplacement dans la ligne, et le reste de l'adresse qui compose l'étiquette.

La comparaison des deux étiquettes doit se faire pour tous les emplacements, si possible en parallèle pour ne pas perdre de temps. Il faut donc prévoir autant de circuits comparateurs que d'emplacements dans le cache et les circuits nécessaires à la sélection de la donnée une fois l'étiquette trouvée (voir figure 6.6). En ce sens, les caches associatifs sont plus complexes et prennent plus de place que les caches directs, et sont aussi plus efficaces en générant moins de défauts de cache. En outre, les étiquettes étant plus longues (il n'y a plus de numéro d'emplacement), le tableau qui les stocke est plus volumineux et occupe lui aussi plus de place sur la puce.

Figure 6.6

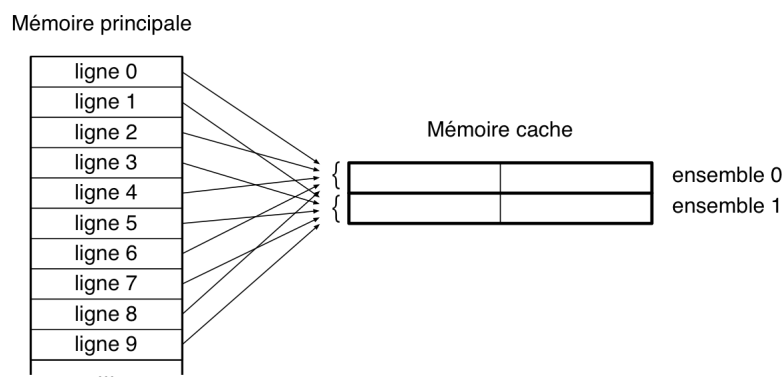


Lecture dans un cache associatif.

Cache mixte ou associatif par ensemble

On peut trouver un bon compromis entre les deux techniques précédentes avec le cache associatif par ensemble. Au lieu de restreindre une ligne mémoire à un emplacement du cache (direct) ou de lui permettre d'occuper toutes les places possibles (complètement associatif), on se donne une liberté limitée en autorisant une ligne mémoire à se mettre à n'importe quel emplacement parmi un ensemble précis de possibilités (voir figure 6.7). On parle de cache par ensemble de n , s'il y a n emplacements possibles dans un ensemble (n est aussi le degré d'associativité du cache). La figure 6.7 présente un cache associatif par ensemble de 2 car il y a deux emplacements dans chaque ensemble.

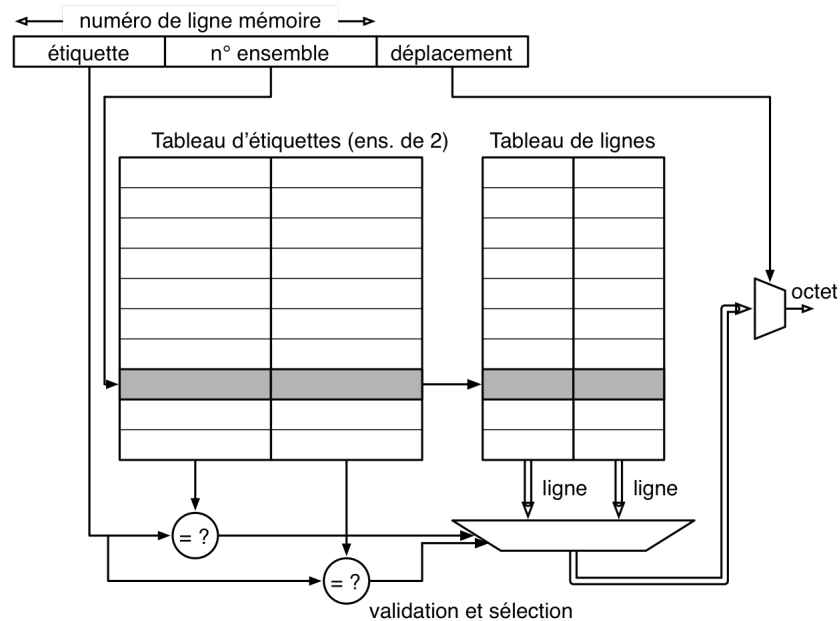
Figure 6.7



Cache associatif par ensemble de 2.

Chaque ligne mémoire est associée à un ensemble via son numéro modulo le nombre d'ensembles. Comme dans le cache direct, une adresse mémoire est découpée en trois champs : déplacement, numéro d'ensemble et étiquette. Une lecture mémoire se fait d'abord comme dans un cache direct, en extrayant le numéro de l'ensemble de l'adresse, puis comme dans un cache associatif, en comparant l'étiquette avec toutes celles stockées dans l'ensemble concerné. Si on a une correspondance, la ligne (associée à l'étiquette) mémorisée dans le cache est récupérée (voir figure 6.8).

Figure 6.8



Lecture dans un cache associatif par ensemble de 2.

On a besoin d'autant de comparateurs que d'emplacements dans un ensemble. C'est donc là aussi un bon compromis en ce qui concerne les circuits associés aux tableaux d'étiquettes et de lignes.

Plus le cache est grand, moins il est intéressant qu'il soit associatif car il y a moins de conflits (deux lignes voulant occuper le même emplacement). C'est pourquoi les grands caches sont en règle générale directs. Par ailleurs, l'intérêt de l'associativité est dégressif : on gagne beaucoup (en réduisant fortement les conflits) à passer d'un cache direct à un cache associatif par ensemble de 2 ou 4, mais il n'est pas très intéressant d'augmenter plus fortement le degré d'associativité (et cela complique les circuits associés). Pour cette raison, les petits caches sont souvent associatifs par ensemble, avec un degré de 2, 4, 8 ou 16.

2.3 Algorithmes de remplacement

Lorsqu'une nouvelle ligne mémoire est référencée et donc amenée dans le cache, se pose la question de savoir à quel emplacement la mettre. Dans le cas d'un cache direct, la réponse est immédiate puisqu'il n'existe qu'une seule possibilité. En revanche, on doit choisir un emplacement dans un cache complètement associatif tandis que, dans un cache associatif par ensemble, on choisit parmi tous ceux de l'ensemble attribué à la ligne mémoire. Dans les deux cas, si un emplacement est libre (c'est-à-dire ne contient aucune information valable), on le choisit pour rapatrier la ligne mémoire. Mais que faut-il faire si le cache entier (associatif) ou l'ensemble (associatif par ensemble) est plein ?

En théorie, il faut évacuer du cache une ligne qui ne sera plus référencée par le processeur afin de faire de la place pour la nouvelle. En pratique, il est impossible au processeur, et *a fortiori* aux circuits liés au cache, de connaître à l'avance les futurs accès mémoire. Il faut donc avoir une estimation de l'usage futur des lignes du cache pour déterminer celle qui est la plus « inutile ». Pour ce faire, plusieurs algorithmes ont été proposés. Ils sont statistiquement plus ou moins efficaces. Cela dépend fortement de la structure du programme, de son usage des variables... Mais cela n'est pas le seul critère. Il faut en plus tenir compte de la rapidité de l'algorithme (même s'il prédit très bien quelle ligne éjecter, s'il est trop long à exécuter, on perd tout le bénéfice du cache) et de sa complexité (qui se traduit par des circuits supplémentaires et donc de la place en moins pour le cache, augmentant les conflits d'accès).

Remplacement aléatoire

La première possibilité d'algorithme part du principe qu'il est impossible de prédire l'avenir et qu'il ne sert à rien d'essayer d'estimer quelle ligne évincer. On peut donc se contenter d'en choisir une au hasard (dans tout le cache ou dans l'ensemble correspondant) pour la remplacer par la nouvelle ligne.

Cet algorithme est facile à implémenter mais n'est pas efficace : rien n'indique qu'on ne va pas réutiliser rapidement la ligne sortie aléatoirement du cache.

Remplacement FIFO

L'idée de base de l'algorithme FIFO (*First In First Out*, premier entré premier sorti) est que des informations mises depuis longtemps dans le cache ne vont plus servir. Chaque ligne qu'il contient est accompagnée de sa date d'entrée (ou simplement d'un compteur) et la plus vieille est remplacée par la ligne provenant de la mémoire principale.

Cet algorithme est statistiquement plus efficace que le précédent, mais il a un défaut : une ligne référencée régulièrement (c'est-à-dire une adresse mémoire à laquelle on a accédé souvent) est retirée du cache quand elle devient la plus vieille, alors qu'elle peut encore servir.

Remplacement LRU

L'important finalement n'est pas l'ancienneté de la ligne mais l'ancienneté de son utilisation. L'algorithme LRU (*Least Recently Used*, la moins récemment utilisée) associe à chaque ligne stockée dans le cache la date de sa dernière utilisation. On extrait du cache celle dont la date de dernière utilisation est la plus ancienne pour faire place à la nouvelle ligne. On part ainsi du principe, assez naturel, qu'une ligne non utilisée depuis longtemps ne le sera pas dans un avenir proche. C'est toujours un pari sur les futurs accès mémoire, mais assez raisonnable et statistiquement (sur des programmes standard) le plus efficace (il provoque le moins de défauts de cache).

Malheureusement, son efficacité se paie par une complexité accrue : il faut associer une date à chaque ligne, et, à chaque accès, la modifier et surtout trier les dates de toutes les étiquettes pour ne garder que la plus ancienne. Ces opérations prennent un certain temps et nuisent à la rapidité du cache.

Remplacement LFU

Une variante de l'algorithme précédent attache aux lignes un compteur s'incrémentant à chaque utilisation. Lors d'un remplacement, la ligne victime a le compteur le plus faible (*Least Frequently Used*, la moins utilisée). Le tri est plus simple sur les compteurs que sur les dates mais laisse dans le cache d'anciennes lignes auxquelles on a souvent accédé à une époque (donc avec un compteur élevé), même si le programme n'en a plus besoin.

Remplacement NMRU

L'algorithme LRU étant souvent trop complexe, une dernière variante consiste, non pas à éjecter la plus vieille ligne, mais à s'assurer que la plus récente reste en cache (car c'est elle qui a probablement le plus de chances d'être référencée de nouveau). Chaque ligne porte un bit supplémentaire, mis à 1 lors de l'accès à celle-ci, ceux des autres lignes étant remis à 0. Lors d'un remplacement, on choisit aléatoirement parmi toutes les lignes ayant ce bit à 0, la dernière ligne à laquelle on a accédé n'étant pas concernée. Cela donne des résultats acceptables pour des circuits supplémentaires assez simples.

Remarque

Parmi tous ces algorithmes, aucun n'est totalement et absolument supérieur à un autre. On peut toujours trouver une suite de références à des adresses mémoire qui privilégierait un algorithme par rapport à un autre (c'est-à-dire qui générerait moins de défauts de cache). Il faut les comparer d'un point de vue statistique sur des programmes assez standard pour avoir une estimation de leurs performances.

2.4 Politique de réécriture

Outre des opérations de lecture en mémoire principale, le processeur effectue également des écritures, par exemple lorsqu'il modifie une donnée. Soit celle-ci n'est qu'en mémoire principale, soit, et cela est plus probable, elle est également dans le cache. Que faut-il alors modifier quand l'information à changer se trouve à deux endroits distincts ? Deux méthodes existent et, comme toujours, la complexité supérieure de l'une est compensée par son efficacité.

Cache write-through

Le plus simple est, lors d'une écriture d'une donnée du cache, de propager l'écriture au niveau supérieur, c'est-à-dire en mémoire principale. Ce mécanisme de cache write-through (littéralement « écriture au travers », on l'appelle aussi « écriture immédiate ») a l'avantage de la simplicité et de la cohérence : l'information mémorisée dans le cache est toujours identique à celle située en mémoire principale. Lors d'un remplacement d'une ligne, celle-ci peut-être simplement éliminée du cache puisqu'elle existe sous la même forme en mémoire principale pour un éventuel usage futur (voir figure 6.9).

Mais cela se traduit par une perte de temps puisque l'écriture doit à chaque fois être répercutée en mémoire principale, qui a un temps d'accès plus élevé. Ce handicap, allant à l'encontre de l'idée même du cache, fait que ce mécanisme n'est plus très utilisé sur les processeurs actuels.

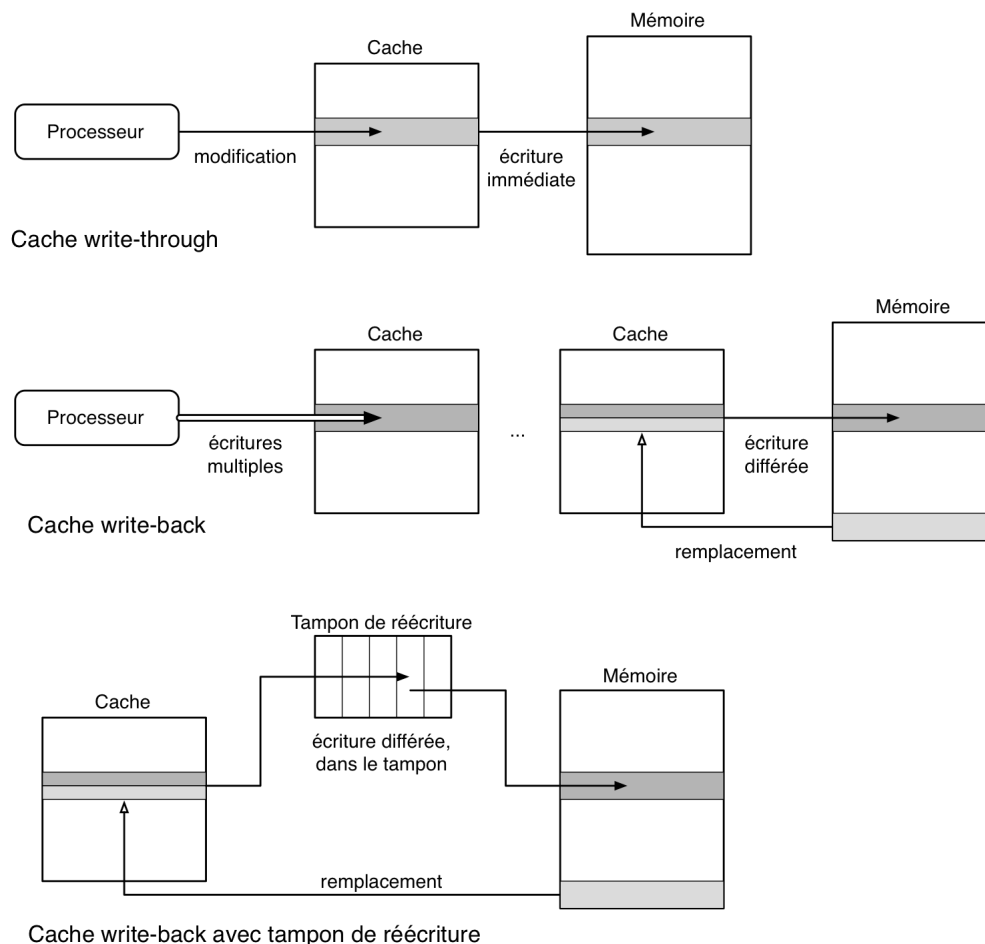
Cache write-back

Rien n'interdit de ne modifier l'information en mémoire principale que lorsque la ligne correspondante disparaît du cache. Le principe du cache write-back (à écriture différée) est, lors d'une écriture, de changer l'information stockée dans le cache en laissant la mémoire principale inchangée. Cela permet de ne pas perdre de temps puisque l'on accède uniquement au cache, beaucoup plus rapide.

En revanche, il faut penser à réécrire l'information correcte en mémoire principale au moment où la ligne est éjectée du cache (sinon il restera toujours l'information originelle en mémoire, et non sa valeur modifiée). Pour ce faire, on ajoute à chaque ligne un bit communément qualifié de « dirty » (bit sale), qui indique si la ligne a été modifiée et s'il faudra la réécrire en mémoire lors de son éviction. Cela prend un certain temps mais, par rapport à la méthode précédente, une ligne, même modifiée plusieurs fois, n'est retransmise qu'une seule fois en mémoire (précédemment elle l'était à chaque écriture), d'où des économies de temps d'accès et de bande passante (voir figure 6.9). En revanche, il est nécessaire de réécrire toute la ligne modifiée alors qu'un cache write-through peut se contenter de réécrire uniquement la donnée modifiée. Suivant les temps d'accès à la mémoire, cela peut être plus long et diminuer l'avantage des caches write-back.

Pour encore améliorer les performances, on peut prévoir un tampon de réécriture, chargé de transmettre à la mémoire les informations à modifier, déchargeant ainsi le cache de ce travail. Lorsqu'une ligne modifiée disparaît du cache, celui-ci l'écrit dans le tampon et redevient rapidement disponible pour une nouvelle requête, sans avoir à attendre que la mémoire principale ait terminé l'écriture de l'information ; c'est le rôle du tampon. Celui-ci peut même mémoriser plusieurs lignes à modifier, permettant au cache de travailler le plus rapidement possible. Le seul cas délicat à gérer est si, immédiatement après la disparition de la ligne modifiée, une nouvelle requête y fait référence car le tampon de réécriture n'a probablement pas terminé le transfert. Le cache doit donc examiner également les lignes stockées dans le tampon (voir figure 6.9).

Figure 6.9



Politiques de réécriture.

Ce défaut de cohérence entre l'information stockée en cache et celle encore en mémoire principale peut quand même poser problème dans le cas de composants accédant directement à la mémoire principale sans passer par le processeur. Certains dispositifs d'entrées/sorties, comme les accès DMA, transfèrent directement les informations entre la mémoire et les périphériques (voir chapitre 8).

Pour finir, il faut indiquer ce que l'on doit faire lorsque le processeur modifie une information qui ne se trouve pas dans le cache, mais uniquement en mémoire principale. Certains systèmes court-circuitent le cache en écrivant directement en mémoire (*no allocate on store miss*). Mais le plus fréquent est de transférer la ligne souhaitée dans le cache, ce qui ramène au cas précédent de l'écriture différée d'une ligne (*allocate on store miss*). L'information est donc déjà dans le cache dans l'éventualité d'un accès futur.

3. AMÉLIORATION DES CACHES

Depuis son introduction dans les années 1970, la mémoire cache a accru son importance du fait de l'augmentation de la vitesse des processeurs. Elle est maintenant un auxiliaire indispensable du processeur et les ingénieurs rivalisent d'inventivité pour améliorer ses performances.

3.1 Temps de récupération d'une instruction

Lorsque le processeur exécute une instruction, la première tâche consiste à la récupérer, soit dans le cache, soit en mémoire principale. S'ajoutant au temps d'exécution interne de l'instruction, ce temps d'accès représente une part importante du temps total d'exécution. Il est fonction des temps d'accès au cache (T_c) et de la pénalité en cas de défaut de cache (T_m , lié au temps d'accès à la mémoire principale), ainsi que du nombre respectif d'accès à l'un et à l'autre. $T_{\text{échec}}$ étant le taux d'échec, c'est-à-dire le pourcentage de défauts de cache, on a un temps moyen d'accès pour une instruction égal à :

$$T_{\text{accès}} = T_{\text{échec}} T_m + (1 - T_{\text{échec}}) T_c$$

Toute amélioration portée à l'un de ces termes améliore le temps moyen d'accès à une instruction, et donc l'efficacité du système.

3.2 Causes des défauts de cache

On peut classer les défauts de cache en trois catégories suivant la raison pour laquelle l'information souhaitée ne se trouve pas dans le cache :

- **Échec obligatoire.** Lors de la première référence à une adresse mémoire, celle-ci ne sera probablement pas dans le cache, sauf si une référence antérieure y a amené une ligne contenant cette adresse.
- **Échec de capacité.** Si un programme nécessite de nombreuses données, certaines d'entre elles risquent d'être extraites du cache pour laisser leur place à d'autres. Lorsque ce dernier est entièrement rempli, toute nouvelle référence mémoire éjecte du cache une ligne qui sera peut-être appelée de nouveau plus tard, provoquant un défaut de cache.
- **Échec de conflit.** Dans un cache associatif, un emplacement libre peut toujours être occupé par une nouvelle ligne alors que, dans une cache direct ou associatif par ensemble, il peut y avoir un conflit d'accès pour un même emplacement, même si d'autres sont libres. Une ligne est dans ce cas évincée du cache et risque de faire défaut plus tard.

Pour pallier les deux derniers cas d'échec, il faut agrandir le cache et éventuellement augmenter son degré d'associativité. C'est une tendance lourde dans les processeurs récents, qui profitent de l'intégration toujours plus poussée de transistors pour prévoir sur la puce une place de plus en plus importante réservée aux caches.

Il est difficile de diminuer le nombre d'échecs obligatoires. Comment faire en sorte qu'une information soit présente dans le cache, avant même d'en avoir besoin ? Certains processeurs essaient de résoudre ce problème en anticipant la lecture d'une ligne voisine d'une ligne à laquelle on a accédé : lors d'un défaut de cache, la ligne demandée est récupérée en mémoire, ainsi que la ligne suivante (bien que le processeur n'y fasse pas référence). On espère que le principe de localité spatiale soit suffisamment vérifié pour en profiter. Le processeur peut aussi prévoir des instructions spéciales de chargement de données dans le cache, qui seront utilisées par le compilateur pour précharger des informations avant leur utilisation. Bien sûr, ce dernier moyen n'est efficace que si le processeur peut exécuter d'autres instructions et donc les chercher dans le cache, pendant le chargement des informations dans ce dernier (sinon, autant attendre simplement la première référence à l'information pour la ramener de la mémoire).

Optimisation du code

La gestion entièrement matérielle du cache ne dispense pas le programmeur de réfléchir à la façon dont le code accède aux données. Il faut essayer de profiter au maximum de la localité des accès pour améliorer l'efficacité du cache. Considérons le programme suivant :

Listing 6.1

```
int A[2000][2000];

main () {
    int n, i, j;

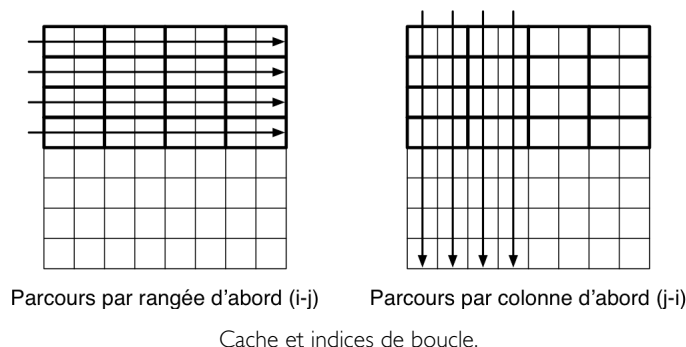
    for (n=0; n<500; n++)
        for (j=0; j<2000; j++)
            for (i=0; i<2000; i++)
                A[i][j] = 1;
}
```

Remplissage d'un tableau

Il est particulièrement inefficace car il accède aux données colonne par colonne alors qu'en C, les tableaux sont stockés en mémoire ligne par ligne. On ne profite donc pas de la localité spatiale lors des accès mémoire. Une simple permutation des indices i et j réduit le temps d'exécution de près de 70 % ! (Ce chiffre dépend évidemment du compilateur, du processeur, de la taille et des autres caractéristiques du cache.)

L'explication est donnée à la figure 6.10. Imaginons que le cache ait des lignes de 2 000 octets, correspondant à cinq cents éléments du tableau. Celui-ci étant stocké rangée par rangée, il y a quatre lignes du cache par rangée (gros traits sur la figure). Lorsque le tableau est parcouru par rangée d'abord, le processeur accède à des données successives dans les lignes du cache ; il profite donc au maximum du principe de localité spatiale. À l'inverse, lorsqu'on le parcourt par colonne, les accès successifs se font dans des lignes de cache différentes ; on est donc amené à récupérer une nouvelle ligne à chaque accès. Si, en raison de sa taille, le cache ne peut contenir toute une colonne, on ne peut réutiliser une ligne déjà chargée lors du passage à la colonne suivante.

Figure 6.10



Un second exemple, simulant une multiplication de matrices de nombres flottants, fonctionne de manière identique :

Listing 6.2

```
float x[1000][1000];
float y[1000][1000];
float z[1000][1000];

main () {
    int i, j, k;
    for (k=0; k<1000; k++)
        for (j=0; j<1000; j++)
            for (i=0; i<1000; i++)
                x[i][j] += y[i][k] * z[k][j];
}
```

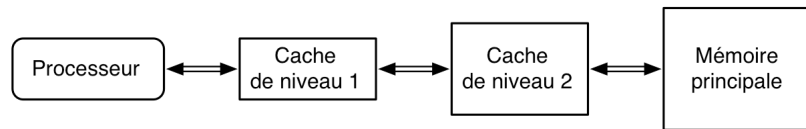
Multiplication de matrice

Là encore, on observe un gain de près de 50 % si l'on met les indices dans un meilleur ordre : $i-j-k$ (meilleur par rapport au stockage rangée par rangée des tableaux en mémoire). On peut même améliorer encore plus le résultat en choisissant un ordre $i-k-j$ (voir l'exercice 4 du chapitre 7).

3.3 Hiérarchiser la mémoire : caches multiniveaux

Pour améliorer le temps d'accès au cache, il faut diminuer sa taille (pour des raisons liées à la construction des circuits électroniques). Mais, pour abaisser le taux d'échec, il est important d'augmenter cette taille. Comment concilier ces deux impératifs *a priori* contradictoires ? Tout simplement en séparant le cache en deux : d'un côté, une petite zone rapide, et de l'autre, une partie plus lente mais plus conséquente (voir figure 6.11).

Figure 6.11



Caches multiniveaux.

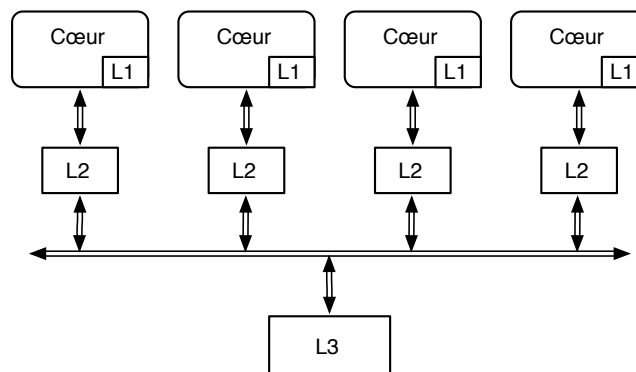
Le cache de niveau 1 (aussi qualifié de « primaire » ou « L1 » pour *Level 1*) est implanté au plus près du processeur, au cœur de la puce. Pour qu'il puisse être rapide (son temps d'accès est de l'ordre de quelques nanosecondes), il est relativement petit (classiquement de 32 Ko à 128 Ko), utilise l'écriture write-through et fonctionne souvent en adressage direct. On sépare ce cache en deux, une partie pour les instructions et une autre pour les données, ce qui permet de doubler la bande passante entre le cache et le processeur, tout en optimisant les caractéristiques de chacun pour leur usage : par exemple, il est inutile de prévoir de mécanisme de réécriture pour le cache d'instructions (pas de programme automodifiant), ce qui permet de gagner de la place.

Lorsqu'une requête mémoire provoque un défaut du cache de niveau 1, elle est transmise au cache de niveau 2. Pour qu'elle ait alors une chance d'aboutir (que l'information soit trouvée), il faut évidemment que celui-ci ait une capacité plus importante, typiquement de 512 Ko à 2 Mo. En contrepartie, il a un temps d'accès plus long (une dizaine de nanosecondes), mais peut être plus compliqué (associatif par ensemble de 4 ou 8, algorithme de remplacement LRU, réécriture write-back...). Le cache de niveau 2 était au départ séparé du processeur (dans un boîtier extérieur), mais physiquement placé sur une même carte et relié à lui par un bus spécial à haut débit (on l'appelait alors « cache externe », par opposition au cache de niveau 1, dit « interne »).

Systèmes multiprocesseurs

Les processeurs actuels ont plusieurs cœurs d'exécution et, toujours dans le but d'accélérer les transferts mémoire, chacun de ces cœurs possède son propre cache de niveau 1, intégré au sein même du pipeline (taille typique de 32 Ko). Un deuxième niveau de cache, là encore propre au cœur d'exécution, permet de mémoriser plus d'informations dans typiquement 256 Ko. Pour finir, un cache de niveau 3 conséquent (plusieurs Méga-octets) est partagé par toutes les unités d'exécution, et parfois même par d'autres entités comme un processeur graphique (voir figure 6.12). Tous ces composants sont maintenant directement intégrés sur la puce en silicium formant le processeur.

Figure 6.12



Processeurs multicœurs et caches.

Cette conception pose le problème de la cohérence des caches. Si tous les processeurs ont leur propre cache, il est possible que l'un d'entre eux fasse une requête pour une information qui provoque un défaut de son cache et la récupère en mémoire principale. Mais un autre processeur a peut-être déjà modifié cette information dans son cache sans avoir répercuté l'opération en mémoire. Il y a alors une incohérence entre les différentes copies de l'information (copies en caches et en mémoire principale). Cela peut par exemple se produire

lorsque qu'un processus possède plusieurs threads, chacun s'exécutant sur un cœur différent et tous partageant une zone de données commune. Pour résoudre ce problème de cohérence des caches, il faut complexifier les circuits de sorte qu'un cache puisse « prévenir » un autre processeur qu'une information qu'il demande a été modifiée et que celle stockée en mémoire principale ou dans son cache n'est plus valide. Les systèmes actuels intègrent dans ce but des protocoles similaires à celui que nous détaillons ci-après.

Le protocole MESI

Chaque ligne de donnée des caches est affectée d'un état parmi les quatre suivants (états qui ont donné l'acronyme nommant le protocole) :

- **Modified** (modifié). La ligne est uniquement présente dans ce cache, elle a été modifiée et n'a pas encore été réécrite en mémoire principale. Une fois cette réécriture effectuée, la ligne passera dans l'état *Exclusive*.
- **Exclusive** (exclusif). La ligne est uniquement présente dans ce cache mais elle est identique à celle se trouvant en mémoire principale (soit parce qu'elle n'a pas été modifiée, soit parce qu'elle a déjà été réécrite). La ligne passera dans l'état *Shared* si un autre cache la charge, ou dans l'état *Modified* si la ligne est modifiée.
- **Shared** (partagé). La ligne est identique à celle se situant en mémoire principale et se retrouve peut-être également dans d'autres caches.
- **Invalid** (invalide). La ligne n'est pas valide dans ce cache.

Les règles de fonctionnement données par le protocole sont là pour garantir la cohérence des caches : il faut éviter de se retrouver dans une situation où plusieurs caches différents auraient une même ligne mémoire contenant des informations différentes.

- Une ligne d'un cache ne peut être modifiée que si elle est déjà dans l'état *Modified* ou l'état *Exclusive*. En effet, dans ce cas, le cache sait qu'aucun autre cache ne possède une copie de la ligne. En revanche, si la ligne est dans l'état *Shared*, il faut d'abord invalider les copies de la ligne se trouvant dans les autres caches. Pour ce faire, un message spécial est diffusé par le cache sur le bus commun demandant aux autres de faire passer la ligne dans l'état *Invalid*.
- Une fois la ligne modifiée (et donc dans l'état *Modified*), le cache doit surveiller les accès mémoire des autres caches pour être sûr qu'ils n'essayent pas de relire la ligne (qui a été invalidée par le mécanisme ci-dessus) directement depuis la mémoire principale : la ligne n'ayant pas encore été réécrite en mémoire, celle-ci contient les anciennes données erronées. Si un autre cache initie une opération de lecture mémoire de la ligne, le premier cache doit intercepter la requête (*snoop*), l'annuler, écrire la ligne modifiée en mémoire puis relancer la requête de l'autre cache. Pour finir, puisque ce deuxième cache aura une copie de la ligne, le premier cache fait passer la ligne de l'état *Modified* à l'état *Shared*.
- Même si une ligne du cache n'est pas modifiée, si elle est dans l'état *Exclusive*, le cache doit surveiller le bus et détecter d'éventuelles demandes de lecture mémoire effectuées par les autres caches concernant la même ligne ; s'il voit une telle requête, le cache doit faire passer sa ligne dans l'état *Shared*.

RÉSUMÉ

La mémoire cache permet d'optimiser les performances du processeur en mémorisant les informations (données et instructions) les plus utilisées. Beaucoup plus rapide que la mémoire principale, elle évite ainsi au processeur d'attendre pour disposer d'une information et commencer à travailler. Elle est aussi plus complexe et de moins grande capacité que la mémoire principale. On optimise son utilisation en jouant sur ses caractéristiques : taille, méthode d'adressage, algorithme, etc. On peut même développer plusieurs niveaux de cache pour adapter au mieux cette mémoire : petit cache très rapide près du processeur, grand cache plus complexe et plus lent proche de la mémoire. Tout cela ne dispense pas le programmeur d'un peu de réflexion pour adapter ses structures de données et leur utilisation aux dispositions du cache.

Problèmes et exercices

Pour exploiter la mémoire cache, il faut comprendre les différentes caractéristiques qui la définissent : adressage, choix d'un emplacement, politiques de remplacement et d'écriture, etc. Les exercices suivants mettent en pratique ces notions, tout en comparant les différentes configurations. Vous verrez également l'influence de l'écriture et de l'optimisation d'un programme sur l'utilisation du cache.

Exercice I : Calculer des temps d'exécution

Un programme se compose d'une boucle de vingt instructions à exécuter quatre fois. La boucle se trouve aux adresses mémoire 0 à 19 (on suppose pour simplifier que chaque adresse mémoire contient un mot qui permet le stockage d'une instruction). Le programme doit s'exécuter sur une machine possédant un cache d'une taille de seize instructions. Les temps d'accès à la mémoire principale et au cache sont respectivement M et C .

1) Le cache est direct, formé de huit lignes de deux instructions. En d'autres termes, les mots mémoire 0 et 1, 16 et 17, etc. vont à l'emplacement 0, que les mots 2 et 3, 18 et 19, etc. vont à l'emplacement 1, etc. Quel est le temps total d'exécution du programme, temps de calcul non compris ?

2) Le cache est maintenant associatif (une ligne mémoire peut venir à n'importe quel emplacement du cache) et la stratégie de remplacement utilisée est LRU (on remplace la ligne la moins récemment utilisée). Quel est le temps d'exécution du programme ?

3) Répondez aux deux premières questions en prenant un cache composé de quatre lignes de quatre mots.

Remarque : lorsque l'on cherche un mot en mémoire, en un temps M on ramène le mot au processeur et toute la ligne correspondante dans le cache. Le processeur peut donc tout de suite passer à l'instruction suivante.

L'exécution de chaque instruction prend soit un temps M si elle est uniquement en mémoire (on la ramène au processeur qui l'exécute), soit un temps C si elle est déjà en cache (c'est le temps d'accès au cache). Il faut donc savoir, pour chaque instruction, si elle se trouve déjà en cache et, sinon, à quel emplacement mettre la ligne correspondante.

1) La première boucle se déroule comme suit :

Tableau 6.1

Instruction	Temps d'exécution	Mise en cache
mot 0	M (cache miss)	emplacement 0
mot 1	C (cache hit)	lecture empl. 0
mot 2	M (cache miss)	emplacement 1
mot 3	C (cache hit)	lecture empl. 1
mot 4	M (cache miss)	emplacement 2
mot 5	C (cache hit)	lecture empl. 2
mot 6	M (cache miss)	emplacement 3
mot 7	C (cache hit)	lecture empl. 3
mot 8	M (cache miss)	emplacement 4

mot 9	<i>C (cache hit)</i>	<i>lecture empl. 4</i>
mot 10	<i>M (cache miss)</i>	emplacement 5
mot 11	<i>C (cache hit)</i>	<i>lecture empl. 5</i>
mot 12	<i>M (cache miss)</i>	emplacement 6
mot 13	<i>C (cache hit)</i>	<i>lecture empl. 6</i>
mot 14	<i>M (cache miss)</i>	emplacement 7
mot 15	<i>C (cache hit)</i>	<i>lecture empl. 7</i>
mot 16	<i>M (cache miss)</i>	emplacement 0
mot 17	<i>C (cache hit)</i>	<i>lecture empl. 0</i>
mot 18	<i>M (cache miss)</i>	emplacement 1
mot 19	<i>C (cache hit)</i>	<i>lecture empl. 1</i>

Première boucle

Les deux premiers emplacements sont partagés, les six autres servent exclusivement aux lignes stockées, qui seront donc toujours dans le cache. La première boucle prend alors $10M + 10C$.

Lors de la deuxième boucle, les mots 0, 1, 2 et 3 sont à récupérer en mémoire puisque les emplacements 0 et 1 sont occupés par les mots 16 à 19, qui disparaîtront de ce fait. La seconde boucle prend donc $2M + 2C$ (mots 0 à 3), suivi de $12C$ (mots 4 à 15) et encore de $2M + 2C$ (mots 16 à 19), soit $4M + 16C$.

Les troisième et quatrième boucles sont identiques, ce qui donne un temps total de :

$$T = 10M + 10C + 3 \times (4M + 16C) = 22M + 58C$$

2) La première boucle se déroule comme précédemment, mais lors de la deuxième boucle, les mots 0 et 1 se mettent à l'emplacement 2 (celui dont l'utilisation est la plus ancienne), puis les mots 2 et 3 à l'emplacement 3, et ainsi de suite. À chaque fois sont éjectés du cache les mots qui vont servir un peu plus tard. Il n'y a donc jamais réutilisation des lignes stockées et cela donne un temps total de :

$$T = 4 \times (10M + 10C) = 40M + 40C$$

M étant plus grand que C , le temps d'exécution est plus long dans ce cas-là.

3) Dans un cache direct, seul le premier emplacement est partagé par les mots 0 à 3 et 16 à 19, tous les autres sont remplis à la première boucle et réutilisés aux trois autres. Cela donne un temps total de :

$$T = 5 \times (M + 3C) + 3 \times (2 \times (M + 3C) + 3 \times 4C) = 11M + 69C$$

Dans un cache associatif, il y a le même problème de non-réutilisation d'un emplacement et le temps total se calcule de la même manière :

$$T = 4 \times (5 \times (M + 3C)) = 20M + 60C$$

Cet exemple montre qu'un cache direct peut être plus intéressant qu'un cache associatif. C'est le cas ici car la boucle est juste un peu plus longue que la capacité du cache, éliminant à chaque tour les instructions qui vont être réutilisées.

Exercice 2 : Comparer les tailles des lignes

Un programme se compose d'une boucle de vingt-quatre instructions à exécuter quatre fois. Les instructions se trouvent, dans l'ordre, aux adresses mémoire 24 à 31, 0 à 3, 24 à 31, 4 à 7 (on suppose pour simplifier que chaque adresse mémoire contient un mot qui permet le stockage d'une instruction). Le programme doit s'exécuter sur une machine possédant un cache d'une taille de douze instructions. Les temps d'accès à la mémoire principale et au cache sont respectivement M et C . Le cache est associatif (une ligne mémoire peut venir à n'importe quel emplacement du cache) et l'algorithme de remplacement utilisé est LRU (on remplace la ligne la moins récemment utilisée).

1) Le cache possède six emplacements de deux instructions. Les lignes que l'on peut transférer sont 0-1, 2-3... 24-25, 26-27, 28-29, 30-31... Quel est le temps total d'exécution du programme, temps de calcul non compris ? Le cache est vide au départ.

- 2) Le cache possède trois emplacements de quatre instructions. Les lignes que l'on peut transférer sont 0-3, 4-7..., 24-27, 28-31... Quel est le temps total d'exécution du programme, temps de calcul non compris ? Le cache est vide au départ.
- 3) Le cache possède deux emplacements de six instructions. Les lignes que l'on peut transférer sont 0-5, 6-11..., 24-29, 30-35... Quel est le temps total d'exécution du programme, temps de calcul non compris ? Le cache est vide au départ.
- 4) Le cache possède un emplacement de douze instructions. Les lignes que l'on peut transférer sont 0-11... 24-35... Quel est le temps total d'exécution du programme, temps de calcul non compris ? Le cache est vide au départ.
- 5) Qu'en concluez-vous sur l'efficacité du cache ?

1) Voici le tableau d'exécution des deux premières boucles :

Tableau 6.2

Boucle 1			Boucle 2		
Instruction	Temps	Cache	Instruction	Temps	Cache
mots 24 et 25	$M + C$	emplac. 0	mots 24 et 25	$2C$	emplac. 0
mots 26 et 27	$M + C$	emplac. 1	mots 26 et 27	$2C$	emplac. 1
mots 28 et 29	$M + C$	emplac. 2	mots 28 et 29	$2C$	emplac. 2
mots 30 et 31	$M + C$	emplac. 3	mots 30 et 31	$2C$	emplac. 3
mots 0 et 1	$M + C$	emplac. 4	mots 0 et 1	$M + C$	emplac. 4
mots 2 et 3	$M + C$	emplac. 5	mots 2 et 3	$M + C$	emplac. 5
mots 24 et 25	$2C$	emplac. 0	mots 24 et 25	$2C$	emplac. 0
mots 26 et 27	$2C$	emplac. 1	mots 26 et 27	$2C$	emplac. 1
mots 28 et 29	$2C$	emplac. 2	mots 28 et 29	$2C$	emplac. 2
mots 30 et 31	$2C$	emplac. 3	mots 30 et 31	$2C$	emplac. 3
mots 4 et 5	$M + C$	emplac. 4	mots 4 et 5	$M + C$	emplac. 4
mots 6 et 7	$M + C$	emplac. 5	mots 6 et 7	$M + C$	emplac. 5

Les deux premières boucles (question 1)

Il faut faire attention à ne pas calculer uniquement le temps de la première boucle et de le multiplier par 4. En effet, au début de la première boucle, le cache est vide alors qu'il y a déjà des informations dedans avant la deuxième. Cela permet à cette boucle d'aller plus rapidement en récupérant des instructions directement dans le cache.

Les boucles 3 et 4 sont identiques à la deuxième. Cela donne un temps total de :

$$T = 8M + 16C + 3 \times (4M + 20C) = 20M + 76C$$

2) Voici le tableau d'exécution des deux premières boucles :

Tableau 6.3

Boucle 1			Boucle 2		
Instruction	Temps	Cache	Instruction	Temps	Cache
mots 24 à 27	$M + 3C$	emplac. 0	mots 24 à 27	$4C$	emplac. 0
mots 28 à 31	$M + 3C$	emplac. 1	mots 28 à 31	$4C$	emplac. 1
mots 0 à 3	$M + 3C$	emplac. 2	mots 0 à 3	$M + 3C$	emplac. 2
mots 24 à 27	$4C$	emplac. 0	mots 24 à 27	$4C$	emplac. 0
mots 28 à 31	$4C$	emplac. 1	mots 28 à 31	$4C$	emplac. 1

mots 4 à 7	$M + 3C$	emplac. 2	mots 4 à 7	$M + 3C$	emplac. 2
------------	----------	-----------	------------	----------	-----------

Les deux premières boucles (question 2)

Les boucles 3 et 4 sont identiques à la deuxième. Cela donne un temps total de :

$$T = 4M + 20C + 3 \times (2M + 22C) = 10M + 86C$$

3) Voici le tableau d'exécution des deux premières boucles :

Tableau 6.4

Boucle 1			Boucle 2		
Instruction	Temps	Cache	Instruction	Temps	Cache
mots 24 à 29	$M + 5C$	emplac. 0	mots 24 à 29	$M + 5C$	emplac. 1
mots 30 à 31	$M + C$	emplac. 1	mots 30 à 31	$M + C$	emplac. 0
mots 0 à 3	$M + 3C$	emplac. 0	mots 0 à 3	$M + 3C$	emplac. 1
mots 24 à 29	$M + 5C$	emplac. 1	mots 24 à 29	$M + 5C$	emplac. 0
mots 30 à 31	$M + C$	emplac. 0	mots 30 à 31	$M + C$	emplac. 1
mots 4 à 5	$M + C$	emplac. 1	mots 4 à 5	$M + C$	emplac. 0
mots 6 à 7	$M + C$	emplac. 0	mots 6 à 7	$M + C$	emplac. 1

Les deux premières boucles (question 3)

Les temps des boucles 3 et 4 sont identiques à celui de la deuxième. Cela donne un temps total de :

$$T = 4 \times (7M + 17C) = 28M + 68C$$

Il faut faire attention à deux points : tout d'abord, lorsque l'on exécute l'instruction 30, on cherche (en un temps M) la ligne contenant les instructions 30 à 35, mais on n'exécute que l'instruction 31 en plus (soit un temps C supplémentaire) ; ensuite, les instructions 4 et 5 se trouvent sur la ligne regroupant les mots 0 à 5, ce qui fait que l'on a besoin d'un accès mémoire supplémentaire pour récupérer l'instruction 6 (et ramener les lignes 6 à 11).

4) Voici le tableau d'exécution des deux premières boucles :

Tableau 6.5

Boucle 1			Boucle 2		
Instruction	Temps	Cache	Instruction	Temps	Cache
mots 24 à 31	$M + 7C$	emplac. 0	mots 24 à 31	$M + 7C$	emplac. 0
mots 0 à 3	$M + 3C$	emplac. 0	mots 0 à 3	$M + 3C$	emplac. 0
mots 24 à 31	$M + 7C$	emplac. 0	mots 24 à 31	$M + 7C$	emplac. 0
mots 4 à 7	$M + 3C$	emplac. 0	mots 4 à 7	$M + 3C$	emplac. 0

Les deux premières boucles (question 4)

Les temps des boucles 3 et 4 sont identiques à celui de la deuxième. Cela donne un temps total de :

$$T = 4 \times (4M + 20C) = 16M + 80C$$

5) L'efficacité du cache n'est pas proportionnelle à la taille de ses lignes. De grandes lignes permettent de mémoriser plus d'information à chaque accès mémoire, mais de petites lignes permettent de garder différentes parties du programme souvent réutilisées. Ainsi, la solution 3 (des lignes de six instructions) empêchent de capitaliser sur les instructions 25 à 32, exécutées deux fois par boucle : la ligne les contenant est éjectée trop rapidement du cache.

Exercice 3 : Comparer les adresses mémoire

Un programme se compose d'une boucle de trente-six instructions à exécuter trois fois. Les instructions se trouvent, dans l'ordre, aux adresses mémoire 0 à 7, 76 à 83, 8 à 15, 76 à 83, 16 à 19 (on suppose pour simplifier que chaque adresse mémoire contient un mot qui permet le stockage d'une instruction). Le programme doit s'exécuter sur une machine possédant un cache d'une taille de seize instructions. Les temps d'accès à la mémoire principale et au cache sont respectivement M et C . Le cache est associatif (une ligne mémoire peut venir à n'importe quel emplacement du cache) et l'algorithme de remplacement utilisé est LRU (on remplace la ligne la moins récemment utilisée).

1) Le cache possède quatre emplacements de quatre instructions. Les lignes que l'on peut transférer sont 0-3, 4-7... 72-75, 76-79, 80-83... Quel est le temps total d'exécution du programme, temps de calcul non compris ? Le cache est vide au départ.

2) Le cache possède deux emplacements de huit instructions. Les lignes que l'on peut transférer sont 0-7, 8-15... 72-79, 80-87... Quel est le temps total d'exécution du programme, temps de calcul non compris ? Le cache est vide au départ.

3) Répondez aux deux questions précédentes en supposant que les instructions sont aux adresses mémoire 0 à 7, 74 à 81, 8 à 15, 74 à 81, 16 à 19. Le cache est vide au départ.

1) Voici le tableau d'exécution des deux premières boucles :

Tableau 6.6

Boucle 1			Boucle 2		
Instruction	Temps	Cache	Instruction	Temps	Cache
mots 0 à 3	$M + 3C$	emplac. 0	mots 0 à 3	$M + 3C$	emplac. 1
mots 4 à 7	$M + 3C$	emplac. 1	mots 4 à 7	$M + 3C$	emplac. 2
mots 76 à 79	$M + 3C$	emplac. 2	mots 76 à 79	$M + 3C$	emplac. 3
mots 80 à 83	$M + 3C$	emplac. 3	mots 80 à 83	$M + 3C$	emplac. 0
mots 8 à 11	$M + 3C$	emplac. 0	mots 8 à 11	$M + 3C$	emplac. 1
mots 12 à 15	$M + 3C$	emplac. 1	mots 12 à 15	$M + 3C$	emplac. 2
mots 76 à 79	$4C$	emplac. 2	mots 76 à 79	$4C$	emplac. 3
mots 80 à 83	$4C$	emplac. 3	mots 80 à 83	$4C$	emplac. 0
mots 16 à 19	$M + 3C$	emplac. 0	mots 16 à 19	$M + 3C$	emplac. 1

Les deux premières boucles (question 1)

La troisième boucle est identique à la deuxième. Cela donne un temps total de :

$$T = 3 \times (7M + 29C) = 21M + 87C$$

2) Voici le tableau d'exécution des deux premières boucles :

Tableau 6.7

Boucle 1			Boucle 2		
Instruction	Temps	Cache	Instruction	Temps	Cache
mots 0 à 7	$M + 7C$	emplac. 0	mots 0 à 7	$M + 7C$	emplac. 1
mots 76 à 79	$M + 3C$	emplac. 1	mots 76 à 79	$M + 3C$	emplac. 0
mots 80 à 83	$M + 3C$	emplac. 0	mots 80 à 83	$M + 3C$	emplac. 1
mots 8 à 15	$M + 7C$	emplac. 1	mots 8 à 15	$M + 7C$	emplac. 0
mots 76 à 79	$M + 3C$	emplac. 0	mots 76 à 79	$M + 3C$	emplac. 1

mots 80 à 83	$M + 3C$	emplac. 1	mots 80 à 83	$M + 3C$	emplac. 0
mots 16 à 19	$M + 3C$	emplac. 0	mots 16 à 19	$M + 3C$	emplac. 1

Les deux premières boucles (question 2)

La troisième boucle est identique à la deuxième. Cela donne un temps total de :

$$T = 3 \times (7M + 29C) = 21M + 87C$$

Les emplacements étant plus grands, on gagne du temps grâce au principe de localité spatiale (on exécute plusieurs instructions de la même ligne), mais on en perd aussi car, comme il y a moins d'emplacements, on ne peut plus profiter du principe de réutilisation. On retombe un peu par hasard sur la même valeur finale.

3) Voici le tableau d'exécution des deux premières boucles :

Tableau 6.8

Boucle 1			Boucle 2		
Instruction	Temps	Cache	Instruction	Temps	Cache
mots 0 à 3	$M + 3C$	emplac. 0	mots 0 à 3	$M + 3C$	emplac. 3
mots 4 à 7	$M + 3C$	emplac. 1	mots 4 à 7	$M + 3C$	emplac. 0
mots 74 à 75	$M + C$	emplac. 2	mots 74 à 75	$M + C$	emplac. 1
mots 76 à 79	$M + 3C$	emplac. 3	mots 76 à 79	$M + 3C$	emplac. 2
mots 80 à 81	$M + C$	emplac. 0	mots 80 à 81	$M + C$	emplac. 3
mots 8 à 11	$M + 3C$	emplac. 1	mots 8 à 11	$M + 3C$	emplac. 0
mots 12 à 15	$M + 3C$	emplac. 2	mots 12 à 15	$M + 3C$	emplac. 1
mots 74 à 75	$M + C$	emplac. 3	mots 74 à 75	$M + C$	emplac. 2
mots 76 à 79	$M + 3C$	emplac. 0	mots 76 à 79	$M + 3C$	emplac. 3
mots 80 à 81	$M + C$	emplac. 1	mots 80 à 81	$M + C$	emplac. 0
mots 16 à 19	$M + 3C$	emplac. 2	mots 16 à 19	$M + 3C$	emplac. 1

Les deux premières boucles (question 3-1)

La troisième boucle est identique à la deuxième. Cela donne un temps total de :

$$T = 3 \times (11M + 25C) = 33M + 75C$$

En changeant l'adresse de placement d'une partie du programme, on dégrade les performances car on n'utilise plus le cache de façon rationnelle : les lignes mémoire sont réparties sur un trop grand nombre d'emplacements, empêchant leur réutilisation.

Voici le tableau d'exécution des deux premières boucles si le cache possède deux emplacements de huit mots :

Tableau 6.9

Boucle 1			Boucle 2		
Instruction	Temps	Cache	Instruction	Temps	Cache
mots 0 à 7	$M + 7C$	emplac. 0	mots 0 à 7	$M + 7C$	emplac. 1
mots 74 à 79	$M + 5C$	emplac. 1	mots 74 à 79	$M + 5C$	emplac. 0
mots 80 à 81	$M + C$	emplac. 0	mots 80 à 81	$M + C$	emplac. 1
mots 8 à 15	$M + 7C$	emplac. 1	mots 8 à 15	$M + 7C$	emplac. 0
mots 74 à 79	$M + 5C$	emplac. 0	mots 74 à 79	$M + 5C$	emplac. 1
mots 80 à 81	$M + C$	emplac. 1	mots 80 à 81	$M + C$	emplac. 0
mots 16 à 19	$M + 3C$	emplac. 0	mots 16 à 19	$M + 3C$	emplac. 1

Les deux premières boucles (question 3–2)

La troisième boucle est identique à la deuxième. Cela donne un temps total de :

$$T = 3 \times (7M + 29C) = 21M + 87C$$

Comme il n'y a pas réutilisation du cache pour la première adresse de placement, il n'y a pas ici d'inconvénient à décaler l'adresse du programme.

Cet exercice montre que de nombreux facteurs peuvent jouer sur l'efficacité du cache, le programmeur n'ayant pas forcément de prise sur certains d'entre eux, qui peuvent être décidés par le compilateur ou le système d'exploitation.

Exercice 4 : Optimiser un programme

Une machine possède un cache d'une taille de seize instructions regroupées en quatre lignes de quatre instructions. Le cache est direct. En d'autres termes, les mots mémoire 0–3, 16–19, etc. vont à l'emplacement 0 du cache, les mots 4–7, 20–23, etc. vont à l'emplacement 1, etc. Les temps d'accès à la mémoire principale et au cache sont respectivement M et C .

1) Un programme se compose d'une boucle de quarante instructions à exécuter trois fois. Les instructions se trouvent, dans l'ordre, aux adresses mémoire 0 à 19, 20 à 23, 0 à 3, 20 à 23, 0 à 3, 20 à 23.

Quel est le temps total d'exécution du programme, temps de calcul non compris ? Le cache est vide au départ.

2) Le compilateur réussit à optimiser le programme et en a réduit la taille en gagnant quelques instructions. Il se compose d'une boucle de trente-six instructions à exécuter trois fois. Les instructions se trouvent, dans l'ordre, aux adresses mémoire 0 à 15, 16 à 19, 0 à 3, 16 à 19, 0 à 3, 16 à 19.

Quel est le temps total d'exécution du programme, temps de calcul non compris ? Le cache est vide au départ.

3) Que concluez-vous des deux précédents résultats ?

1) Voici le tableau d'exécution des deux premières boucles :

Tableau 6.10

Boucle 1			Boucle 2		
Instruction	Temps	Cache	Instruction	Temps	Cache
mots 0 à 3	$M + 3C$	emplac. 0	mots 0 à 3	$4C$	emplac. 0
mots 4 à 7	$M + 3C$	emplac. 1	mots 4 à 7	$M + 3C$	emplac. 1
mots 8 à 11	$M + 3C$	emplac. 2	mots 8 à 11	$4C$	emplac. 2
mots 12 à 15	$M + 3C$	emplac. 3	mots 12 à 15	$4C$	emplac. 3
mots 16 à 19	$M + 3C$	emplac. 0	mots 16 à 19	$M + 3C$	emplac. 0

mots 20 à 23	$M + 3C$	emplac. 1	mots 20 à 23	$M + 3C$	emplac. 1
mots 0 à 3	$M + 3C$	emplac. 0	mots 0 à 3	$M + 3C$	emplac. 0
mots 20 à 23	$4C$	emplac. 1	mots 20 à 23	$4C$	emplac. 1
mots 0 à 3	$4C$	emplac. 0	mots 0 à 3	$4C$	emplac. 0
mots 20 à 23	$4C$	emplac. 1	mots 20 à 23	$4C$	emplac. 1

Les deux premières boucles (question 1)

Les deux premiers emplacements sont partagés entre plusieurs lignes, ce qui nuit à l'efficacité. Néanmoins, on parvient à profiter (principe de localité temporelle) de la répétition de certaines instructions. La troisième boucle est identique à la deuxième. Cela donne un temps total de :

$$T = 7M + 33C + 2 \times (4M + 36C) = 15M + 105C$$

2) Voici le tableau d'exécution des deux premières boucles :

Tableau 6.11

Boucle 1			Boucle 2		
Instruction	Temps	Cache	Instruction	Temps	Cache
mots 0 à 3	$M + 3C$	emplac. 0	mots 0 à 3	$M + 3C$	emplac. 0
mots 4 à 7	$M + 3C$	emplac. 1	mots 4 à 7	$4C$	emplac. 1
mots 8 à 11	$M + 3C$	emplac. 2	mots 8 à 11	$4C$	emplac. 2
mots 12 à 15	$M + 3C$	emplac. 3	mots 12 à 15	$4C$	emplac. 3
mots 16 à 19	$M + 3C$	emplac. 0	mots 16 à 19	$M + 3C$	emplac. 0
mots 0 à 3	$M + 3C$	emplac. 0	mots 0 à 3	$M + 3C$	emplac. 0
mots 16 à 19	$M + 3C$	emplac. 0	mots 16 à 19	$M + 3C$	emplac. 0
mots 0 à 3	$M + 3C$	emplac. 0	mots 0 à 3	$M + 3C$	emplac. 0
mots 16 à 19	$M + 3C$	emplac. 0	mots 16 à 19	$M + 3C$	emplac. 0

Les deux premières boucles (question 2)

L'emplacement 0 est maintenant partagé par les lignes qui se répètent au sein de la boucle ! On perd donc tout le bénéfice du cache sur une boucle puisque ces lignes s'évincent mutuellement. On ne profite que de la répétition de certaines instructions d'une boucle sur l'autre. La troisième est identique à la deuxième. Cela donne un temps total de :

$$T = 9M + 27C + 2 \times (6M + 30C) = 21M + 87C$$

3) « L'amélioration » réalisée par le compilateur a augmenté le nombre d'accès mémoire par une mauvaise répartition des instructions dans les emplacements, même si l'on a gagné en nombre d'instructions. On a $6M$ en plus et $18C$ en moins. Par conséquent, si M est plus grand que $3C$, le programme « optimisé » s'exécute plus lentement !

Exercice 5 : Réduire la taille des boucles

Une machine possède un cache d'une taille de vingt-quatre instructions regroupées en six emplacements de quatre instructions. Le cache est associatif (une ligne mémoire peut venir à n'importe quel emplacement du cache) et l'algorithme de remplacement utilisé est LRU (on remplace la ligne la moins récemment utilisée). Les temps d'accès à la mémoire principale et au cache sont respectivement M et C .

1) Un programme se compose d'une boucle de vingt-huit instructions à exécuter cinq fois. Les instructions se trouvent, dans l'ordre, aux adresses mémoire 0 à 27. Quel est le temps total d'exécution du programme, temps de calcul non compris ? Le cache est vide au départ.

- 2) Le code est réorganisé et la boucle est maintenant découpée en deux : on exécute d'abord cinq fois les instructions 0 à 13 puis cinq fois les instructions 14 à 27. Quel est le temps total d'exécution du programme, temps de calcul non compris ? Le cache est vide au départ.
- 3) Que concluez-vous des deux précédents résultats ?

1) Voici le tableau d'exécution des deux premières boucles :

Tableau 6.12

Boucle 1			Boucle 2		
Instruction	Temps	Cache	Instruction	Temps	Cache
mots 0 à 3	$M + 3C$	emplac. 0	mots 0 à 3	$M + 3C$	emplac. 1
mots 4 à 7	$M + 3C$	emplac. 1	mots 4 à 7	$M + 3C$	emplac. 2
mots 8 à 11	$M + 3C$	emplac. 2	mots 8 à 11	$M + 3C$	emplac. 3
mots 12 à 15	$M + 3C$	emplac. 3	mots 12 à 15	$M + 3C$	emplac. 4
mots 16 à 19	$M + 3C$	emplac. 4	mots 16 à 19	$M + 3C$	emplac. 5
mots 20 à 23	$M + 3C$	emplac. 5	mots 20 à 23	$M + 3C$	emplac. 0
mots 24 à 27	$M + 3C$	emplac. 0	mots 24 à 27	$M + 3C$	emplac. 1

Les deux premières boucles (question 1)

Il n'y a aucune réutilisation du cache d'une boucle sur l'autre car celle-ci est trop grande pour tenir en entier dedans. L'algorithme de remplacement utilisé étant LRU, les instructions de la boucle sont enlevées du cache à chaque fois avant leur réutilisation.

Les trois boucles suivantes sont identiques à la deuxième. Cela donne un temps total de :

$$T = 5 \times (7M + 21C) = 35M + 105C$$

2) Voici le tableau d'exécution des deux premières itérations de la première boucle :

Tableau 6.13

Boucle 1			Boucle 2		
Instruction	Temps	Cache	Instruction	Temps	Cache
mots 0 à 3	$M + 3C$	emplac. 0	mots 0 à 3	$4C$	emplac. 0
mots 4 à 7	$M + 3C$	emplac. 1	mots 4 à 7	$4C$	emplac. 1
mots 8 à 11	$M + 3C$	emplac. 2	mots 8 à 11	$4C$	emplac. 2
mots 12 à 13	$M + C$	emplac. 3	mots 12 à 13	$2C$	emplac. 3

Les deux premières itérations de la première boucle

Le processeur peut maintenant profiter au maximum de l'efficacité du cache. L'intégralité de la première boucle est stockée en cache et, après l'avoir chargée lors de la première itération, on mesure la puissance du cache qui a supprimé tout accès mémoire. Les trois autres itérations étant identiques, on obtient un temps total pour la première boucle de :

$$T = 4M + 10C + 4 \times (14C) = 4M + 66C$$

La seconde boucle se déroule de façon équivalente :

Tableau 6.14

Boucle 1			Boucle 2		
Instruction	Temps	Cache	Instruction	Temps	Cache
mots 14 à 15	$2C$	emplac. 3	mots 14 à 15	$2C$	emplac. 3
mots 16 à 19	$M + 3C$	emplac. 4	mots 16 à 19	$4C$	emplac. 4

mots 20 à 23	$M + 3C$	emplac. 5	mots 20 à 23	$4C$	emplac. 5
mots 24 à 27	$M + 3C$	emplac. 0	mots 24 à 27	$4C$	emplac. 0

Les deux premières itérations de la seconde boucle

On gagne même un accès mémoire car les instructions 14 et 15 sont déjà dans la ligne qui a été ramenée à la fin de la première boucle (l'exécution de l'instruction 12 provoque le transfert de la ligne contenant les mots 12 à 15 dans le cache). Les trois autres itérations étant identiques, on obtient un temps total pour la seconde boucle de :

$$T = 3M + 11C + 4 \times (14C) = 3M + 67C$$

Soit un temps total de :

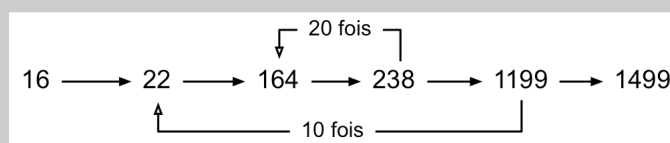
$$T = 7M + 133C$$

3) L'écriture du programme a une influence sur les performances du cache. Si une boucle tient entièrement dedans, il est beaucoup plus efficace. On peut donc avoir intérêt à multiplier de petites boucles, même si on ajoute des instructions, car elles peuvent ainsi résider en entier dans le cache et l'on profite au maximum du principe de réutilisation.

Exercice 6 : Un exemple plus important

Un programme se compose de deux boucles for imbriquées : une petite, interne, et une plus grande, externe. La structure générale du programme est donnée à la figure 6.13.

Figure 6.13



Structure du programme.

Les adresses décimales données déterminent l'emplacement des deux boucles ainsi que le début et la fin du programme. Tous les emplacements mémoire des différents blocs, 16–21, 22–163, 164–238, etc., contiennent des instructions devant être exécutées séquentiellement. Le programme tourne sur une machine possédant un cache. Celui-ci est direct, de taille globale de 1024 mots et chaque emplacement pour une ligne fait 128 mots. Une instruction occupe un mot en mémoire. Les temps d'accès à la mémoire principale et au cache sont respectivement M et C .

Quel est le temps total d'exécution du programme, temps de calcul non compris ? Le cache est vide au départ. Estimez le facteur d'amélioration résultant de l'utilisation du cache.

Le cache faisant 1024 mots et les lignes étant de 128 mots, il y a huit emplacements dans le cache. La correspondance est directe. Donc les mots mémoire 0 à 127 peuvent aller à l'emplacement 0, les mots 128 à 255 à l'emplacement 1... les mots 896 à 1023 à l'emplacement 7, les mots 1024 à 1151 à l'emplacement 0, les mots 1152 à 1279 à l'emplacement 1, etc.

Première itération

- L'exécution des instructions de 16 à 127 prend $M + 111C$ car, lors de l'accès au mot 16, on charge toute la première ligne dans le cache.
- Lors de l'accès au mot 128, on charge tous les mots de la deuxième ligne, c'est-à-dire tous les mots de 128 à 255. Donc, entre autres, toute la boucle interne est chargée. L'exécution prend $M + 127C + 19 \times 75C$. En effet, on exécute une fois toutes les instructions de 128 à 255, soit $M + 127C$, et encore dix-neuf fois la boucle interne, qui se trouve en entier dans le cache et comprend soixante-quinze instructions.
- Les instructions 256 à 383 prennent $M + 127C$ (comme d'habitude). De même pour 384 à 511, 512 à 639, 640 à 767, 768 à 895, 896 à 1023. Soit un total de $6M + 6 \times 127C$ pour les instructions de 256 à 1023.
- Lorsque l'on accède à l'instruction 1024, il faut la chercher en mémoire et transférer la ligne dans le cache. Or cette ligne va à l'emplacement 0. On enlève donc la ligne de mots d'adresses 0 à 127 du cache. Cela prend $M + 127C$ pour exécuter les instructions de 1024 à 1151.

- De même, l'accès au mot 1152 oblige à transférer les lignes 1152 à 1279 à l'emplacement 1 du cache, à la place des mots d'adresses 128 à 255. Cela prend $M + 47C$ d'exécuter ces instructions (car à 1199, on boucle).

Pour la première itération, on a un total de $10M + 2\,599C$.

Deuxième itération et suivantes

- L'emplacement 0 contient les adresses 1024 à 1151. Il faut donc récupérer les lignes 0 à 127 pour accéder à l'instruction à l'adresse 22. D'où un temps de $M + 105C$ pour les instructions 22 à 127.
- De même, les instructions 128 à 255 ne sont pas dans le cache (car l'emplacement 1 du cache comprend les instructions mémoire 1152 à 1279). Donc, comme avant, il faut $M + 127C + 19 \times 75C$ pour toutes ces instructions, en comptant la boucle interne.
- Par contre, les instructions 256 à 1023 sont dans le cache, elles n'ont pas été remplacées. Il faut donc $6 \times 128C$ pour les exécuter.
- L'emplacement 0 du cache ne contient plus les mots 1024 à 1151 : il faut $M + 127C$.
- De même, l'emplacement 1 ne contient pas ce qu'il faut : $M + 47C$.

La deuxième itération prend $4M + 2\,599C$. Les huit itérations suivantes sont identiques : $8 \times (4M + 2\,599C)$.

Fin du programme

- Les instructions 1200 à 1279 sont dans le cache : $80C$.
- On transfère 1280 à 1407 : $M + 127C$.
- Pour finir, on transfère 1408 à 1499 (en fait 1535) : $M + 91C$.

La fin du programme prend $2M + 298C$.

Total

Le temps total est donc de :

$$T = 10M + 2599C + 9 \times (4M + 2599C) + 2M + 298C = 48M + 26288C$$

L'efficacité du cache, mesurée comme le nombre d'accès au cache sur le nombre d'accès total, est de :

$$26288 / (26288 + 48) = 99,8\%$$

Exercice 7 : Influence de la réécriture

Un cache possède des lignes de 8 octets. Le temps d'accès à la mémoire principale lors d'un transfert de ligne est de 50 ns pour le premier octet et de 5 ns par octet suivant pour le reste de la ligne.

1) Quel est le temps moyen de récupération d'une ligne si le cache est write-through ? Si 30 % des lignes du cache sont modifiées, quel est le temps moyen de récupération d'une ligne si le cache est write-back ?

2) Si, en moyenne, chaque ligne est modifiée trois fois, combien de temps chaque cache passe-t-il à réécrire les données en mémoire principale lors du remplacement de la ligne ? À partir de combien de réécritures par ligne le cache write-back est-il plus intéressant ?

1) La récupération d'une ligne prend 50 ns pour le premier octet puis 5 ns pour chacun des sept octets restants, soit 85 ns au total. C'est le temps pour un cache write-through. En revanche, dans le cas d'un cache write-back, il faut inclure le temps de réécriture d'une ligne modifiée, soit de nouveau 85 ns, dans 30 % des cas. Le temps total est alors de $85 + 0,3 \times 85$ soit 110,5 ns.

2) Pour un cache write-through, chaque réécriture ne concerne qu'un octet et utilise donc 50 ns. Il y a trois réécritures, donc un temps moyen de 150 ns par ligne modifiée. Un cache write-back ne réécrit les données qu'une seule fois, au remplacement de la ligne, mais doit la réécrire en totalité ; Cela prend donc 85 ns.

Ce temps de 85 ns est le même quel que soit le nombre de modifications de la ligne dans un cache write-back. En revanche, un cache write-through utilise 50 ns par modification pour changer, en mémoire principale, l'octet modifié. S'il y a, en moyenne, plus de $\frac{85}{50} = 1,7$ changements par ligne modifiée, le cache write-back est plus avantageux.

Chapitre 7

Mémoire virtuelle

Il est intéressant d'individualiser les espaces mémoire des différents programmes s'exécutant sur le processeur. Cela permet de maximiser la taille de la mémoire proposée à un programme et d'introduire des mécanismes de protection de chacun de ces espaces. La mémoire virtuelle utilise la pagination pour séparer ces derniers de la mémoire physique en gardant sur le disque une partie de la mémoire utilisée. Les transferts entre disque et mémoire sont alors gérés par un composant intégré au processeur, la MMU, à l'aide de tables de correspondances entre espace virtuel et espace réel. De son côté, la segmentation autorise un découpage logique de l'espace mémoire d'un programme en fonction de son utilisation pour rendre plus efficace son positionnement en mémoire physique.

Mémoire virtuelle

- 1. Principe 162
- 2. Implémentation 164
- 3. Segmentation 172

Problèmes et exercices

- 1. Manipuler la table de pages 175
- 2. Saturer le processeur 178
- 3. Comparer cache et mémoire virtuelle..... 178
- 4. Multiplier des matrices en mémoire..... 180
- 5. Comparer les algorithmes de remplacement 182
- 6. Anomalie de Belady 183
- 7. Gérer les défauts de page..... 183

I. PRINCIPE

Dans le modèle de fonctionnement vu aux chapitres précédents, un seul programme se place en totalité en mémoire principale et s'exécute sur le processeur en ayant un accès complet à la mémoire via les références utilisées dans les instructions. Afin de détacher l'espace mémoire utilisé par le programme de l'espace mémoire physique réellement disponible, il est nécessaire que les références indiquées par ledit programme et utilisées par le processeur lors de l'exécution des instructions soient différentes de l'adressage physique permettant un accès aux boîtiers mémoire.

I.1 Exécution d'un programme

Sauf exception, un programme n'a pas besoin d'être stocké entièrement en mémoire principale pour être exécuté par un processeur, comme le montrent les exemples suivants :

- Le code peut souvent être découpé en unités fonctionnelles, qui ne sont pas activées ensemble. Un programme est rarement appelé à tout faire en même temps. Il est plutôt composé d'étapes successives déclenchées les unes après les autres. On peut donc se contenter d'amener en mémoire principale la partie active à un instant donné.
- Une partie du code d'un programme peut être réservée au traitement des erreurs inhabituelles. Les fonctions correspondantes ne seront donc peut-être jamais exécutées et n'ont aucun besoin de résider en mémoire principale pendant toute l'exécution du programme.
- On peut avoir défini une zone de données beaucoup plus grande que la partie réellement utilisée. Souvent, on déclare des structures (des tableaux par exemple) d'une taille suffisante pour contenir l'intégralité des données, mais on ne les remplit pas complètement, une grande partie étant alors inutilisée.

Se donner la possibilité de n'avoir qu'un processus (c'est-à-dire un programme en cours d'exécution) partiel en mémoire principale amène plusieurs avantages :

- Le code et les données d'un programme ne sont plus limités par la capacité de la mémoire principale.
- Les processus occupant moins de place en mémoire, il est possible d'en charger davantage et de les exécuter « en même temps » en traitant successivement quelques instructions de chacun d'entre eux (exécution multitâche).

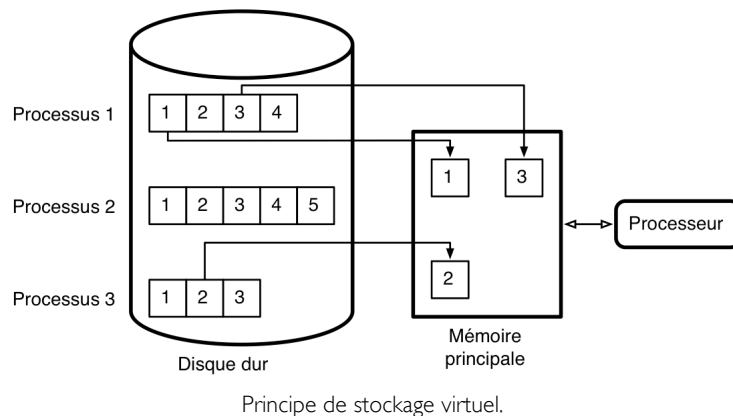
Avec la baisse du coût des boîtiers mémoire, permettant un accroissement conséquent de la capacité de la mémoire principale, ces deux avantages sont devenus moins pertinents. Néanmoins, la mémoire virtuelle garde toujours son intérêt dans le contexte de la multiprogrammation : elle protège les espaces mémoire des différents processus s'exécutant sur le processeur (voir section 2.3).

I.2 Pagination

Si un processus n'est que partiellement présent en mémoire principale, il faut se donner un autre support de mémorisation où placer le reste du programme, en l'occurrence le disque dur. Il réunit l'intégralité de l'espace mémoire de chaque processus, séparé de la mémoire physique et découpé en pages qui peuvent être également en mémoire physique (voir figure 7.1). Dans tous les cas, elles doivent l'être pour que le processeur puisse y accéder (il ne sait pas comment faire référence à une information sur le disque). L'espace de swap (c'est-à-dire d'échange) désigne la partie sur le disque réservée aux pages en question.

La figure 7.1 montre trois processus en cours d'exécution. Le premier a un espace mémoire composé de quatre pages dont deux sont présentes en mémoire principale. Le deuxième n'a aucune page en mémoire principale et ne pourra donc pas s'exécuter avant que le système ne rapatrie une de ses pages en mémoire. Le dernier a une page en mémoire, probablement celle en cours d'exécution.

Figure 7.1

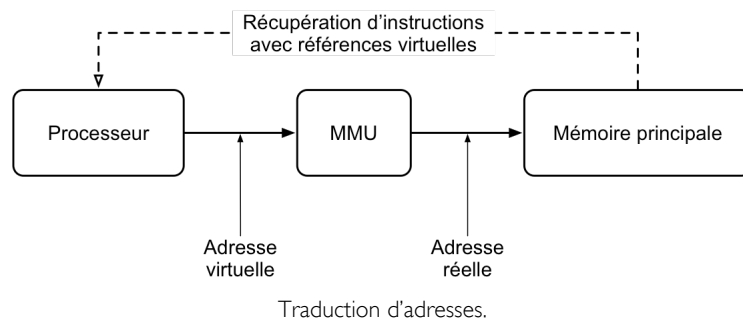


Adressages virtuel et réel

Puisque l'espace mémoire d'un processus et la mémoire physique sont disjoints, il faut prévoir un double adressage. Le premier est virtuel (aussi qualifié de « logique », voir section 3.2). Il permet au processus de faire des références mémoire dans son propre espace, quand il a besoin de l'adresse d'une instruction lors d'une rupture de séquence par exemple, ou d'une donnée en mémoire. Le processeur se sert également de ces références, par exemple pour récupérer la prochaine instruction à exécuter.

En revanche, les pages de chaque processus étant situées n'importe où en mémoire, il faut transformer ces adresses virtuelles en adresses physiques (dites aussi « réelles ») pour que l'on puisse faire référence à une donnée située dans un boîtier mémoire. Il y a donc une traduction à effectuer et c'est le rôle d'un composant spécial appelé MMU (*Memory Management Unit*, voir figure 7.2). Anciennement situé dans un circuit intégré extérieur au processeur (et donc optionnel), il est maintenant implanté directement dans celui-ci sur la puce en silicium. On gagne ainsi en rapidité de fonctionnement.

Figure 7.2



Le cœur du processeur (unité de contrôle, registres divers et PC...) ne manipule donc que des adresses virtuelles, les adresses réelles étant utilisées uniquement pour les accès à la mémoire principale et entre la MMU et les boîtiers mémoire.

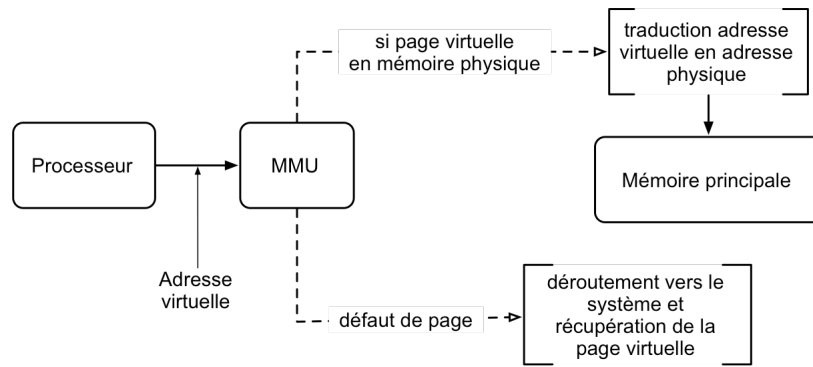
I.3 Algorithme d'accès à la mémoire

Supposons qu'il n'y ait qu'un seul processus en cours d'exécution sur le processeur (voir également section 2.3). La mémoire virtuelle située sur le disque, composée de l'espace virtuel du processus, est découpée en pages de quelques kilo-octets, souvent 4 Ko. Elles forment l'unité de transfert entre le disque et la mémoire principale. La mémoire physique est également découpée en emplacements de la taille d'une page, appelés « cadres de page » (*page frame*) et pouvant chacun contenir une page virtuelle.

Lorsque le processeur effectue un accès mémoire (lecture de la prochaine instruction, récupération d'une donnée), donc à l'aide d'une adresse virtuelle, les opérations suivantes sont effectuées (voir figure 7.3) :

1. L'adresse virtuelle est envoyée à la MMU.
2. Si la page virtuelle est déjà en mémoire physique dans un cadre, la MMU la traduit en adresse physique et poursuit l'accès mémoire. Sinon, il y a un défaut de page. Dans ce cas, la MMU doit donner la main au système d'exploitation pour qu'il cherche la page sur le disque et la mette en mémoire physique. Cela fait, le processeur peut tenter de nouveau l'accès mémoire.

Figure 7.3



Algorithme de la MMU.

Ce mécanisme, appelé « pagination », est efficace pour séparer la gestion de l'espace mémoire d'un processus (protection, taille) et la gestion de la mémoire réelle. Cependant, il y a un prix à payer : l'accès au support de stockage secondaire (le disque dur) pour récupérer une page virtuelle et l'amener en mémoire est beaucoup plus lent (environ 10 ms) que l'accès à la mémoire principale (environ 50 ns). C'est pénalisant du point de vue des performances.

2. IMPLÉMENTATION

Afin de mémoriser les correspondances entre les pages virtuelles et les cadres de page en mémoire physique, la MMU maintient une table qui enregistre les couples correspondants et génère les adresses physiques équivalentes. Plusieurs améliorations de l'implémentation sont possibles pour accélérer cette traduction.

2.1 Table des pages

La table des pages indique pour chaque page virtuelle si elle est déjà présente en mémoire physique et, si c'est le cas, dans quel cadre de page. C'est un grand tableau indexé par le numéro de page virtuelle et qui donne le numéro de page physique où elle est située (voir figure 7.4). Un bit de validité signale si l'information présente dans la ligne correspondante est correcte ou non, c'est-à-dire si elle donne bien un numéro de page physique ou s'il s'agit d'octets sans signification.

Cette table des pages est stockée en mémoire principale et son adresse physique est mémorisée dans un registre spécial de la MMU. Lors d'une traduction, la MMU effectue donc un premier accès mémoire pour récupérer l'entrée dans la table des pages avant de faire l'accès mémoire proprement dit (s'il n'y a pas de défaut de page).

Figure 7.4

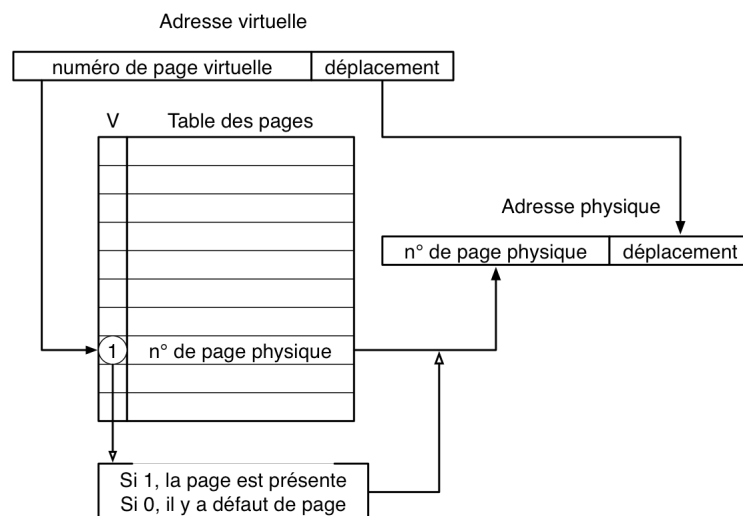


Table des pages.

Une adresse virtuelle est formée de deux parties : les bits de poids faible correspondent au déplacement dans la page permettant de récupérer l'octet demandé et le reste des bits forme le numéro de page virtuelle. Sur une page de 2^d octets, d bits indiquent le déplacement. Les tailles des numéros de page virtuelle ou physique sont liées à d'autres paramètres. Le nombre de bits des adresses physiques dépend de l'implémentation matérielle des boîtiers mémoire (puisqu'elles servent au final à accéder réellement à la mémoire principale). Longtemps formées de 32 bits, elles vont jusqu'à 64 bits (ou moins en raison de contraintes de place entre autres) dans les processeurs les plus récents. Quant à la taille des adresses virtuelles, elle est conditionnée par la capacité du processeur à les manipuler, via ses registres et son unité de contrôle. Elles sont passées de 52 bits (sur certains processeurs) à 64 puis 80 bits sur des processeurs récents, cette évolution étant liée à l'accroissement de la taille des adresses physiques. Cela a bien sûr une implication sur l'écriture des programmes puisque les adresses contenues dans les instructions sont virtuelles : un changement de taille impliquerait, en théorie, une réécriture des codes en assembleur par un compilateur *ad hoc* capable d'adapter le nombre de bits des adresses dans les instructions. Heureusement, des mécanismes de compatibilité sont prévus dans les processeurs pour que l'on puisse utiliser d'anciennes adresses virtuelles de taille plus petite.

Note

Pour diminuer la taille des adresses virtuelles, les processeurs utilisent également le mécanisme de segmentation (voir section 3), qui transforme, avant toute référence mémoire, une adresse logique sur, par exemple, 32 bits, en une adresse virtuelle sur 52 bits. Ils ne manipulent plus alors que des adresses logiques de taille inférieure aux adresses virtuelles.

Table à plusieurs niveaux

La table des pages étant indexée par le numéro de page virtuelle, sa taille est directement fonction du nombre de bits correspondant dans une adresse virtuelle. Avec des pages de 4 Ko (et donc un déplacement exprimé sur 12 bits) et une adresse virtuelle sur 32 bits, le numéro de page virtuelle occupe 20 bits, soit 2^{20} pages possibles. Si chaque entrée de la table est de 3 octets, il faut 3 Mo pour la stocker, ce qui est déjà considérable. Si les adresses virtuelles étaient sur 64 bits et les pages de 4 Mo, il y aurait 2^{42} pages virtuelles, et la table occuperait (pour une entrée toujours formée de 3 octets) 12 To et serait impossible à stocker en mémoire principale ni même sur un disque. Pour ces cas extrêmes, nous verrons plus loin une autre méthode de traduction. Mais pour des cas intermédiaires (table trop grande pour être mise en entier en mémoire mais pouvant résider sur le disque), on utilise des numéros de pages virtuelles à plusieurs niveaux. Le principe est de les découper en deux (ou plus), les bits de poids fort donnant un numéro d'hyperpage, ceux de poids faible indiquant la page virtuelle dans cette hyperpage. Cela permet de n'avoir en mémoire que la table des hyperpages (de taille moins importante) et de charger en mémoire (à partir du disque), à la demande, la table des pages correspondante (voir figure 7.5).

Figure 7.5

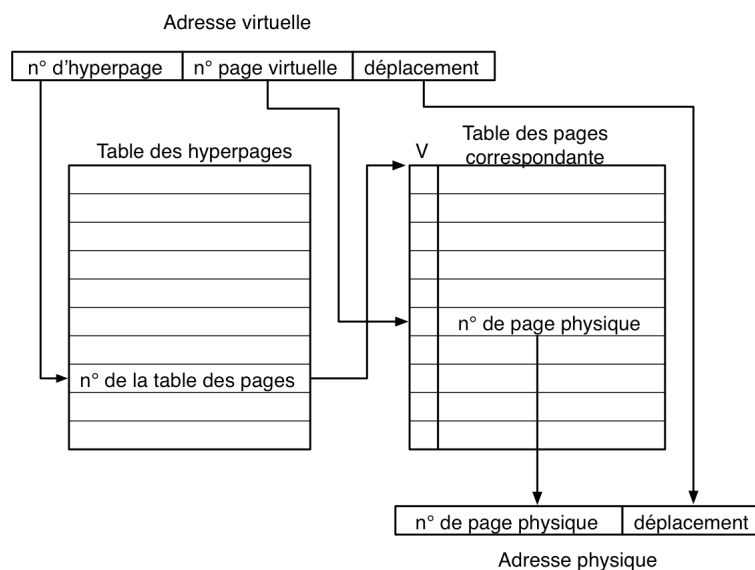


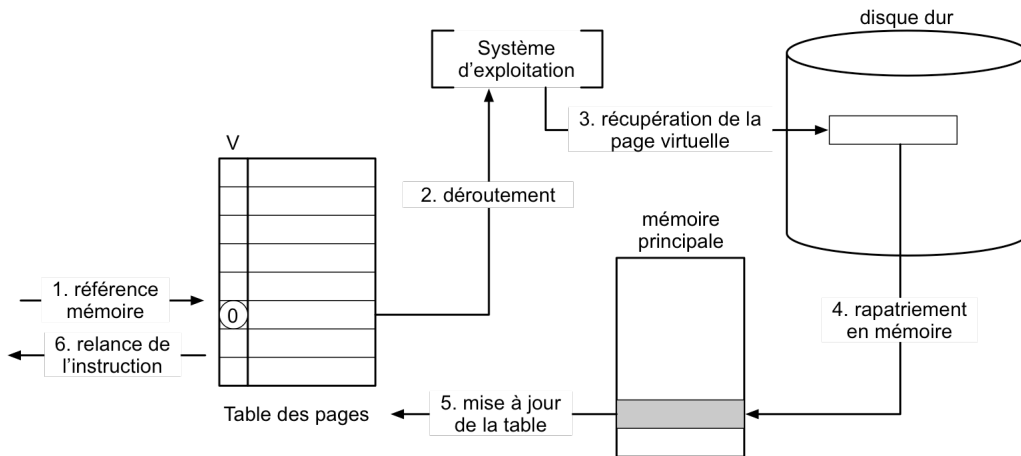
Table des pages à deux niveaux.

Malheureusement, cela implique également un ralentissement de l'accès mémoire puisque la MMU est obligée de consulter deux tables au lieu d'une avant de pouvoir effectuer la traduction de l'adresse virtuelle en adresse physique.

2.2 Algorithme de défaut de page

Lors d'une référence à une case mémoire (notée 1 à la figure 7.6), la MMU détecte parfois que la page virtuelle demandée par le processus n'est pas encore disponible en mémoire physique et qu'elle n'existe que sur le disque. En ce sens, chaque entrée est dotée d'un bit de validité de la table des pages. La MMU doit alors signaler un défaut de page (2), demander au système d'exploitation de récupérer sur le disque la page virtuelle (3) et de l'amener en mémoire physique (4), et relancer l'instruction ayant provoqué ce défaut de page (6).

Figure 7.6



Traitement d'un défaut de page.

La MMU génère une interruption de défaut de page (voir chapitre 8) pour donner la main au système d'exploitation. Le traitement est alors le suivant :

1. L'instruction en cours est arrêtée.
2. Une interruption de défaut de page est levée, donnant la main au système d'exploitation.
3. Le système suspend l'exécution du processus car celle-ci ne peut pas se poursuivre avant que la page virtuelle ne soit ramenée en mémoire physique.
4. Le système vérifie que la référence mémoire effectuée par le processus est valide.
5. Le système choisit un cadre de page, libre s'il en existe un, dans la mémoire physique pour rapatrier la page virtuelle.
6. Le système programme une opération de lecture sur le disque afin de récupérer la page virtuelle et de la mettre en mémoire physique.
7. Pendant l'opération de lecture, le système peut donner la main à un autre processus pour qu'il poursuive son exécution sur le processeur.
8. Une nouvelle interruption prévient le système de la fin du transfert de la page virtuelle en mémoire physique. Celui-ci peut alors mettre à jour la table des pages et débloquer le processus initial, qui reprendra son exécution au niveau de l'instruction fautive à son prochain passage sur le processeur.

Comme nous l'avons déjà signalé, les défauts de page sont coûteux en temps car ils nécessitent un accès disque, beaucoup plus lent que l'accès mémoire, et il faut limiter leurs occurrences lors de l'exécution d'un processus. Heureusement, un programme tend à regrouper ses références mémoire (grâce aux principes de localité, déjà évoqué pour la mémoire cache) et le processeur est rarement obligé de rapatrier une nouvelle page virtuelle à chaque instruction.

Remplacement de page

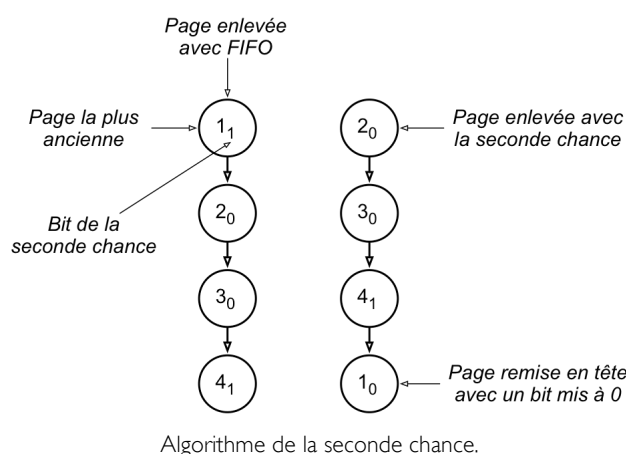
Quand plusieurs processus s'exécutent sur le processeur et occupent chacun une partie importante de leur espace virtuel, tous les cadres en mémoire physique, moins nombreux que les pages virtuelles, sont parfois occupés par des pages déjà ramenées. Au moment où le système doit en choisir un pour transférer depuis le disque une page virtuelle contenant une référence ayant provoqué un défaut de page, il doit alors expulser une page virtuelle de la mémoire physique pour la remplacer par la nouvelle (on dit que le système d'exploitation fait du *swap*). On retrouve la problématique associée au remplacement d'une ligne de la mémoire cache vue au chapitre 6. Comment choisir la page à extraire ? En théorie, il suffit d'en enlever une qui ne sera plus utilisée ou dont la prochaine utilisation est la plus lointaine. Mais, ici aussi, il est impossible de

connaître à l'avance la suite des références mémoire que vont effectuer les processus. Il faut donc faire une approximation sur les futurs besoins en pages mémoire et enlever celle qui a de bonnes chances de ne pas être demandée tout de suite. On retrouve alors des algorithmes similaires au remplacement des lignes du cache (voir chapitre 6).

- **Remplacement FIFO.** On extrait la page la plus ancienne de la mémoire physique.
- **Remplacement LRU.** On extrait la page dont la dernière utilisation est la plus ancienne.
- **Remplacement LFU.** On extrait la page qui a été la moins utilisée.
- **Remplacement MFU.** On extrait la page qui a été la plus utilisée.

L'algorithme FIFO risquant d'éliminer une page ancienne mais à laquelle on a régulièrement accédé, il existe une variante, appelée « remplacement de la deuxième chance ». Chaque page porte un bit de référence mis à 1 à chaque accès. Lorsqu'une page est sélectionnée comme étant la plus ancienne, si son bit de référence est à 0, elle est extraite ; s'il est à 1, on lui donne une seconde chance en sélectionnant la page suivante dans l'ordre d'arrivée (la première voit son bit remis à 0 et sa date d'arrivée modifiée). Cette technique, assez utilisée, a la simplicité de FIFO en respectant les pages couramment référencées (voir figure 7.7).

Figure 7.7



Algorithme de la seconde chance.

LRU et LFU sont assez efficaces mais nécessitent un dispositif matériel supplémentaire pour tenir à jour un compteur d'utilisation (LFU) ou une date de dernière utilisation (LRU). Cette mise à jour devant se faire à chaque accès mémoire sur la page (et pas à chaque défaut de page), le dispositif matériel correspondant doit être rapide pour ne pas pénaliser outre mesure les accès. Pour cette raison, ces deux algorithmes ne sont pas répandus.

L'algorithme MFU (Most Frequently Used) a été expérimenté avec l'idée qu'une page peu référencée vient probablement d'arriver en mémoire physique et va donc servir dans un avenir proche. Il vaut mieux la garder et éliminer celle qui a déjà beaucoup servi. Malheureusement, ce n'est pas efficace et cet algorithme est peu utilisé.

Bit de modification

Lors de l'élimination d'une page virtuelle située en mémoire physique pour libérer un cadre, le système doit vérifier si, depuis son arrivée en mémoire, elle n'a pas été modifiée (celle-ci peut par exemple contenir des données d'un programme). Si elle ne l'a pas été, on peut la remplacer directement par la nouvelle page. En cas de modification, il faut la réécrire sur le disque avant de faire place nette. À cet effet, la table des pages contient, pour chaque cadre, un bit de modification (*dirty bit*), qui indique si une écriture a eu lieu sur la page.

Cela ralentit encore plus le traitement d'un défaut de page car il faut éventuellement effectuer deux accès au disque, un pour réécrire la page modifiée et un second pour récupérer la nouvelle page. Pour cette raison, on peut encore améliorer l'algorithme de la deuxième chance en tenant également compte du bit de modification : une page à laquelle on a peu accédé (bit de référence à 0) et non modifiée est la candidate idéale au remplacement.

2.3 Mémoire virtuelle et exécution multitâche

Le coût de la mémoire RAM baisse constamment et il est rare, aujourd'hui, de ne pas avoir suffisamment de mémoire sur un ordinateur (on peut toujours en ajouter, l'investissement est rentable) pour exécuter l'ensemble des programmes souhaités. Plus de mémoire physique signifie moins de défauts de page et donc des accès mémoire plus rapides. Pour cette raison, le mécanisme de mémoire virtuelle, autrefois utilisé pour

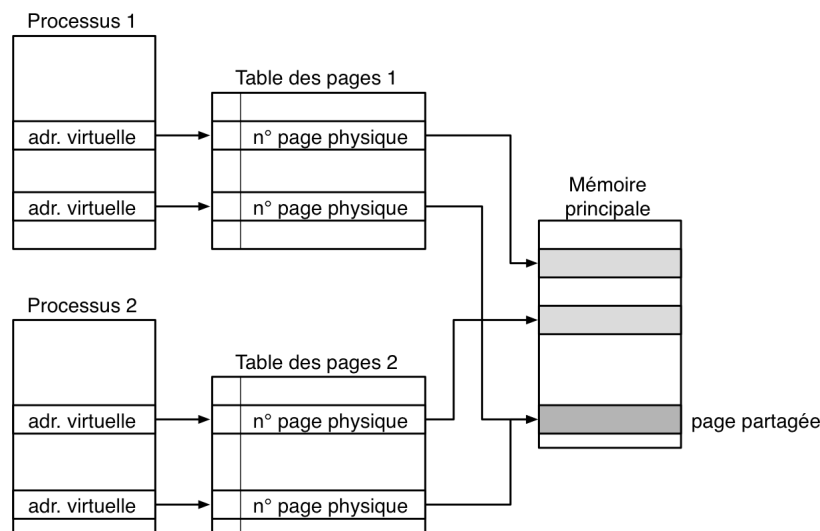
pallier une carence, est désormais mis en œuvre pour l'exécution multitâche, lorsque plusieurs processus s'exécutent simultanément sur le processeur.

En effet, il est intéressant de séparer les espaces mémoire des différents processus en cours d'exécution, pour les raisons suivantes :

- Un processus peut ainsi avoir un usage libre de son espace mémoire, indépendamment des autres processus, de la quantité de mémoire physique, et de son utilisation par l'ensemble des programmes en cours d'exécution.
- L'espace mémoire d'un processus est protégé des accès malveillants ou simplement erronés d'un autre programme qui s'exécute.

Chaque processus se voit attribuer un espace mémoire propre, couvrant l'intégralité des adresses qu'il est susceptible de générer dans les instructions. Il n'a ainsi aucun moyen de faire référence à une adresse mémoire d'un autre processus car il n'a accès qu'à des adresses virtuelles situées dans son propre espace (voir figure 7.8), sauf demande explicite de partage d'espace mémoire.

Figure 7.8



Espaces virtuels de plusieurs processus.

L'inconvénient de ce mécanisme est que les processus font référence aux mêmes adresses virtuelles, correspondant à des adresses physiques différentes. Le système d'exploitation ne peut donc pas se contenter de maintenir une table des pages : il en faut une par processus (voir figure 7.8). Cela complique le changement de contexte du processus lors d'une commutation (passage d'un processus à un autre) puisqu'il faut remplacer l'adresse de la table dans la MMU.

Deux processus ont aussi la possibilité de partager volontairement un espace mémoire via le système d'exploitation : ils peuvent voir chacun une de leurs pages virtuelles envoyée dans un cadre physique commun ; toute modification d'une information par l'un est répercutée dans l'adresse physique et donc vue par l'autre (voir figure 7.8).

2.4 Accélération de la traduction

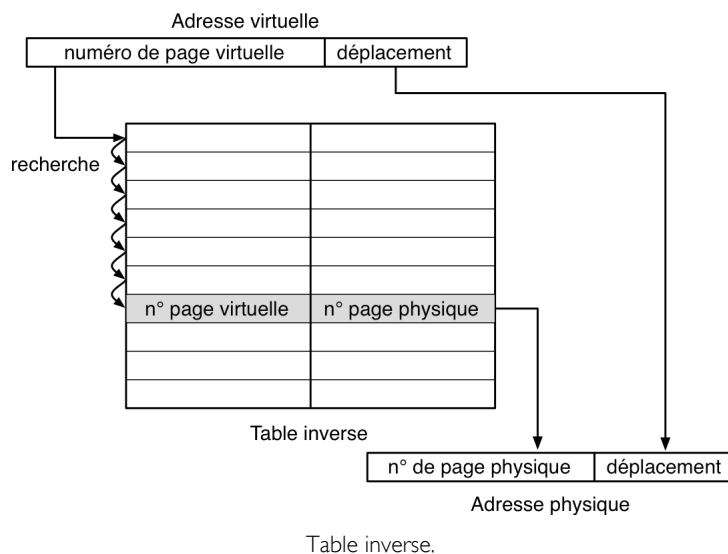
À chaque référence mémoire, la MMU doit traduire une adresse virtuelle en adresse réelle à l'aide de la table des pages. Ce mécanisme souffre de deux handicaps : la taille excessive de la table et la lenteur de la traduction liée à l'accès supplémentaire à cette table. On peut alors améliorer le fonctionnement en disposant autrement les informations.

Table inverse

La taille de la table est directement proportionnelle au nombre de pages virtuelles et donc à la longueur des adresses virtuelles. Lorsque celles-ci étaient écrites sur 32 bits, on pouvait stocker la table complète en mémoire (elle occupait quelques mégaoctets), mais cela est impossible depuis qu'elles tiennent sur 64 ou 80 bits. Il faut donc trouver un autre moyen de mémoriser la correspondance entre adresses virtuelle et physique. Pour ce faire, le plus simple est de conserver les couples directement dans une grande table. Comme il y en a au maximum autant que de pages physiques, on peut prévoir une table inverse qui donne, pour

chaque cadre de page physique, la page virtuelle qui y est stockée. La recherche à partir d'un numéro de page virtuelle est plus compliquée car, au lieu de se servir de ce numéro comme index, il faut parcourir toutes les entrées de la table et chercher si cette page virtuelle est dans un cadre physique (voir figure 7.9).

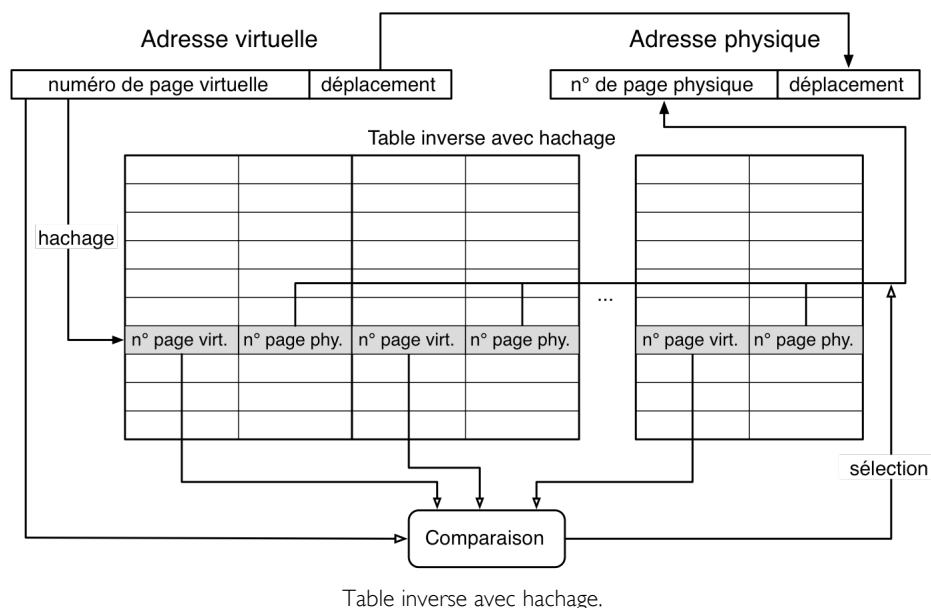
Figure 7.9



La taille de la table inverse est uniquement fonction du nombre de cadres de page présents en mémoire physique : avec une mémoire de 1 Go, des pages de 4 Ko, il y a 2^{18} cadres ; si chaque entrée est sur 16 octets (il y a plus d'informations à stocker), la table occupe 4 Mo en mémoire physique, soit une valeur acceptable.

Malheureusement, la recherche d'un numéro de page virtuelle parmi toutes les entrées ralentit fortement l'accès à l'adresse physique et la MMU utilise plus volontiers une fonction de hachage à partir de l'adresse virtuelle pour sélectionner directement une entrée où pourrait se trouver la correspondance. On calcule un index à partir du numéro de page virtuelle en prenant une fonction de hachage de ses bits. On sait que si la page virtuelle est en mémoire physique, la correspondance est mémorisée dans la ligne donnée par l'index. Mais comme pour toute fonction de hachage, des collisions peuvent survenir (plusieurs adresses virtuelles différentes donnant le même index). La table est alors prévue pour mémoriser plusieurs couples par ligne et une recherche parmi tous ces couples permet de trouver le numéro de la page virtuelle, si elle est présente en mémoire physique (voir figure 7.10).

Figure 7.10



Alors que dans une table inverse standard, tous les emplacements peuvent être utilisés pour mémoriser un couple, dans une table avec hachage, seule la ligne correspondant à la valeur de hachage est autorisée. Si

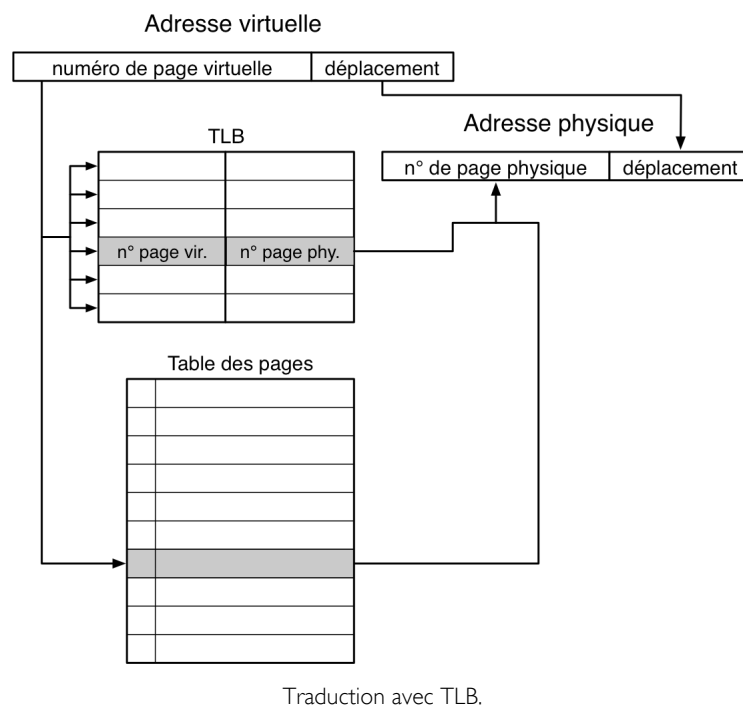
toutes les entrées de la ligne sont occupées par des couples et qu'une nouvelle correspondance entre adresses virtuelle et physique se présente, il faut retirer une entrée pour mettre la nouvelle (et augmenter les risques futurs de défaut de page), même si d'autres lignes sont disponibles. Il y a donc une moins bonne utilisation de la table, qui doit souvent être un peu plus grande qu'une table inverse standard si l'on veut garder des performances correctes.

Tampon de traduction anticipé

Quelle que soit la forme de la table des pages (plusieurs niveaux, inverse, avec hachage...), il faut la consulter à chaque accès mémoire pour trouver l'adresse physique correspondant à l'adresse virtuelle voulue. Cette table étant elle-même en mémoire physique (à une adresse réelle connue de la MMU), cela impose systématiquement un premier accès mémoire dans la table suivi d'un second pour trouver l'information, doublant le temps nécessaire. Pour éviter ce ralentissement, toutes les MMU incluent un mécanisme d'accélération des traductions destiné à éviter la consultation mémoire de la table. Il s'agit simplement de garder les derniers couples utilisés dans un tampon de traduction anticipé (appelé TLB, *Translation Lookaside Buffer*), formé d'une petite mémoire associative rapide, similaire à la mémoire cache. De 256 octets à quelques kilo-octets, elle est implantée directement dans la MMU (dans le processeur, et donc d'accès rapide).

Lors d'une présentation d'une adresse virtuelle à la MMU, celle-ci lance la consultation de la table et, en parallèle, effectue une recherche associative dans tout le TLB ou une partie seulement (suivant qu'il est complètement associatif ou associatif par ensemble, comme peut l'être la mémoire cache). Si l'on a accédé récemment à la page virtuelle (c'est souvent le cas conformément aux principes de localité), l'adresse physique correspondante a déjà été mémorisée dans le TLB et la MMU peut rapidement générer l'adresse réelle de l'information demandée, sans consulter la table des pages (voir figure 7.11). Au final, l'adresse physique est générée soit par le TLB, soit en cas d'échec par la table.

Figure 7.11



Une traduction d'adresse virtuelle peut donc se conclure de trois façons :

- La recherche dans le TLB est concluante (*TLB hit*) : la MMU génère rapidement l'adresse physique équivalente et la pénalité d'accès est minime.
- La TLB ne contient pas la correspondance (*TLB miss*, échec TLB), mais la page virtuelle est en mémoire physique : un accès à la table des pages permet d'avoir son adresse réelle et le temps d'accès est doublé.
- La page virtuelle n'est pas encore en mémoire physique (et il y a forcément un échec TLB) : la MMU soulève une interruption de défaut de page, le système d'exploitation prend le relais pour effectuer un accès disque afin de ramener la page virtuelle en mémoire, au prix d'une pénalité d'accès importante.

La présence d'une TLB complique la commutation lors d'un changement de processus. En effet, les correspondances entre adresses virtuelles et physiques ne sont valables que pour un processus alors que tous utilisent des adresses virtuelles identiques (rappelons que chaque processus possède son propre espace

virtuel). Il faut donc invalider toutes les entrées de le TLB lors d'un changement de processus ou associer un numéro de processus à chaque entrée de le TLB.

2.5 Mémoire virtuelle et mémoire cache

Comment la mémoire virtuelle interfère-t-elle avec le cache ? Ce dernier stocke les informations auxquelles on a accédé régulièrement et utilise l'adresse demandée pour récupérer la ligne voulue. En fait, il l'utilise une première fois pour sélectionner une ligne (cache direct) ou un ensemble (cache associatif par ensemble) et une seconde fois pour tester la présence de l'information en comparant les étiquettes.

Il faut donc savoir quelle adresse est utilisée : l'adresse virtuelle ou physique ? Quatre combinaisons sont possibles.

Adressage virtuel, étiquetage virtuel

Le cache peut utiliser l'adressage virtuel pour sélectionner la ligne ou l'ensemble. Cela permet un accès rapide puisqu'il n'y a pas de traduction d'adresse à effectuer avant la consultation du cache. Une comparaison des étiquettes virtuelles est immédiatement possible, mais cela a un inconvénient dans le contexte de l'exécution de plusieurs processus. Dans l'ensemble des espaces mémoire virtuels qui leur sont individuellement dédiés, les adresses virtuelles s'expriment de façon identique. On ne sait donc pas si une étiquette présente dans une ligne du cache correspond au processus en cours ou à l'un des programmes précédemment exécutés. Il faut alors soit mémoriser un numéro de processus avec la ligne, soit invalider toutes les entrées du cache lors d'un changement de processus, ce qui, dans les deux cas, complique le mécanisme.

Adressage physique, étiquetage physique

Il n'y a plus de problème d'ambiguïté entre processus, mais l'accès au cache nécessite systématiquement une traduction d'adresse et est par là même ralenti. Cette technique est souvent mise en œuvre dans les caches de niveau 2, plus lents (la pénalité supplémentaire de traduction n'est pas un facteur trop aggravant, proportionnellement parlant).

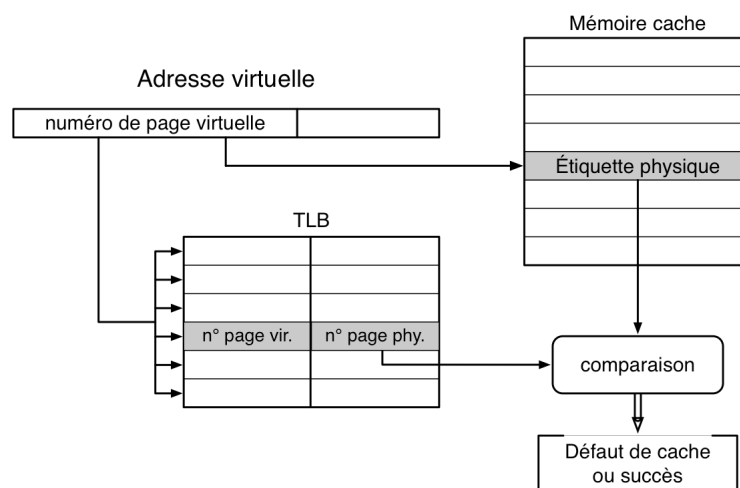
Adressage physique, étiquetage virtuel

Cette combinaison n'a aucun intérêt. L'adressage physique nécessite une traduction d'adresse avant l'accès et l'étiquetage virtuel provoque des confusions entre les informations de processus différents situées à de mêmes adresses virtuelles.

Adressage virtuel, étiquetage physique

Avec un adressage virtuel, l'accès au cache se fait sans traduction d'adresse (donc rapidement) et celle-ci peut même être lancée en parallèle. Si la correspondance est trouvée dans le TLB (donc, là encore, rapidement), on dispose en même temps de l'étiquette physique stockée dans le cache et de l'adresse physique souhaitée. Une comparaison indique s'il y a un défaut de cache ou non, sans perte de temps (voir figure 7.12). On a souvent ce cas de figure car, si l'information voulue est dans le cache, cela signifie qu'on y a accédé récemment et donc que la correspondance entre adresses virtuelles et physique est dans le TLB. Pour cette raison, ce mécanisme est souvent choisi pour les caches de niveau 1.

Figure 7.12



Cache à adressage virtuel et étiquetage physique.

2.6 Mémoire virtuelle et programmation

Les remarques énoncées à propos du cache et de la programmation valent pour la mémoire virtuelle : plus le principe de localité est vérifié dans le programme, plus nombreuses sont les références dans les mêmes pages et moins il y a de défauts de page et donc de ralentissements à l'exécution. Les accès aux variables ne doivent donc pas se faire au hasard mais, dans la mesure du possible, sur des cases mémoire voisines. Ainsi, on évite de passer son temps à enlever une page virtuelle de la mémoire physique pour aller en chercher une nouvelle sur le disque (voir exercice 4).

3. SEGMENTATION

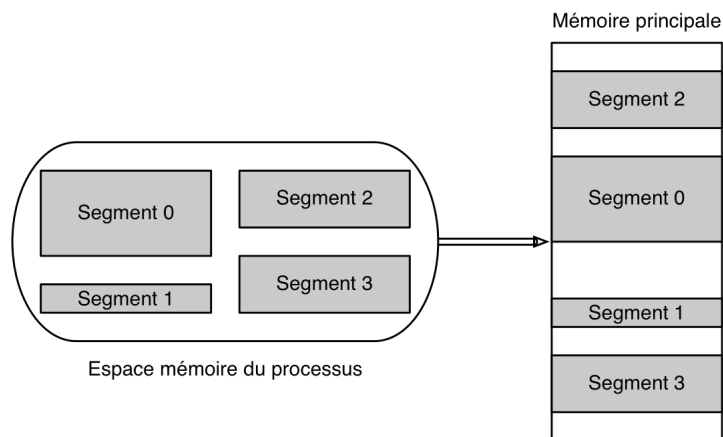
La pagination est un mécanisme matériel totalement transparent du point de vue du programmeur, qui n'a aucun moyen de savoir comment est organisé l'espace mémoire d'un processus. Au mieux, il peut détecter d'éventuels ralentissements dus à des défauts de page. Les processeurs ajoutent souvent un mécanisme de segmentation, qui est un découpage logique de l'espace mémoire d'un processus, accessible au programmeur et au compilateur.

3.1 Segmentation simple

Nous avons jusqu'à présent considéré l'espace mémoire d'un processus comme un espace linéaire d'adresses (virtuelles ou réelles), mais cela ne correspond pas vraiment à la vue logique que le programmeur a de l'utilisation de la mémoire. Un programme se compose de code, lui-même découpé en modules ou fonctions différentes, en une pile, en zones de stockage de variables globales, de constantes, etc. Ces zones sont indépendantes les unes des autres, de taille et d'usage différents. L'intérêt de pouvoir travailler de façon séparée sur chacune d'entre elles et de les référencer individuellement est évident. En ce sens, elles doivent avoir leur propre adresse mémoire.

L'espace mémoire d'un processus est ainsi découpé en segments, répertoriés par numéro et stockés n'importe où en mémoire principale (voir figure 7.13). Chaque référence mémoire effectuée par le processus (adresse d'une instruction lors d'un saut ou désignation d'une donnée) se fait de façon explicite via l'un des numéros attribués, ou de façon implicite si chaque segment est spécifique (par exemple, si un seul contient du code, l'adresse d'une instruction référence forcément une adresse de ce segment-là).

Figure 7.13



Segmentation de l'espace mémoire.

Les avantages de la segmentation et de la séparation des références mémoire sont multiples.

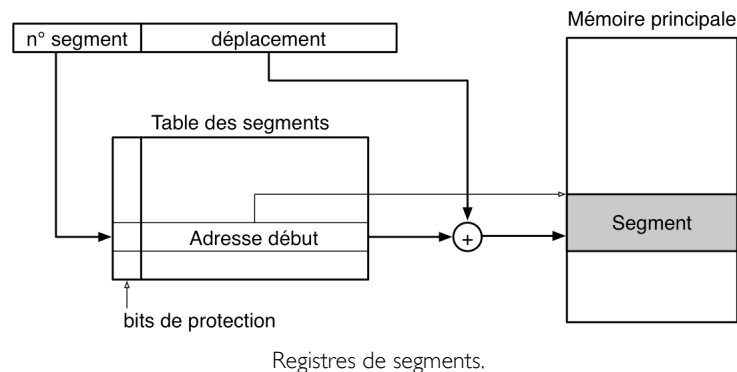
- Les segments peuvent s'accroître en fonction des besoins, sans que la taille de l'un conditionne la taille de l'autre.
- Les références mémoire des différents segments sont indépendantes. Ainsi, la croissance d'un segment (par modification du code par exemple) n'oblige pas à changer les références mémoire des autres, ce qui serait le cas si tout l'espace mémoire du processus était stocké d'un bloc.

- Il est possible de prévoir un mécanisme de protection des segments en fonction de leur utilisation, empêchant par exemple l'accès à un segment de code en écriture (car il est interdit de modifier des instructions) ou l'exécution d'une valeur numérique se trouvant dans un segment de données.

Adresses mémoire

Une adresse mémoire faisant explicitement référence à un segment est composée de deux parties : un numéro de segment suivi d'un déplacement à l'intérieur de celui-ci. L'adresse de début de chaque segment est mémorisée dans un registre, inscrit dans une table des segments. À partir du numéro de segment, la MMU récupère son adresse de début et l'additionne au déplacement (voir figure 7.14).

Figure 7.14



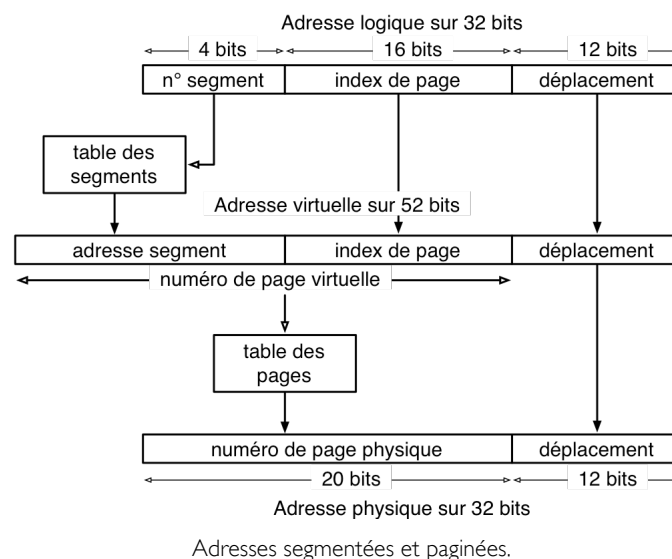
Registres de segments.

C'est le programmeur (s'il écrit en assembleur) ou le compilateur qui initialise la table des segments avec toutes les adresses de début de ces derniers.

3.2 Segmentation et pagination

Les deux techniques sont souvent combinées : l'espace mémoire d'un processus est tout d'abord réparti en plusieurs segments dans une grande mémoire virtuelle, puis celle-ci est paginée dans la mémoire physique. Ainsi, à la figure 7.15, un processus a un espace mémoire dont les adresses logiques s'expriment sur 32 bits. Les quatre bits de poids fort correspondent à un numéro de segment (il y en a donc seize différents) permettant de récupérer, via la table des segments, son adresse de début dans une grande mémoire virtuelle de 2^{52} octets. Cette mémoire virtuelle est ensuite paginée dans une mémoire physique de 4 Go maximum, dont les adresses réelles s'écrivent sur 32 bits.

Figure 7.15

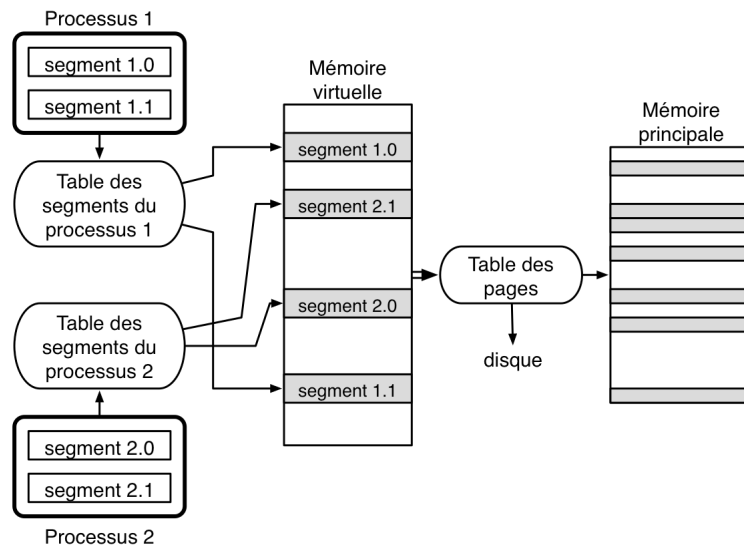


Adresses segmentées et paginées.

Au lieu d'avoir un espace virtuel par processus, les espaces mémoire logiques de chaque programme exécuté sont répartis dans une grande mémoire virtuelle paginée. On a donc, non pas une table des pages par processus, mais une seule table, car il n'y a qu'un grand espace virtuel. En revanche, on a une table des

segments par processus (qu'il faut changer à chaque commutation) pour qu'une même adresse logique, mais appartenant à deux processus différents, pointe sur deux adresses virtuelles distinctes (voir figure 7.16).

Figure 7.16



Segmentation et pagination.

RÉSUMÉ

La mémoire virtuelle permet d'offrir aux processus un espace mémoire propre, indépendant de la mémoire physique, plus grand et protégé des autres accès. La MMU, composant intégré au processeur, fait la correspondance entre adresses virtuelles utilisées par un processus et adresses physiques servant *in fine* à accéder à la mémoire réelle. Elle a à sa disposition une table des pages, qui stocke les couples, et un tampon de traduction rapide, qui accélère la recherche des correspondances en mémorisant les derniers couples utilisés. On combine souvent la segmentation, qui sépare en plusieurs entités logiques l'espace mémoire d'un processus, et la pagination pour répartir les espaces mémoire de plusieurs processus dans une grande mémoire virtuelle.

Problèmes et exercices

Comprendre le mécanisme de mémoire virtuelle implique de maîtriser le fonctionnement des algorithmes d'accès ou de remplacement des pages dans la table qui les contient. On peut ensuite évaluer ses performances en fonction du nombre de processus, du temps d'accès mémoire ou de la disposition des données dans un programme.

Exercice I : Manipuler la table de pages

La mémoire physique d'un ordinateur fait 1 Ko. Elle est découpée en quatre pages (de 0 à 3) de 256 octets. Un processus a un espace virtuel de 2 Ko (soit huit pages, de 0 à 7) et fait successivement référence aux adresses virtuelles 240, 546, 600, 547, 10, 1578, 2022, 1100, 56, 1300.

- 1) Indiquez, pour chaque adresse virtuelle, le numéro de la page virtuelle et la valeur du déplacement.
- 2) Indiquez, pour chaque référence, l'adresse physique à laquelle on accède réellement. Donnez la table des pages et la table inverse après chaque accès, et ce pour les algorithmes de remplacement FIFO, LRU et MFU. Combien y a-t-il de défauts de page pour chaque algorithme ? Les tables sont vides au départ.

- 1) Les pages étant de 256 octets, il suffit de prendre le quotient et le reste de la division de l'adresse par 256. Les adresses virtuelles sont sur 11 bits, dont 3 pour le numéro de page virtuelle et 8 pour le déplacement.

Tableau 7.1

Adresse virtuelle	Page virtuelle	Déplacement
240	0	240
546	2	34
600	2	88
547	2	35
10	0	10
1578	6	42
2022	7	230
1100	4	76
56	0	56
1300	5	20

Correspondances entre adresses virtuelle et physique

- 2) Pour l'algorithme FIFO, la table des pages après chaque accès est la suivante. Les références se font successivement de gauche à droite et chaque colonne donne la table des pages après la référence correspondante. Une croix (×) indique la disparition de la page virtuelle de la mémoire physique.

Tableau 7.2

	240	546	600	547	10	1578	2022	1100	56	1300
0	0	0	0	0	0	0	0	×	1	1
1										
2		1	1	1	1	1	1	1	×	
3										
4								0	0	0

5										2
6						2	2	2	2	×
7							3	3	3	3

Table des pages pour l'algorithme FIFO

Lors de l'accès à 240, situé dans la page virtuelle 0, on charge cette page dans le cadre physique 0. L'accès à 546 (page virtuelle 2) charge la page 2 dans le cadre physique 1. Les accès suivants ne chargent pas de nouvelle page, jusqu'à l'accès à 1578 (la page virtuelle 6 vient dans le cadre 2) et à 2022 (page virtuelle 7 dans le cadre 3). Lors de l'accès à 1100, on atteint une nouvelle page virtuelle (page 4). Il faut donc expulser une ancienne page de la mémoire physique car il n'y a plus de cadre de libre : la plus ancienne (algorithme FIFO) est la page 0, située dans le cadre 0. L'accès à 56 provoque l'expulsion de la page 2, remplacée dans le cadre 1 par la page 0, et l'accès à 1300 fait de même avec la page 6, remplacée par la page 5 dans le cadre 2.

Pour l'algorithme FIFO, la table inverse après chaque accès est la suivante :

Tableau 7.3

	240	546	600	547	10	1578	2022	1100	56	1300
0	0	0	0	0	0	0	0	4	4	4
1		2	2	2	2	2	2	2	0	0
2						6	6	6	6	5
3							7	7	7	7

Table inverse pour l'algorithme FIFO

Voici les adresses physiques auxquelles on a réellement accédé :

Tableau 7.4

	240	546	600	547	10	1578	2022	1100	56	1300
Adr.	240	290	344	291	10	554	998	76	312	532

Adresses physiques pour l'algorithme FIFO

Il y a sept défauts de page au total. Seules les références 600, 547 et 10 n'en provoquent pas.

Pour l'algorithme LRU, la table des pages après chaque accès est la suivante. Les références se font successivement de gauche à droite et chaque colonne donne la table des pages après la référence correspondante. Une croix (×) indique la disparition de la page virtuelle de la mémoire physique.

Tableau 7.5

	240	546	600	547	10	1578	2022	1100	56	1300
0	0	0	0	0	0	0	0	0	0	0
1										
2		1	1	1	1	1	1	×		
3										
4								1	1	1
5										2
6						2	2	2	2	×
7							3	3	3	3

Table des pages pour l'algorithme LRU

Pour l'algorithme LRU, la table inverse après chaque accès est la suivante :

Tableau 7.6

	240	546	600	547	10	1578	2022	1100	56	1300
0	0	0	0	0	0	0	0	0	0	0
1		2	2	2	2	2	2	4	4	4
2						6	6	6	6	5
3							7	7	7	7

Table inverse pour l'algorithme LRU

Voici les adresses physiques auxquelles on a réellement accédé :

Tableau 7.7

	240	546	600	547	10	1578	2022	1100	56	1300
Adr.	240	290	344	291	10	554	998	332	56	532

Adresses physiques pour l'algorithme LRU

Il y a six défauts de page au total. Seules les références 600, 547, 10 et 56 n'en provoquent pas.

Pour l'algorithme MFU, la table des pages après chaque accès est la suivante. Les références se font successivement de gauche à droite et chaque colonne donne la table des pages après la référence correspondante. Une croix (×) indique la disparition de la page virtuelle de la mémoire physique.

Tableau 7.8

	240	546	600	547	10	1578	2022	1100	56	1300
0	0	0	0	0	0	0	0	0	0	×
1										
2		1	1	1	1	1	1	×		
3										
4								1	1	1
5										0
6						2	2	2	2	2
7							3	3	3	3

Table des pages pour l'algorithme MFU

Pour l'algorithme MFU, la table inverse après chaque accès est la suivante :

Tableau 7.9

	240	546	600	547	10	1578	2022	1100	56	1300
0	0	0	0	0	0	0	0	0	0	5
1		2	2	2	2	2	2	4	4	4
2						6	6	6	6	6
3							7	7	7	7

Table inverse pour l'algorithme MFU

Voici les adresses physiques auxquelles on a réellement accédé :

Tableau 7.10

	240	546	600	547	10	1578	2022	1100	56	1300
Adr.	240	290	344	291	10	554	998	332	56	20

Adresses physiques pour l'algorithme MFU

Il y a six défauts de page au total. Seules les références 600, 547, 10 et 56 n'en provoquent pas.

Exercice 2 : Saturer le processeur

Un ordinateur a une mémoire principale de cinq pages (de 0 à 4) et offre aux processus une mémoire virtuelle de dix pages, gérée par l'algorithme LRU.

- 1) Cinq processus sont lancés. Celui qui porte le numéro 0 travaille uniquement avec la page 0, le 1 avec la page 1, et ainsi de suite. Combien y a-t-il de défauts de page ?
- 2) Un sixième processus de caractéristiques identiques est lancé. Combien y a-t-il de défauts de page ? Qu'en pensez-vous ?

1) Il y a un défaut de page la première fois qu'un processus référence une page. Ensuite, elles sont toutes en mémoire principale. Il y a donc cinq défauts de page pour l'exécution de tous les processus.

2) Les références se font dans l'ordre 0, 1, 2, 3, 4, puis 5 qui enlève la page 0, puis 0 qui enlève la 1, et ainsi de suite. Il y a maintenant un défaut de page à chaque référence ! Le système s'écroule : il passe son temps à amener des pages du disque en mémoire principale à chaque référence mémoire, ce qui ralentit énormément l'exécution des processus.

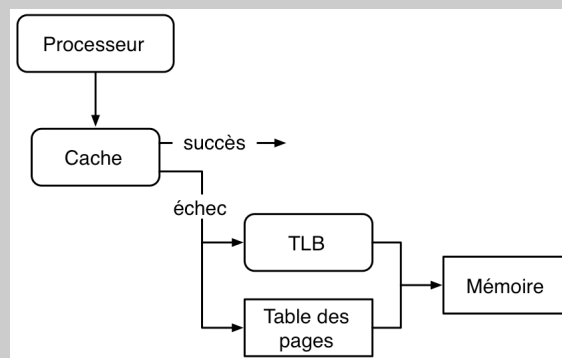
Le phénomène risque même de s'amplifier. Lorsque le système détecte un défaut de page, il lance la récupération de la page virtuelle sur le disque et essaie d'exécuter un autre processus. Celui-ci va également provoquer un défaut de page. Rapidement, tous les processus sont alors en défaut de page et le taux d'utilisation du processeur baisse. Il est alors possible que le système d'exploitation essaie de lancer d'autres processus en attente pour augmenter l'utilisation du processeur, provoquant encore plus de défauts de page. L'écroulement s'amplifie et le système passe la plus grande partie de son temps à commuter les processus et à attendre les pages virtuelles depuis le disque, sans que le processeur n'arrive à exécuter quoi que ce soit. Une solution est alors de diminuer les défauts de page en tuant ou en suspendant certains processus, jusqu'à arriver à une utilisation correcte de la mémoire principale.

Exercice 3 : Comparer cache et mémoire virtuelle

Un ordinateur possède une mémoire principale ayant un temps d'accès de 100 ns.

- 1) On installe sur le système un mécanisme de mémoire virtuelle avec une table des pages stockée en mémoire principale et une TLB pour accélérer les traductions. Le temps d'accès au TLB est de 10 ns et, lors d'une traduction d'adresse, la recherche dans ce tampon se fait en parallèle avec la récupération de la table. Le taux de succès du TLB est de 95 % (95 % des adresses virtuelles ont leur adresse physique correspondante dans le TLB ; pour les 5 % qui ne l'ont pas, il faut consulter la table). Quel est le temps moyen d'accès à une information en mémoire ?
- 2) L'ordinateur n'a plus de mécanisme de mémoire virtuelle. Il utilise un cache auquel on accède en 10 ns avant la consultation de la mémoire principale (on lance une requête dans le cache ; si celle-ci est infructueuse, on adresse ensuite la mémoire principale). Le taux de défauts de cache est de 10 %. Quel est le temps moyen d'accès à une information en mémoire ?
- 3) En plus du cache, on ajoute le mécanisme de mémoire virtuelle décrit à la première question. L'accès à une information est décrit à la figure 7.17. On consulte d'abord le cache et, en cas d'échec, on lance en parallèle une requête TLB et une consultation de la table des pages pour avoir l'adresse physique de l'information voulue.

Figure 7.17



Cache et mémoire virtuelle.

Le cache est indexé grâce à l'adresse virtuelle (il n'y a donc pas de traduction avant) et les étiquettes sont virtuelles (aucune traduction n'est nécessaire pour détecter si elles sont égales). On suppose de plus qu'il n'y a pas d'ambiguïté d'adresses entre processus. Quel est le temps moyen d'accès à une information en mémoire ?

1) Lors de 95 % des accès, le TLB fournit 10 ns en la correspondance entre adresses virtuelle et physique. Les 5 % restants obligent à un accès mémoire (en 100 ns car l'accès s'effectue en parallèle avec la recherche dans le TLB) pour consulter la table des pages. Dans les deux cas, on a besoin d'un accès mémoire supplémentaire pour récupérer effectivement l'information. On a donc un temps moyen de :

$$T = 0,95 \times 110 + 0,05 \times 200 = 114,5$$

Soit une dégradation des performances de seulement 14,5 %. Sans le TLB, le temps d'accès serait de 200 ns, soit une dégradation de 100 % ! Pour améliorer encore plus le temps moyen, on peut soit accélérer la consultation du TLB, soit augmenter le taux de succès, par exemple en agrandissant la capacité du tampon.

2) Le temps moyen est de 10 ns pour la consultation du cache, plus éventuellement (dans 10 % des cas) 100 ns pour adresser la mémoire principale. Soit un temps moyen de 20 ns :

$$T = 10 + 0,1 \times 100 = 20$$

3) Après la consultation du cache en 10 ns, il faut dans 10 % des cas un accès à la mémoire virtuelle et donc un temps moyen de 114,5 ns. Soit au final un temps moyen de 21,45 ns :

$$T = 10 + 0,1 \times 114,5 = 21,45$$

La dégradation due à la mémoire virtuelle est encore moins importante (7,25 %) que par rapport à la question 1 en raison du temps incompressible d'accès au cache.

Ces résultats confirment l'importance majeure du cache dans l'accélération des transferts de données entre la mémoire et le processeur. La mémoire virtuelle, avec une TLB, ne dégrade pas trop les performances, à condition que les défauts de page (non pris en compte ici) soient peu fréquents car ils sont pénalisants.

Exercice 4 : Multiplier des matrices en mémoire

Considérons le programme suivant :

Listing 7.1

```
int X[1024][1024];
int Y[1024][1024];
int Z[1024][1024];

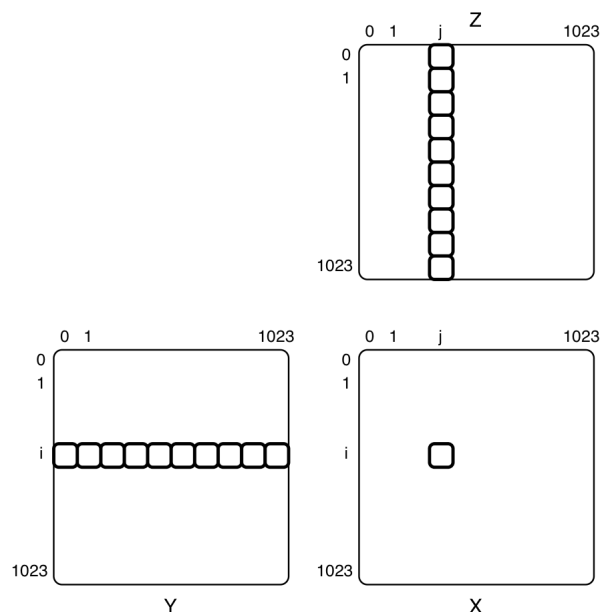
main () {
    int i, j, k;
    for (i=0; i<1024; i++)
        for (j=0; j<1024; j++)
            for (k=0; k<1024; k++)
                X[i][j] += Y[i][k] * Z[k][j];
}
```

Multiplication de matrice

- 1) Chaque entier est stocké sur 4 octets et les pages virtuelles sont de 4 Ko. Combien de pages virtuelles faut-il pour stocker les trois tableaux ?
- 2) Les tableaux sont stockés ligne par ligne (la ligne étant le premier indice, la colonne le second). À quelles pages accède-t-on lors de l'exécution de la boucle intérieure (i et j fixés) ? Combien faut-il de cadres de page pour n'avoir aucun défaut de page (après la première référence) lors de l'exécution de cette boucle intérieure ?
- 3) Répondez à la question précédente en supposant que les boucles sont inversées, dans l'ordre k-j-i. Y a-t-il un ordre encore meilleur ?
- 4) On suppose que Z est une matrice constante qui intervient comme membre droit lors de multiplications de matrices. Peut-on améliorer son stockage pour diminuer le nombre de cadres nécessaires si l'on ne veut pas changer l'ordre des boucles i-j-k ?

- 1) Chaque ligne contient 1 024 entiers, donc 4 096 octets, soit 4 Ko, exactement une page virtuelle. Il faut donc 1 024 pages par matrice, soit 3 072 pages virtuelles au total.
- 2) i et j étant fixés, on accède, lors de l'exécution de la boucle intérieure, à un élément de la matrice X, à une ligne unique de Y (la ligne i) et à une colonne unique de Z (celle de j), comme indiqué à la figure 7.18.

Figure 7.18

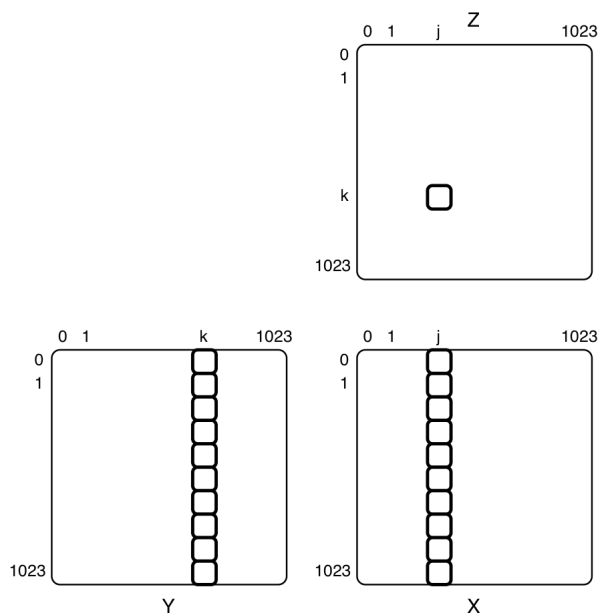


Multiplication de matrices – I.

On a donc besoin d'une page pour l'élément de X, d'une page pour la ligne de Y, et des 1 024 pages de Z, soit 1 026 pages au total.

3) k et j sont fixés et i varie. Lors du calcul partiel de x , effectué dans la boucle intérieure, le programme accède donc à une colonne de X , à une colonne de Y et à un élément de Z (voir figure 7.19).

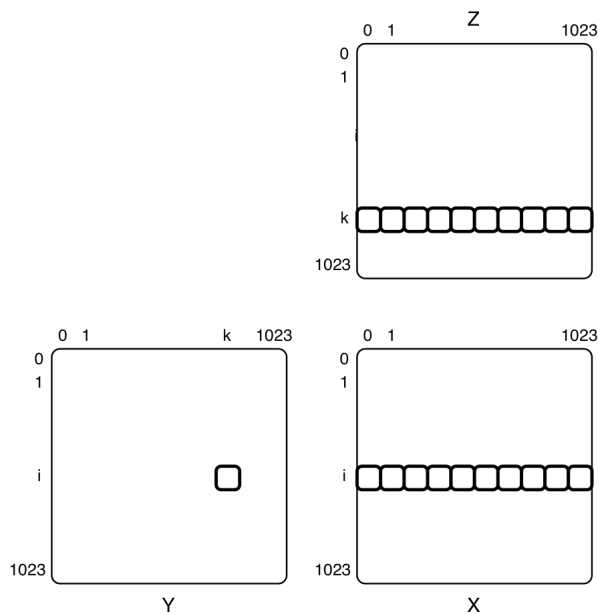
Figure 7.19



Multiplication de matrices – 2.

Cela nécessite l'accès à toutes les pages de x , à toutes les pages de Y et à une page de Z , soit 2 049 pages au total. On voit que l'important est de garder une ligne fixe et de faire varier l'indice des colonnes pour toujours accéder à la même page. Il vaut alors mieux garder i et k fixes et faire varier j dans la boucle intérieure, c'est-à-dire prendre l'ordre i - k - j (voir figure 7.20).

Figure 7.20



Multiplication de matrices – 3.

Lors du calcul partiel de x , effectué dans la boucle intérieure, on référence une page de x , une page de Y et une page de Z à chaque exécution de la boucle intérieure, soit seulement trois pages. Si la mémoire physique est limitée, cette dernière solution est beaucoup plus intéressante car elle provoque moins de défauts de page que les autres.

4) Si l'on garde l'ordre i-j-k, l'accès à la matrice Z se fait par colonne, ce qui n'est pas efficace. Pour y accéder ligne par ligne, il suffit de la transposer (c'est-à-dire d'inverser lignes et colonnes) et d'inverser l'ordre des indices dans le calcul : $X[i][j] += Y[i][k] * Z[j][k];$.

Exercice 5 : Comparer les algorithmes de remplacement

Notre ordinateur est équipé d'une mémoire physique découpée en trois cadres de page et d'une mémoire virtuelle comportant quatre pages (0 à 3).

1) Proposez une suite de douze références de pages virtuelles provoquant moins de défauts de page pour l'algorithme FIFO que pour l'algorithme LRU.

2) Proposez une suite de douze références de pages virtuelles provoquant moins de défauts de page pour l'algorithme LRU que pour l'algorithme MFU.

3) Proposez une suite de douze références de pages virtuelles provoquant moins de défauts de page pour l'algorithme MFU que pour l'algorithme FIFO.

1) Il suffit de réutiliser une page : LRU supprime des pages anciennes qui vont être réutilisées plus tard. Le tableau ci-après donne une suite de références successives (de gauche à droite) provoquant plus de défauts de page pour LRU que pour FIFO. Une croix (x) indique un défaut de page et la page éjectée de la mémoire physique est mentionnée entre parenthèses.

Tableau 7.11

Réf.	0	1	2	0	3	1	2	3	0	3	1	2
FIFO	x	x	x		x (0)				x (1)		x (2)	x (3)
LRU	x	x	x		x (1)	x (2)	x (0)		x (1)		x (2)	x (3)

FIFO plus efficace que LRU

2) Il suffit de réutiliser de nombreuses fois deux pages : MFU les enlève à tour de rôle de la table alors que LRU les garde. Le tableau ci-après donne une suite de références successives (de gauche à droite) provoquant plus de défauts de page pour MFU que pour LRU. Une croix (x) indique un défaut de page et la page éjectée de la mémoire physique est mentionnée entre parenthèses.

Tableau 7.12

Réf.	0	1	2	2	3	3	2	2	3	3	2	2
LRU	x	x	x		x (0)							
MFU	x	x	x		x (2)		x (3)		x (2)		x (3)	

LRU plus efficace que MFU

3) On réutilise de temps en temps de vieilles pages : FIFO les enlève de la table alors que MFU les conserve. Le tableau ci-après donne une suite de références successives (de gauche à droite) provoquant plus de défauts de page pour FIFO que pour MFU. Une croix (x) indique un défaut de page et la page éjectée de la mémoire physique est mentionnée entre parenthèses.

Tableau 7.13

Réf.	0	1	2	2	3	3	0	0	1	2	2	3
MFU	x	x	x		x (2)					x (0)		
FIFO	x	x	x		x (0)		x (1)		x (2)	x (3)		x (0)

MFU plus efficace que FIFO

En conclusion, aucun algorithme de remplacement n'est plus efficace qu'un autre dans l'absolu, mais leurs performances dépendent de l'organisation du programme et de la suite de références de pages virtuelles qu'il effectue. On ne peut parler que d'efficacité statistique, sur des programmes classiques ou de tests.

Exercice 6 : Anomalie de Belady

Un ordinateur possède une mémoire virtuelle de cinq pages (0 à 4) et un programme effectue une suite de références à des pages virtuelles comme suit : 0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4. L'algorithme de remplacement utilisé est FIFO.

Combien y a-t-il de défauts de page si la mémoire physique possède trois cadres ? Et si elle en possède quatre ?

Le tableau ci-après donne les défauts de page s'il y a trois ou quatre cadres en mémoire physique, l'algorithme de remplacement utilisé étant FIFO.

Tableau 7.14

Réf.	0	1	2	3	0	1	4	0	1	2	3	4
3 cadres	×	×	×	×	×	×	×			×	×	
4 cadres	×	×	×	×			×	×	×	×	×	×

Défauts de page avec trois ou quatre cadres

Il y a neuf défauts de page si la mémoire est formée de trois cadres alors qu'il y en a un de plus avec quatre cadres ! Autrement dit, augmenter la quantité de mémoire physique ne garantit pas d'avoir moins de défauts de page. Ce phénomène est appelé « anomalie de Belady », du nom de son découvreur en 1969. Il est lié à l'algorithme FIFO. D'autres algorithmes, comme LRU, ne présentent pas d'anomalie équivalente.

Exercice 7 : Gérer les défauts de page

Dans un système à mémoire virtuelle, à chaque référence mémoire issue du processeur, la MMU vérifie que la page correspondante est bien en mémoire physique. Si ce n'est pas le cas, elle génère un déroutement de défaut de page, qui provoque l'exécution de la fonction système chargée d'aller chercher la page manquante sur le disque. Quelle précaution faut-il prendre au niveau de l'implémentation de cette partie du système gérant la mémoire virtuelle pour éviter un blocage de cet algorithme lors d'un défaut de page ?

La fonction de traitement est composée d'instructions effectuant des références mémoire. Il faut être sûr que ces références, ainsi que les instructions en question, sont bien en mémoire physique car s'il y a un défaut de page durant le traitement, l'algorithme est bloqué. Le système doit donc verrouiller les pages contenant cette fonction pour qu'elles ne soient jamais enlevées de la mémoire et remises sur le disque.

Chapitre 8

Entrées/sorties

Le processeur serait inutile sans moyen de communication. Les périphériques constituent ainsi l'interface entre l'ordinateur et le monde extérieur. Des bus de communication standard véhiculant des signaux d'interruption (pour prévenir le processeur d'un événement imprévu, externe ou interne), des données et des adresses, permettent aux périphériques de communiquer avec le processeur. Ces bus sont connectés à des contrôleurs d'entrées/sorties, dirigés par des programmes spécifiques chargés de faire la traduction entre information côté processeur et transfert côté périphériques.

Parmi les périphériques les plus courants se trouvent les systèmes de stockage dont nous donnerons une description succincte en explicitant les technologies utilisées.

Entrées/sorties

1. Bus	186
2. Interruptions.....	189
3. Gestion des entrées/sorties.....	192
4. Technologies de stockage.....	196

Problèmes et exercices

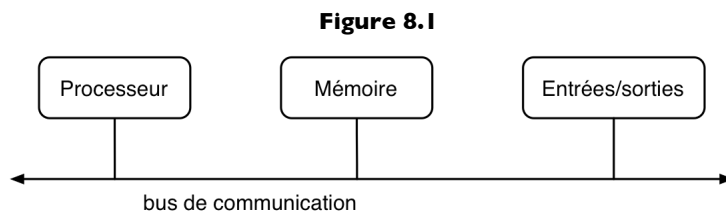
1. Débit du bus	199
2. Exception de défaut de page.....	199
3. Priorités des interruptions	200
4. Vecteur d'interruption	200
5. Interrogation et interruption.....	200
6. Intérêt de l'interrogation	201
7. DMA et interrogation	201
8. Capacité d'une disquette	202

I. Bus

Lorsque plusieurs composants de l'ordinateur (processeur, cache, mémoire, périphérique d'entrées/sorties...) veulent échanger de l'information, ils utilisent un bus, c'est-à-dire un ensemble de fils reliant plusieurs circuits entre eux, dont la structure et les spécifications fonctionnelles sont bien définies.

I.1 Organisation du bus

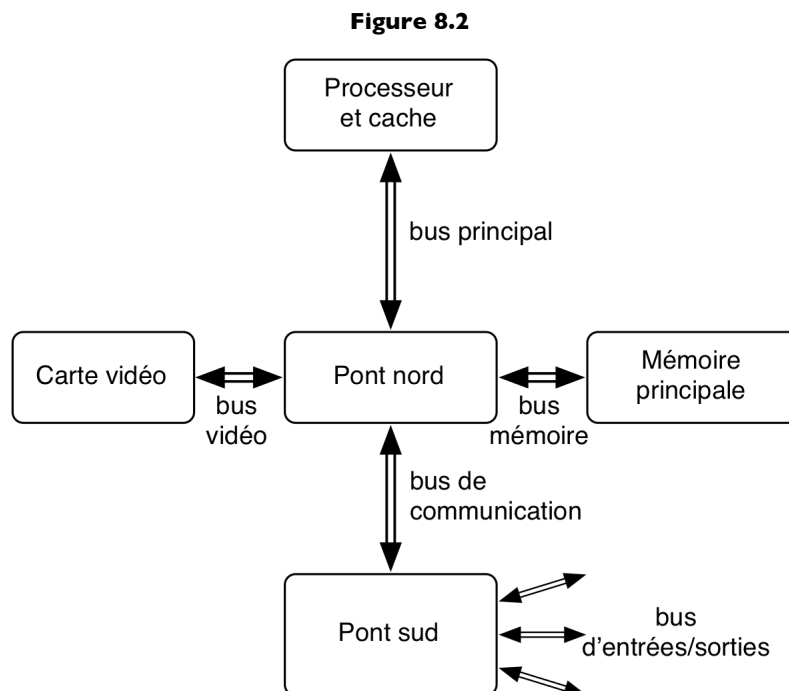
La présentation la plus simple de l'architecture de von Neumann (voir chapitre 3) est reproduite à la figure 8.1. On a un seul bus de communication entre les différents composants.



Architecture de von Neumann.

Si les premiers micro-ordinateurs se contentaient d'un seul bus de communication, les machines actuelles en demandent davantage : un processeur (tout comme la mémoire) travaille plus rapidement que les périphériques d'entrées/sorties (voir chapitre 5) et il serait contraint à des périodes d'inactivité lors de l'occupation du bus, si celui-ci était unique, par les unités les plus lentes.

On détermine les caractéristiques des bus de communication en fonction des unités connectées : rapidité et débit sont privilégiés lorsqu'il s'agit du processeur, alors que la standardisation est le critère essentiel des bus d'entrées/sorties. La figure 8.2 (presque identique à la figure 3.2 du chapitre 3) montre une architecture d'un ordinateur récent, avec ses différents bus individualisés.



Architecture d'un ordinateur récent.

Relié au processeur, le bus principal, aussi appelé « bus système » ou FSB (*Front Side Bus*), est l'unique moyen de communication du processeur. Il doit être le plus rapide et le plus large possible. Il est relié à un circuit intégré spécialisé, le pont nord (*northbridge*), chargé de répartir les informations entre processeur, mémoire, carte vidéo et le reste des composants via un deuxième circuit, le pont sud (*southbridge*), en charge des différentes entrées/sorties. Cette paire de circuits (regroupée en un contrôleur système à la figure 3.2 du chapitre 3) se

trouve parfois dans un même boîtier et s'appelle un « chipset » (ensemble de circuits). La tendance actuelle est à l'intégration de plus en plus poussée de ces composants sur la puce du processeur. On a ainsi une diminution de l'encombrement et de la consommation énergétique (points importants pour une utilisation nomade), associée à une amélioration des performances liée à la proximité géographique des circuits.

La plupart des bus se caractérisent par une structure en trois parties :

- **Le bus d'adresses.** Cet ensemble de fils véhicule les adresses permettant de désigner un élément ou une case mémoire. Sa largeur (son nombre de bits) donne une indication sur l'espace mémoire adressable. De 16 bits à la fin des années 70 (soit 64 Ko adressables), il est passé à 20 bits puis à 32 bits. Sur les processeurs récents, suivant les modèles, il atteint les 36 à 42 bits, rarement plus. Il véhicule des adresses physiques pour communiquer avec la mémoire principale. Sa taille est donc indépendante des adresses virtuelles utilisées par un processus (voir chapitre 7). Il est souvent unidirectionnel car c'est le processeur qui indique une adresse aux autres composants.
- **Le bus de données.** Cet ensemble de fils permet de transférer les informations (données et instructions) d'un composant à un autre. Sa largeur influence directement le débit d'informations disponible sur le bus. S'il est de 8 bits, il permet d'échanger 1 octet à la fois, soit huit fois moins de données qu'un bus de 64 bits (valeur standard actuelle) fonctionnant à la même vitesse.
- **Le bus de contrôle.** Il transporte les signaux de contrôle indiquant le type d'opération désiré (lecture ou écriture mémoire, accès vidéo, opération d'entrées/sorties...), dont les éléments (adresse, donnée) sont disponibles sur les deux autres bus. Il est bidirectionnel de sorte que les composants de l'ordinateur qu'il relie peuvent communiquer si besoin.

Sur certains bus, les parties adresses et données sont multiplexées sur des mêmes fils. Cela signifie qu'il n'existe qu'un seul bus adresses/données sur lequel circulent alternativement (en fonction des demandes bien sûr) signaux d'adresses et signaux de données. Cela permet d'économiser des fils, mais complique la gestion des commandes.

Arbitrage du bus

Lorsque plusieurs éléments sont connectés à un même bus et que plusieurs d'entre eux demandent simultanément à le contrôler pour émettre une requête, il risque d'y avoir un conflit d'accès, d'où la nécessité d'avoir un arbitrage du bus, pouvant être centralisé ou décentralisé.

S'il est centralisé, l'un des composants est désigné comme arbitre. Les autres effectuent une demande d'accès au bus *via* une ligne spécialisée du bus de contrôle et l'arbitre accède à la requête de l'un d'entre eux en fonction de priorités établies à l'avance (ou tournantes). Ce système simple surcharge l'arbitre ou oblige à ajouter un circuit dédié à l'arbitrage.

Dans le cadre d'un arbitrage décentralisé, aucun arbitre n'est désigné mais les différents éléments discutent entre eux pour attribuer le bus. Lorsque l'un d'entre eux veut l'obtenir, il émet une requête sur des lignes spéciales du bus de contrôle et compare son niveau de priorité avec celui des autres éléments ayant également envoyé une requête d'accès. Le plus prioritaire sait alors que le bus lui est attribué et qu'il peut l'occuper.

I.2 Horloge du bus

Le paramètre temporel est fondamental lors d'un transfert d'informations. Les composants de l'ordinateur travaillant à des rythmes différents, il est indispensable de pouvoir les synchroniser : à quel moment une commande est-elle envoyée ? Quand une donnée est-elle disponible ? Pour ce faire, deux techniques sont utilisées : la synchronisation par signaux de validation et d'acquiescement, mise en œuvre sur les bus asynchrones, et la transmission en parallèle d'une horloge, de l'information et des commandes, mise en œuvre sur les bus synchrones.

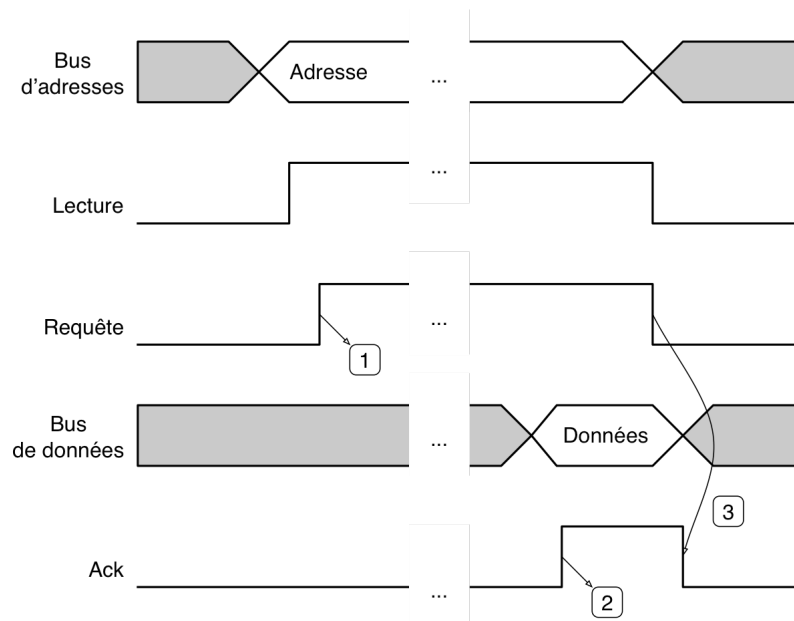
Bus asynchrone

Le bus asynchrone relie deux composants sans contraintes temporelles fortes. Chacun peut travailler à son propre rythme. Un mécanisme de « poignée de main » (*handshaking*) par signaux de requête et d'acquiescement permet de signaler la présence d'une demande ou d'une réponse. La figure 8.3 donne un exemple de lecture mémoire effectuée par le processeur.

Après avoir positionné l'adresse voulue sur le bus d'adresses, le processeur indique son souhait d'effectuer une lecture en validant la commande de lecture sur le bus de contrôle. La mise à l'état haut de la ligne de requête sur le bus de contrôle (1) est le signal indiquant à la mémoire qu'un accès est demandé. Au bout d'un certain temps (quelconque puisqu'il n'y a aucune référence temporelle globale), l'information devient disponible sur le bus de données et le boîtier mémoire le signale en validant une ligne d'acquiescement (2). Celle-ci permet au processeur de savoir qu'il peut récupérer l'information sur le bus. Cela fait, il enlève l'adresse du bus

d'adresses, remet le signal de lecture au repos et désactive le signal de requête, prévenant ainsi le boîtier mémoire que celui-ci n'a plus besoin de maintenir l'information sur le bus de données. La mémoire se déconnecte du bus et désactive le signal d'acquittement (3).

Figure 8.3



Lecture d'une donnée en mémoire *via* un bus asynchrone.

L'avantage du bus asynchrone est sa grande souplesse car il peut s'adapter à tous les types de composants, quelle que soit leur fréquence de fonctionnement. Malheureusement, cela se paie par une plus grande complexité du matériel, qui doit gérer des signaux supplémentaires. En fait, la spécialisation des bus (mémoire, entrées/sorties...) limite la variété des éléments qui peuvent y être connectés et l'on peut imposer un référentiel de temps sur un bus synchrone. Le bus asynchrone n'est maintenant plus utilisé comme bus principal de l'ordinateur, mais uniquement pour les entrées/sorties.

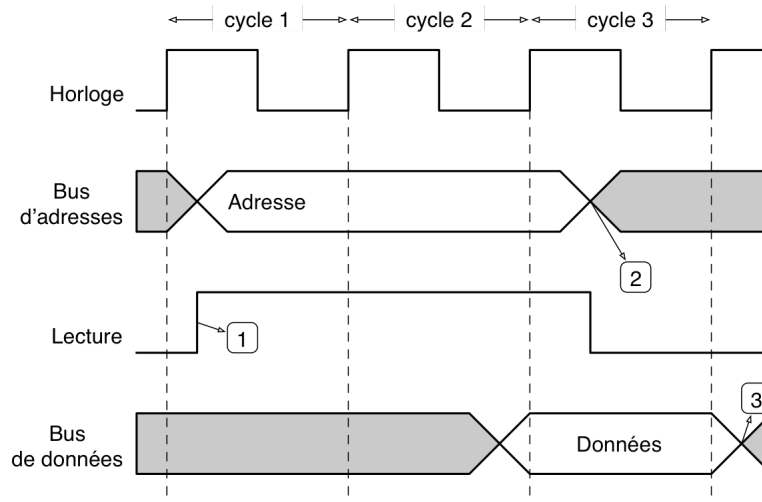
Bus synchrone

Le bus synchrone a la particularité d'intégrer un signal d'horloge sur un des fils de la partie contrôle. Ce signal permet aux différentes unités connectées de se synchroniser entre elles lors d'un échange d'informations. Les spécifications du bus précisent le diagramme temporel de chaque échange (qui doit faire quoi, à quel moment) et doivent être respectées par tous les composants connectés puisqu'il n'y a plus de signaux indiquant explicitement la disponibilité des informations. La figure 8.4 montre un exemple de lecture de donnée en mémoire *via* un bus synchrone.

Au début d'un cycle d'horloge, le processeur envoie l'adresse demandée sur le bus d'adresses et un signal de lecture mémoire (1). Les spécifications du bus indiquent qu'il doit maintenir ces signaux pendant deux cycles (2), temps maximum pendant lequel la mémoire peut travailler car il est aussi précisé que l'information renvoyée doit être disponible au début du troisième cycle d'horloge. Le processeur récupère alors cette information pendant le troisième cycle, à la fin duquel elle est retirée (3). Aucun signal n'indique les différents événements (mise en place de l'adresse, disponibilité des données...); seule la description temporelle des échanges dans les spécifications, respectées par tous, permet de savoir quand chaque composant doit faire son travail.

Le bus synchrone simplifie les unités fonctionnelles du point de vue matériel (puisque'il y a moins de signaux à gérer), mais oblige à une plus grande rigueur de construction car il faut intégrer les contraintes temporelles de chaque composant dès la conception.

Figure 8.4



Lecture d'une donnée en mémoire via un bus synchrone.

2. INTERRUPTIONS

Le processeur passe son temps à exécuter les instructions « écrites » par le programmeur et qui ne concernent que le fonctionnement des programmes de son cru. Cependant, un événement externe ou interne imprévu peut se produire, qui nécessite une réaction de l'ordinateur. Il faut alors temporairement interrompre le fonctionnement normal du processeur pour lui faire exécuter un programme en réponse à l'événement.

2.1 Origine des interruptions

Interruption externe

Le premier type d'interruption est un signal électrique véhiculé par le bus de contrôle et envoyé par un élément extérieur au processeur, qui l'informe qu'un événement vient de se produire et qu'il convient, peut-être, de le traiter :

- Soit il s'agit d'un événement prévu, qui se produit à un instant t connu. Par exemple, une interruption générée par un timer se déclenchant régulièrement permet au processeur d'effectuer périodiquement une tâche, sans que l'on ait besoin d'inclure les instructions correspondantes dans tous les programmes (par exemple, redonner régulièrement la main au système d'exploitation pour permettre une exploitation multitâche entre plusieurs processus).
- Soit il s'agit d'un événement prévu, qui se produit à un instant t non prévisible. Ainsi, une opération d'entrées/sorties étant longue par rapport au rythme de fonctionnement du processeur, celui-ci attend rarement qu'elle se termine pour poursuivre l'exécution d'un programme. En envoyant une interruption sur le bus de contrôle, l'unité d'entrées/sorties peut prévenir le processeur de la fin de l'opération demandée.
- Soit il s'agit d'événements imprévus, catalogués dans l'environnement de l'ordinateur, susceptibles de survenir à n'importe quel moment, voire jamais. Par exemple, un onduleur peut prévenir l'ordinateur d'une coupure de courant (nécessitant la sauvegarde des données pendant le temps de fonctionnement de la batterie de l'onduleur), une carte réseau peut signaler l'arrivée d'une communication, la souris peut indiquer un déplacement, etc.

Toutes ces interruptions externes sont aussi qualifiées de « matérielles ». Elles proviennent souvent de périphériques liés à l'ordinateur et informent le processeur d'un événement.

Interruption interne

Exception

Un deuxième type d'interruption, appelé « exception », caractérise un dysfonctionnement interne du processeur nécessitant un traitement particulier. Voici quelques exemples :

- **Erreur d'adresse.** Si aucun mécanisme de mémoire virtuelle n'est utilisé, une adresse mémoire doit correspondre à une adresse physiquement présente. Or, la taille maximale de l'espace adressable par le processeur dépasse presque toujours la taille de la mémoire physique installée. Il est donc possible que l'on fasse référence à une adresse mémoire inexistante, reconnue par aucun boîtier mémoire. Si la mémoire virtuelle est activée, l'accès à l'adresse peut être interdit (via le mécanisme de segmentation et de pagination, le système peut fixer des restrictions d'accès — écriture interdite, exécution non autorisée, etc. — sur des adresses, segments ou pages). Dans les deux cas, le processeur ne peut pas exécuter correctement l'instruction et doit donc signaler une erreur.
- **Instruction illégale.** Chaque instruction en langage assembleur est traduite en code numérique par le programme assembleur ; mais chaque code numérique n'est pas forcément associé à une instruction car la taille d'une instruction sous forme numérique (par exemple 32 bits) permet beaucoup plus de combinaisons que nécessaire. L'assembleur ne peut pas se tromper et générer un code incorrect, mais un bogue dans le programme (ou un programme malveillant de type virus) peut corrompre les instructions correctes situées en mémoire et les transformer en valeurs numériques erronées. Une tentative, par le processeur, d'exécuter un tel code provoque une exception et une erreur d'exécution du programme.
- **Erreur arithmétique.** Certaines opérations arithmétiques peuvent provoquer une exception. Ainsi, le programmeur demande parfois qu'un débordement soit traité comme une exception. L'exemple le plus classique est celui d'une division par zéro. Dans ce cas, le processeur ne peut pas effectuer le calcul et doit stopper l'exécution du programme car cela ne servirait à rien de travailler avec une fausse valeur numérique.
- **Défaut de page.** Une instruction fait référence à une adresse virtuelle contenue dans une page qui n'a pas encore été ramenée en mémoire principale. Il faut donc suspendre le programme en cours et demander au système d'exploitation d'aller chercher la page correspondante sur le disque.

Tous les cas d'exception relèvent du même traitement qu'une interruption externe : le processeur interrompt l'exécution du programme en cours et exécute des instructions spécialement prévues pour gérer la cause de l'interruption.

Appel système

Le dernier cas d'interruption provient également d'une source logicielle mais, cette fois, elle est totalement volontaire. Un processus peut souhaiter que le système d'exploitation effectue certaines opérations particulières, par exemple lancer une entrée/sortie, et lui donner la main en ce sens. Les instructions correspondantes ne sont jamais dans le processus lui-même (car elles dépendent directement du matériel, que ne connaît pas forcément le programmeur), mais incluses dans le code du système d'exploitation. Le processus doit donc appeler le système. Pour ce faire, il utilise une instruction spéciale du processeur, appelée, suivant les modèles, « appel système » ou « trappe » (INT, TRAP ou SC pour *System Call*). Elle joue exactement le même rôle qu'une interruption extérieure, c'est-à-dire qu'elle permet de suspendre le processus en cours pour exécuter une fonction de traitement de l'interruption (et ainsi donner la main au système).

2.2 Traitement d'une interruption

Lorsqu'une interruption survient, le processeur doit suspendre l'exécution du programme en cours et exécuter les instructions prévues par le système d'exploitation pour le traitement de l'interruption. L'algorithme utilisé est alors le suivant :

1. Le processeur finit de traiter l'instruction en cours (voir note ci-après).

Note

Il y a cependant quelques cas particuliers. Si c'est l'instruction elle-même qui a provoqué une exception fatale, elle n'est évidemment pas exécutée. Si le processeur est superscalaire et/ou fonctionne avec un pipeline (voir chapitre 3), il termine toutes les instructions en cours dans le pipeline avant de traiter l'interruption pour être sûr d'être dans un état stable. En revanche, si l'interruption est un défaut de page, le processeur ne peut pas compléter l'instruction. Il devra alors la relancer lorsque la page sera disponible en mémoire. Cela complique la gestion de l'interruption car l'instruction a pu être déjà partiellement exécutée et d'autres instructions postérieures peuvent être présentes dans un pipeline.

2. Le programme de traitement de l'interruption va certainement utiliser les registres du processeur. Il faut donc les sauvegarder pour que, lors de la reprise du programme principal (après traitement), le processeur puisse être remis dans l'état dans lequel il était avant l'interruption. On sauvegarde donc le contexte du processus : valeur des registres généraux, du compteur ordinal et du registre d'état, des pointeurs de pile, etc. Cette sauvegarde s'effectue classiquement en mémoire principale dans une

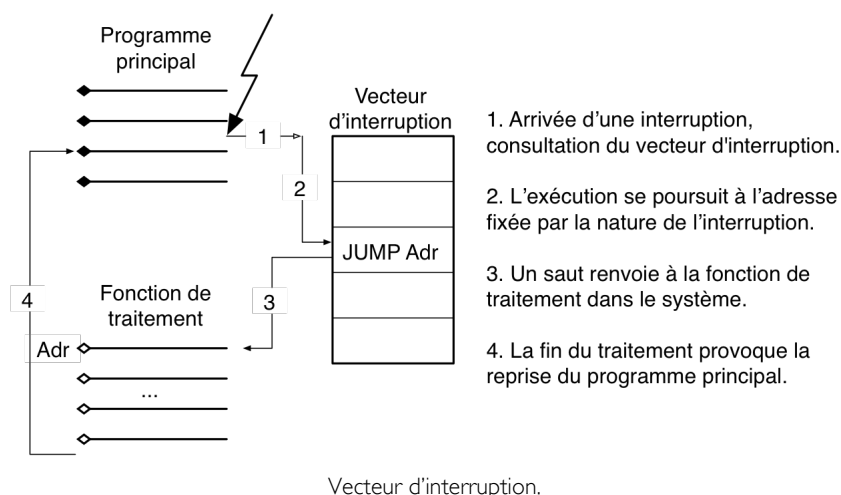
structure de pile spécialement prévue à cet effet. Elle est donc assez coûteuse en temps et on l'accélère parfois en limitant la quantité de registres sauvegardés : soit de manière imposée (par exemple en ne sauvegardant pas les registres flottants, rarement utilisés pour traiter une interruption et qui ne risquent donc pas d'être changés), soit de manière libre (en laissant le programme de traitement sauvegarder les registres qu'il modifie et uniquement ceux-là).

3. Dans le cas d'une interruption matérielle externe, le processeur doit en déterminer la source. On peut prévoir sur le bus de contrôle autant de lignes d'interruptions que de possibilités, mais au prix d'une forte complexité du bus et d'un manque de flexibilité. Le processeur peut aussi avoir une seule entrée d'interruption et procéder à l'interrogation successive de toutes les sources possibles *via* le bus. Mais, si elles sont nombreuses, cela ralentit alors le traitement. Autre cas de figure, toujours avec une seule entrée d'interruption : le composant déclencheur place sur le bus de données un identificateur informant le processeur de la cause et de la localisation de l'interruption.
4. Une fois la source de l'interruption détectée, le processeur doit calculer l'adresse mémoire de la fonction de traitement. Chaque modèle de processeur définit une liste d'interruptions et associe à chacune une adresse fixe où doit se trouver le début de la fonction de traitement. La place qui lui est réservée peut être soit de quelques centaines d'octets, et le système aura probablement la possibilité de traiter complètement l'interruption, soit limitée à une instruction de saut pointant ailleurs en mémoire sur le reste du code. L'ensemble de ces adresses fixes où se trouvent les différentes fonctions de traitement (même réduites à un simple branchement) s'appelle le vecteur d'interruption (voir figure 8.5).
5. Une fois l'adresse de la fonction de traitement mise dans le compteur ordinal, le processeur reprend son activité normale, en l'exécutant : il récupère la première instruction et poursuit son exécution jusqu'à une instruction spéciale, *Return From Interrupt*, qui indique la fin du traitement.
6. Si l'interruption n'est pas une erreur fatale ou ne nécessite pas la suspension temporaire du processus, le processeur récupère depuis la mémoire principale le contexte du programme initial pour remettre les valeurs sauvegardées dans les registres. Entre autres, il repositionne le compteur ordinal pour qu'il pointe sur l'instruction qui devait être exécutée au moment de la survenue de l'interruption.

Une interruption peut aussi forcer le système à suspendre le processus en cours, par exemple pour effectuer une entrée/sortie (en cas de défaut de page par exemple). À la fin du traitement, on ne récupère pas le contexte du processus initial (en attente que l'entrée/sortie se termine) mais celui d'un autre processus que le système choisira d'exécuter.

Enfin, il y a le cas des exceptions fatales (erreur d'adressage, instruction erronée, division par zéro...), qui obligent le système à terminer brutalement le processus et donc à en reprendre un autre à la fin du traitement de l'interruption.

Figure 8.5



Priorité

Toutes les interruptions n'ont pas la même niveau d'importance et s'organisent au sein d'une hiérarchie propre à chaque processeur et associée à une échelle de priorité. Les exceptions fatales sont plus graves (et nécessitent donc un traitement prioritaire) que les interruptions liées à une opération d'entrées/sorties (de priorité variable suivant la vitesse des périphériques).

Si une interruption survient pendant le traitement d'une autre, le processeur met en attente la moins prioritaire pour traiter celle de plus grande priorité. On peut ainsi avoir des traitements d'interruption imbriqués. Les interruptions moins prioritaires que le niveau courant de priorité sont dites « masquées » car leur traitement est différé. Le processeur peut en arriver à masquer toutes les interruptions, sauf celles dites « non masquables », qui correspondent aux cas les plus graves, nécessitant souvent un redémarrage général de la machine.

3. GESTION DES ENTRÉES/SORTIES

Les entrées/sorties se caractérisent par l'extrême diversité des périphériques, tant du point de vue du fonctionnement, de la vitesse, que des commandes. De plus, elles correspondent à la partie la moins figée de l'ordinateur : on remplace ou on ajoute des périphériques, tout en gardant le même processeur. C'est pourquoi il faut passer par un circuit spécialisé chargé de faire la traduction des signaux entre le bus système et les périphériques, au lieu de les relier directement. Ce circuit est le contrôleur d'entrées/sorties.

3.1 Diversité des périphériques

Les périphériques sont en charge de la communication de l'ordinateur avec l'utilisateur, du matériel ou d'autres ordinateurs, qui s'opère sous forme d'entrées/sorties. On distingue trois sens de communication :

- Les périphériques d'entrées : clavier, souris, écran tactile, tablette graphique, manette de jeu, scanner, caméra numérique, etc.
- Les périphériques de sorties : écran graphique, imprimante, enceintes, etc.
- Les périphériques d'entrées/sorties : imprimante multifonctions, disquette, disque dur, clef amovible, CD, DVD, bande magnétique, carte réseau, modem, etc.

Tous ces appareils sont variés et diffèrent sur de nombreux points :

- **La vitesse de transfert de l'information depuis ou vers le périphérique.** Elle peut aller de quelques octets par seconde, par exemple dans le cas d'un clavier, à plusieurs dizaines de mégaoctets par seconde pour un disque dur ou une interface réseau.
- **Le format de données.** Certains périphériques travaillent bit par bit, comme les modems réseaux, d'autres échangent les informations octet par octet, voire plusieurs octets à la fois pour plus d'efficacité et de rapidité.
- **Les commandes envoyées et les signaux émis.** Chaque type de périphérique envoie des commandes et émet des signaux qui lui sont propres, pour signaler une activité au processeur ou transférer des données. Un disque dur lance des commandes de déplacement de la tête de lecture ou signale le verrouillage d'une disquette, une imprimante prévient d'un manque de papier, une carte réseau informe de l'arrivée d'un accès entrant, etc.

Cette diversité est un frein à la connexion des périphériques sur le bus système car cela obligerait le processeur à les contrôler directement, et induirait un ralentissement des transferts de données et une diminution de la bande passante avec la mémoire principale. De plus, on cherche à optimiser le bus système pour maximiser son efficacité. Ainsi, chaque modèle de processeur, voire d'ordinateur, se retrouve avec un bus spécifique, empêchant sa normalisation. En revanche, l'interchangeabilité des périphériques pousse à avoir une certaine standardisation dans la façon de les relier à l'ordinateur.

On place donc entre le bus système (ou un premier bus de communication) et le périphérique un contrôleur d'entrées/sorties, chargé de faire l'interface entre la logique générale de l'ordinateur (que l'on retrouve sur le bus) et la logique propre au périphérique, éventuellement standardisée.

3.2 Contrôleur d'entrées/sorties

La tâche du contrôleur d'entrées/sorties, placé entre le bus et le périphérique, est :

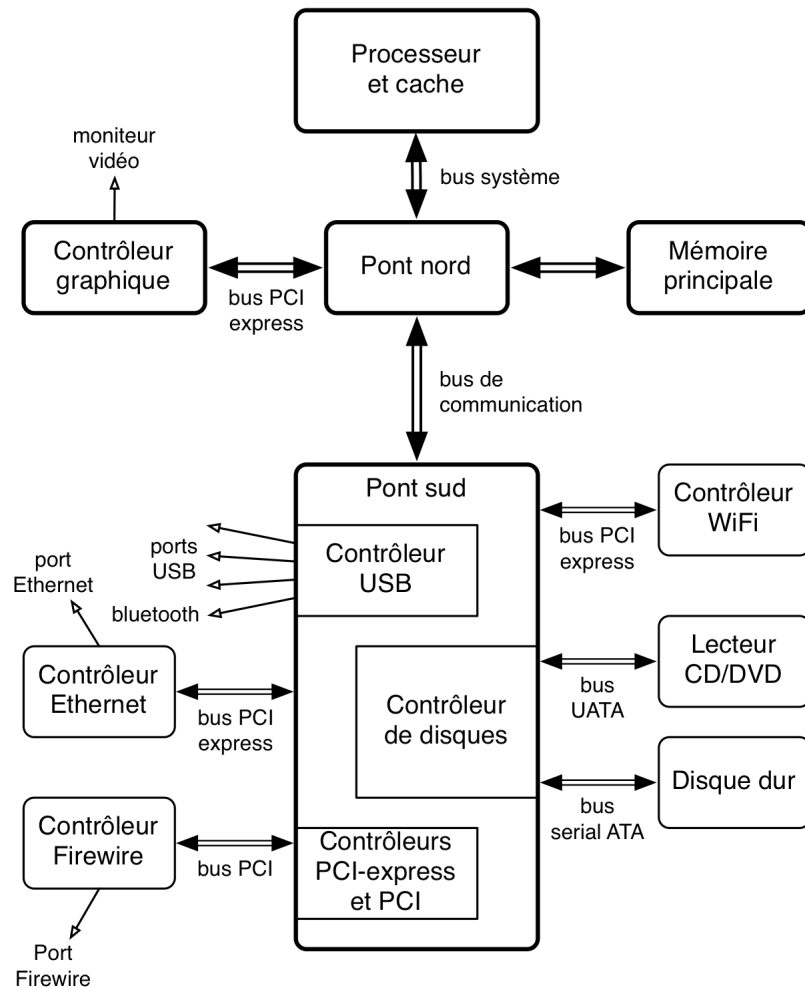
- de piloter l'opération d'entrées/sorties à la place du processeur ;
- de formater les données et de les mémoriser temporairement pour adapter leur format et la vitesse de leur transfert entre le bus de communication et le périphérique ;
- de permettre le branchement de divers modèles de périphériques *via* une interface externe standardisée (anciennes interfaces parallèle et série, interfaces PS/2, SCSI, USB, FireWire...).

Le contrôleur d'entrées/sorties est soit situé à l'intérieur d'un chipset (voir section 1.1) soit relié à ce dernier via un bus spécifique (bus PCI, PCI express...). Il est parfois sur une carte d'entrées/sorties amovible, reliée au bus spécifique par un connecteur. La figure 8.6 montre la structure simplifiée d'un ordinateur récent de type Macintosh.

Le contrôleur graphique transforme les commandes d'affichage envoyées par le processeur en signaux pour un moniteur vidéo. Les commandes passent d'abord par le pont nord qui les met (à l'aide d'un contrôleur PCI-express) dans un format compatible avec les spécifications du bus PCI-express sur lequel est branchée une carte vidéo.

Sur le pont sud se trouvent deux contrôleurs spécifiques, USB et disques, permettant de dialoguer avec les périphériques correspondants. Il y a aussi un contrôleur PCI-express et un contrôleur PCI permettant de relier divers contrôleurs extérieurs.

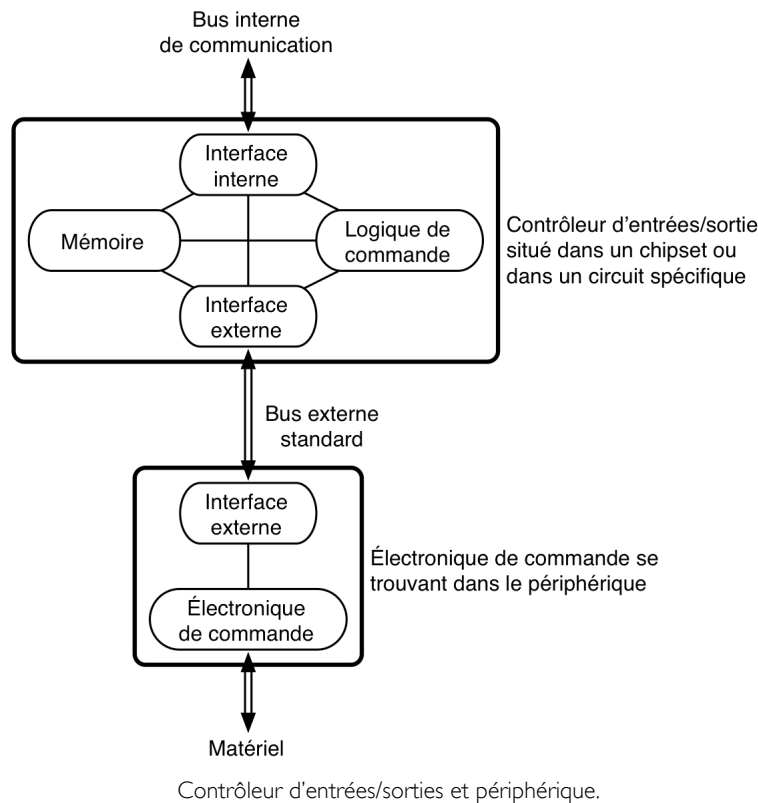
Figure 8.6



Bus et contrôleurs d'entrées/sorties.

Chaque contrôleur contient des circuits permettant de s'interfacier avec un bus (souvent PCI ou PCI-express) du côté de l'ordinateur, une logique de commande pour piloter le périphérique, une mémoire interne pour mémoriser les données en circulation, associée à des registres de commande et une interface externe standard pour connecter le périphérique. Du côté de ce dernier, on retrouve l'interface externe, ainsi que la logique de contrôle pour travailler directement avec le matériel, envoyer des commandes aux pièces mécaniques, adapter les signaux électriques, etc. (voir figure 8.7).

Figure 8.7



3.3 Gestion logicielle des entrées/sorties

Le programmeur souhaitant effectuer une opération d'entrées/sorties n'a pas à inclure dans son code les instructions nécessaires pour commander le contrôleur. Il y a une telle variété de circuits et de périphériques qu'il lui serait impossible de les connaître tous. De plus, cela nécessiterait une programmation en assembleur car les commandes correspondantes ne sont pas disponibles dans les langages évolués. Ces instructions se trouvent dans le système d'exploitation et c'est via ce système, en appelant la fonction voulue, par exemple par un appel système, que le programmeur déclenche l'opération. Pour chaque périphérique, il faut installer un pilote spécifique (*driver*) dans le système d'exploitation, qui contient tout le code nécessaire au contrôle de l'appareil : envoi de commandes au contrôleur, récupération des données, traitement des erreurs.

Espace des entrées/sorties

Les instructions processeur commandant les entrées/sorties peuvent être spécialisées, de type IN et OUT, et l'on a des lignes spécifiques aux contrôleurs d'entrées/sorties sur le bus de contrôle. L'autre possibilité est de rattacher les mémoires des cartes d'entrées/sorties (registres de contrôle indiquant la commande à effectuer ou le statut d'une opération, mémoire de données pour les transferts...) à l'espace mémoire général de l'ordinateur. Il n'y a alors plus d'instructions spécialisées et toutes les interactions avec le contrôleur d'entrées/sorties se font à l'aide de simples instructions de transfert mémoire (le contrôleur est connecté au bus d'adresses et, lors d'une lecture ou d'une écriture, il reconnaît son adresse ; il peut alors échanger les informations depuis ou vers ses registres ou sa mémoire via le bus de données).

La première solution est plus simple au niveau des cartes d'entrées/sorties, mais oblige à une spécialisation du bus de contrôle et donc offre moins de souplesse au niveau des différentes possibilités d'entrées/sorties. La seconde permet une uniformisation des commandes d'entrées/sorties (transfert vers ou depuis la mémoire), sans contrainte sur le bus de contrôle, mais les cartes d'entrées/sorties doivent contenir des circuits supplémentaires reconnaissant les adresses mémoire.

Entrée/sortie par interrogation

Comment un périphérique signale-t-il une activité au processeur ou transfère-t-il des données ? Il existe plusieurs modes de fonctionnement, qui diffèrent par la complexité associée aux circuits d'entrées/sorties. Pour des entrées/sorties simples, il est facile de demander au processeur d'interroger périodiquement le

périphérique afin de savoir si ce dernier nécessite son attention. Voici un exemple de lecture de données d'un périphérique (telle la position d'une souris) effectuée par interrogation :

1. Le processeur envoie une commande de lecture au périphérique.
2. Le processeur envoie une commande de lecture du statut du périphérique, qui indique soit que la donnée est prête soit que la lecture est toujours en cours. Dans ce dernier cas, il peut décider d'interroger de nouveau le périphérique un peu plus tard ou simplement de boucler sur l'envoi de la commande de lecture du statut jusqu'à disponibilité de la donnée. Il peut également signaler une erreur qui demande un traitement spécial.
3. Une fois la donnée disponible, le processeur la transfère dans une case mémoire (en passant par un registre interne).
4. Si toutes les données souhaitées ont été lues, l'opération est terminée, sinon le processeur reprend au point 1 en envoyant une nouvelle commande de lecture.

Cette entrée/sortie par interrogation (on parle aussi de scrutation ou de *polling*) est simple à gérer au niveau de la carte d'entrées/sorties car celle-ci est totalement passive, se contentant de répondre aux sollicitations du processeur. Mais ce mode de fonctionnement oblige le processeur à une attente active pendant tout le déroulement de l'opération, l'empêchant d'effectuer d'autres tâches. On ne peut donc se permettre de l'utiliser que pour des périphériques ayant un temps de réponse court et transférant peu de données à chaque fois, comme une souris que l'on interroge régulièrement (par exemple tous les centièmes de seconde) pour mettre à jour la position du pointeur.

Entrée/sortie par interruption

On peut apporter une première amélioration à l'algorithme précédent. Au lieu de consulter le contrôleur d'entrées/sorties en permanence, le processeur attend que celui-ci le prévienne de la fin d'une opération (erreur, disponibilité de la donnée) par une interruption. Reprenons l'exemple d'une lecture de données, utilisant cette fois une interruption :

1. Le processeur envoie une commande de lecture au périphérique.
2. Pendant le temps de l'opération, le processeur n'a pas à interroger le contrôleur et peut exécuter des instructions d'autres processus en attente.
3. À un moment, le contrôleur d'entrées/sorties envoie un signal d'interruption au processeur pour lui demander son attention. Le système d'exploitation reprend donc la main pour interroger le contrôleur en lui envoyant une commande de lecture du statut du périphérique. Celui-ci peut alors indiquer que la donnée est prête ou prévenir d'une erreur.
4. Si la donnée est disponible, le processeur la transfère dans une case mémoire (en passant par un registre interne).
5. Si toutes les données souhaitées ont été lues, l'opération est terminée, sinon le processeur reprend au point 1 en envoyant une nouvelle commande de lecture.

Ce mode de fonctionnement par interruption est bien adapté aux entrées/sorties asynchrones pour lesquelles on ne sait pas quand la donnée sera prête. Il permet au processeur de ne pas gaspiller le temps correspondant au délai d'attente de disponibilité du périphérique.

Cependant, comme précédemment, on perd encore beaucoup de temps dans les transferts de données car celles-ci empruntent deux fois le bus (une fois du périphérique au processeur, puis entre le processeur et la mémoire). De plus, le processeur est quand même relativement occupé puisqu'il doit gérer l'opération octet par octet. C'est donc un mode réservé aux transferts de données courts.

Entrée/sortie par accès direct à la mémoire

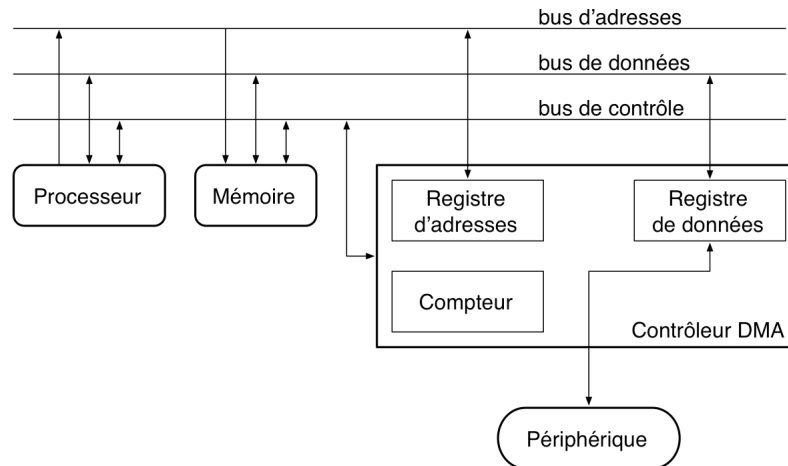
Lors du transfert de grandes quantités de données, le contrôleur d'entrées/sorties a tout intérêt à travailler directement avec la mémoire sans passer par le processeur. Il faut alors doter la carte d'entrées/sorties d'un peu d'intelligence par l'insertion d'un contrôleur DMA (*Direct Memory Access*). Ce dernier a la possibilité de prendre le contrôle du bus pour envoyer les données vers la mémoire depuis le périphérique, et inversement. Il inclut un registre d'adresses relié au bus d'adresses, un registre de données relié d'un côté au périphérique et de l'autre au bus de données, et un compteur (voir figure 8.8).

Pour effectuer la lecture de plusieurs octets et les placer en mémoire, le processeur initialise d'abord le registre d'adresses du contrôleur en lui envoyant l'adresse de la première case mémoire où les données sont à stocker. Le compteur, quant à lui, est chargé avec le nombre d'octets à transférer. Le processeur peut alors envoyer la commande de lecture au contrôleur (via le bus de contrôle), qui va se charger de toute l'opération.

Le contrôleur récupère les données du périphérique et les stocke dans une mémoire locale. Il utilise ensuite le bus pour les transférer vers la mémoire, à la manière du processeur. La première adresse est envoyée sur le

bus d'adresses avec la première donnée et un signal d'écriture mémoire. Une fois l'écriture effectuée dans la mémoire principale, le registre d'adresses est incrémenté, le compteur est décrémenté et, s'il n'est pas nul, le contrôleur recommence l'opération d'écriture avec la donnée suivante. Lorsque toute l'information voulue a été écrite en mémoire, le contrôleur prévient le processeur de la fin de l'opération par un signal d'interruption.

Figure 8.8



Entrées/sorties par DMA.

Le processeur est déchargé de tout le travail lié à l'entrée/sortie, mais cela augmente la complexité du contrôleur, qui doit avoir les circuits nécessaires au contrôle du bus. Il faut aussi faire attention au problème de conflits d'accès au bus entre le processeur et le contrôleur DMA, puisque les deux peuvent souhaiter l'utiliser pour leurs transferts de données. Souvent, le contrôleur DMA profite des instants où le processeur n'utilise pas le bus (accès au cache, décodage et exécution d'une instruction...) pour s'en rendre maître. Dans le schéma de la figure 8.6, ce contrôleur DMA se trouve intégré au chipset sur le pont sud, au niveau du contrôleur de disques.

Entrée/sortie par processeur spécialisé

Les entrées/sorties par DMA ne gèrent que les transferts de données. On peut généraliser la méthode en concevant une carte d'entrées/sorties possédant son propre processeur, qui exécute un programme stocké dans une mémoire auxiliaire, gérant alors la totalité de l'entrée/sortie (calculs, transferts, erreurs...). Le processeur principal indique simplement au processeur spécialisé l'adresse du programme à exécuter et est prévenu de la fin de l'exécution par une interruption. Cela décharge le processeur principal et permet d'exécuter des entrées/sorties compliquées (affichage vidéo associé à des calculs, multiples transferts de données...) car les programmes d'entrées/sorties du processeur spécialisé peuvent être complexes. De plus, ce dernier n'est pas généraliste, mais optimisé (au niveau de ses circuits électroniques internes) pour le domaine concerné, permettant une efficacité bien plus importante que si l'opération était gérée par le processeur de l'ordinateur (c'est par exemple le cas des processeurs graphiques situés sur les cartes vidéo).

Dans certains cas, les processeurs spécialisés communiquent avec les périphériques via d'autres circuits d'entrées/sorties qui fonctionnent suivant les modes précédents, monopolisant alors le processeur spécialisé sans monopoliser le processeur principal de l'ordinateur.

4. TECHNOLOGIES DE STOCKAGE

Les périphériques sont beaucoup trop nombreux et divers pour que l'on puisse les décrire tous en détail. Nous allons ici simplement présenter les technologies de stockage de mémoire de masse, indispensables aux ordinateurs actuels.

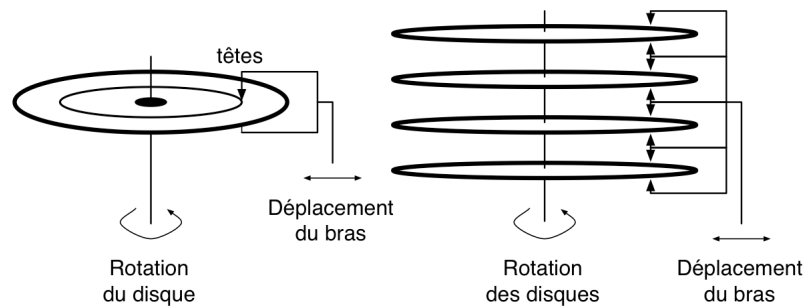
4.1 Technologie magnétique

Tous les stockages de masse fondés sur une technologie magnétique mémorisent les bits sous la forme de la polarisation magnétique de particules d'oxyde magnétique.

Une disquette est constituée d'un disque plastique recouvert d'un film de ces particules. Une tête de lecture/écriture (une par face), composée d'une bobine, se déplace au-dessus du disque suivant un rayon. Le disque lui-même est en rotation pour permettre aux têtes de lecture d'accéder à tout son contenu (voir figure 8.9). En écriture, le courant électrique est transformé en champ magnétique par la bobine, forçant la polarisation magnétique des particules se trouvant sous la tête. En lecture, on mesure le courant généré dans la tête de lecture lorsqu'elle passe au-dessus des particules. Suivant la polarisation, le courant induit est différent ; il est ensuite amplifié et mis en forme avant d'être envoyé au contrôleur.

Un disque dur est constitué de la même manière (voir figure 8.9), mais avec plusieurs disques en métal ou en verres superposés et des têtes de lecture réduites, pouvant travailler sur des particules plus petites (ces deux facteurs contribuant à une capacité plus importante).

Figure 8.9

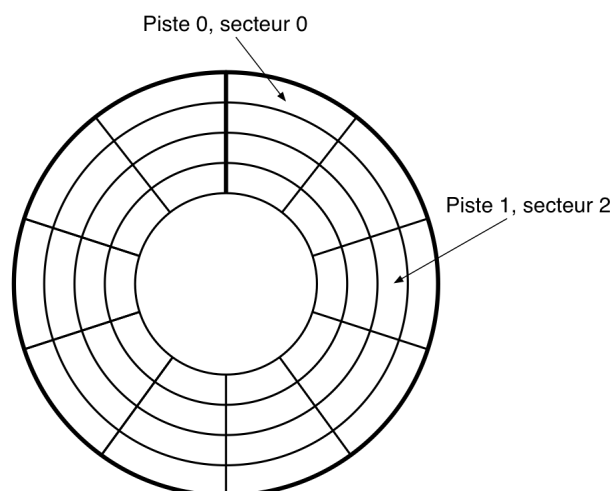


Disquette et disque dur.

Une autre différence entre une disquette et un disque dur est la vitesse de rotation du disque. Elle est de 300 tours par minute pour une disquette alors qu'un disque dur tourne à une vitesse de 5 000 à 15 000 tours par minute, permettant un débit d'informations plus important en provenance du disque dur (la tête de lecture « voit » passer plus de particules par seconde). De plus, à ces vitesses, un très fin coussin d'air se crée sous la tête, ce qui permet à cette dernière de flotter au-dessus du disque à une distance inférieure au micron, éliminant tous les frottements. Malheureusement, cela oblige également à sceller le disque dur pour éviter qu'une poussière (d'une taille bien supérieure à l'espace entre la tête et le disque) ne vienne perturber le fonctionnement.

Chaque face de chaque disque (appelée « plateau ») est divisée en pistes concentriques. La tête de lecture n'a donc pas un déplacement continu, mais pas à pas. Une piste est découpée en secteurs, souvent quelques dizaines (précisément dix à la figure 8.10). Un secteur contient des bits de données et des bits de contrôle servant à synchroniser l'horloge du lecteur avec la rotation du disque, à positionner la tête (en incluant l'identité du secteur) et à effectuer un contrôle d'erreurs. Cela explique que la capacité brute d'un disque soit plus importante que sa capacité utilisable, avant même un formatage de haut niveau par un système d'exploitation.

Figure 8.10



Un plateau

Pistes et secteurs.

Pour accéder à une information, il suffit de connaître sa localisation : plateau, piste, secteur et décalage par rapport au début du secteur.

Une bande magnétique fonctionne sur le même principe, mais le support est ici une bande plastique enroulée entre deux plots, qui défile devant une tête fixe. L'accès est séquentiel puisqu'il faut faire défiler toute la bande avant d'arriver à une information voulue. Elle est donc incompatible avec une utilisation courante, mais sert pour les sauvegardes.

4.2 Technologie optique

Tous les disques optiques fonctionnent sur le même principe d'enregistrement des bits : les deux états binaires sont définis par la réflexion lumineuse sur une surface. Une diode laser mesure cette réflexion pour détecter les bits individuels. On distingue différents types de disques optiques :

- **CD-ROM.** Il stocke les bits sous forme de trous disposés sur un support plastique ou en verre. Ces trous sont créés par pressage lors de la fabrication du CD-ROM. Ils ont un coefficient de réflexion différent du reste du support, permettant l'enregistrement des deux états binaires.
- **CD-R.** Le disque est vierge, mais un laser de puissance plus forte que celle nécessaire à la lecture peut « brûler » la surface, créant une zone réfléchissant mal la lumière. Une fois la surface brûlée, on ne peut plus modifier l'information. Ce disque est donc inscriptible une fois.
- **CD-RW.** Le disque est recouvert d'une couche composée d'un matériau pouvant passer, sous l'effet de la chaleur, d'un état amorphe à un état cristallin, et *vice-versa*, chacun ayant un coefficient de réflexion différent. Le laser peut donc chauffer localement le matériau pour changer son état et mémoriser un bit.
- **DVD-ROM.** Il fonctionne sur le même principe que le CD, à ceci près que le laser utilisé a une plus petite longueur d'onde, permettant de mesurer des « trous » plus petits et donc plus nombreux, d'où une capacité plus importante. Certains DVD utilisent les deux faces du disque pour doubler la capacité. D'autres proposent deux couches par face, un système de lentilles permettant de focaliser le laser sur l'une ou l'autre, là encore en doublant la capacité. Les DVD Blu-Ray utilisent un laser bleu, de longueur d'onde encore plus petite, et travaillant avec des « trous » encore plus petit et plus nombreux.

RÉSUMÉ

On retrouve à différents endroits de l'ordinateur des bus de communication, souvent découpés en trois parties : adresses, données et contrôle. Ils servent à véhiculer toutes les informations et commandes entre les composants de l'ordinateur, par exemple les signaux d'interruption qui permettent à un périphérique de prévenir le processeur d'un événement. Les périphériques sont commandés par des contrôleurs d'entrées/sorties placés entre les bus de communication internes et les interfaces externes via lesquelles se branchent les périphériques. Leur accès se fait au travers d'un pilote spécifique, qui doit préalablement être chargé dans le système d'exploitation.

Problèmes et exercices

Les entrées/sorties sont un mécanisme matériel géré par le logiciel. Il faut donc combiner les deux approches pour comprendre leur fonctionnement. Les exercices suivants mettent en pratique les notions d'interruptions, d'interrogation et de transfert par DMA. Il est également question de la répartition des bits sur une disquette.

Exercice 1 : Débit du bus

- 1) En reprenant la figure 8.4 (bus synchrone), calculez le débit du bus lors d'une opération de lecture sachant que sa fréquence est de 100 MHz et que le bus transfère 16 bits à la fois.
- 2) Un bus a une fréquence de 500 MHz et la mémoire peut envoyer 64 bits sur chaque front d'horloge (montant et descendant). Calculez le débit maximum, en ne tenant pas compte des temps d'accès à la mémoire (supposez que tous les cycles sont occupés par des données).

1) La mémoire envoie 16 bits, soit 2 octets, tous les trois cycles. Avec une fréquence de 100 MHz, chaque cycle dure 10 ns. Le débit est donc de 2 octets en 30 ns, soit 66,66 Mo/s.

2) Chaque cycle dure 2 ns mais la mémoire transfère les données à chaque front, donc deux fois par cycle, c'est-à-dire toutes les nanosecondes. Elle transfère 64 bits, soit 8 octets à la fois. Le débit maximum est donc de 8 Go/s.

C'est en fait un débit théorique car il correspond à un envoi de données en mode « rafale », une fois l'accès à la mémoire engagé ; normalement, il faut aussi quelques cycles préliminaires pour l'accès au boîtier.

Exercice 2 : Exception de défaut de page

Un processeur autorise le mode d'adressage indirect avec pré-décrémentation (voir chapitre 3). Pourquoi cela complique-t-il, au niveau du processeur, le traitement de l'exception de défaut de page ?

Le défaut de page est la seule exception qui risque d'interrompre une instruction au milieu de son exécution si elle fait référence à une adresse de la mémoire virtuelle non encore ramenée en mémoire principale. Comme le traitement du défaut de page est long, le processeur exécute d'autres instructions pendant ce temps-là. Lorsque la page virtuelle est chargée en mémoire physique et que le processus fautif a de nouveau la main sur le processeur, celui-ci relance, depuis son début, l'exécution de l'instruction ayant provoqué l'exception. Il ne faut donc pas tenir compte d'éventuelles modifications opérées lors de la première exécution, avortée, de l'instruction.

Dans le cadre d'un adressage indirect avec pré-décrémentation, comme dans l'instruction de chargement d'un octet LDB $r1, -(r2)$, le processeur calcule tout d'abord l'adresse mémoire référencée en prenant le registre $r2$ et en le décrémentant, puis il fait un accès mémoire. Si cette adresse virtuelle n'est pas encore en mémoire physique, la MMU génère un défaut de page, mais $r2$ a déjà été modifié. Si le processeur ne prend pas de précaution, il recommence la décrémentation de $r2$ (donc il l'effectue une fois de trop) lorsqu'il relance l'exécution de l'instruction après le traitement du défaut de page. La solution est de calculer l'adresse virtuelle en soustrayant 1 à $r2$, mais en ne validant la décrémentation de $r2$ qu'à la fin de l'exécution de l'instruction (et non pas lors de la première tentative d'exécution).

Exercice 3 : Priorités des interruptions

On considère les quatre types d'interruptions externes ou internes suivants :

- interruption externe provenant d'un périphérique ;
- tentative d'exécution d'une instruction illégale ;
- interruption thermique indiquant une surchauffe du processeur ;
- appel système.

Classez ces interruptions par ordre de priorité (quelle interruption peut interrompre le traitement d'une autre, moins prioritaire ?).

L'interruption thermique est la moins prioritaire car elle ne nécessite pas un traitement immédiat (au niveau de la nanoseconde). Il n'y a aucune raison de la privilégier alors qu'on pourra toujours la traiter (en ralentissant le processeur par exemple) une fois le traitement des autres interruptions terminé.

La tentative d'exécution d'une instruction illégale est la plus prioritaire car n'importe quelle fonction de traitement d'une interruption peut être erronée et comporter une instruction illégale. Il faut donc pouvoir interrompre son exécution.

Une interruption externe provenant d'un périphérique, probablement une fin d'entrées/sorties, nécessite un traitement par le système d'exploitation et donc la possibilité d'effectuer un appel système en interne. Ce dernier est donc plus prioritaire. Mais on n'a aucun risque à retarder légèrement une interruption externe arrivant pendant l'exécution d'un appel système, jusqu'à la fin de celui-ci.

Les interruptions sont donc classées comme suit par ordre de priorité décroissante :

- tentative d'exécution d'une instruction illégale ;
- appel système ;
- interruption externe provenant d'un périphérique ;
- interruption thermique indiquant une surchauffe du processeur.

Exercice 4 : Vecteur d'interruption

Quel est l'avantage d'avoir un vecteur d'interruption différenciant l'adresse de traitement de chaque interruption plutôt qu'un point d'entrée unique dans le système d'exploitation pour toutes les interruptions ?

S'il y a un point d'entrée unique, cela signifie que la différenciation de traitement se fait dans le code du système et toute modification du traitement d'une interruption implique une modification de ce code.

Si les interruptions sont déjà différenciées par le vecteur d'interruption, modifier le traitement de l'une d'entre elles revient simplement à modifier l'adresse du branchement localisé dans le vecteur (voir figure 8.5) pour pointer sur l'adresse de début de la nouvelle fonction, sans toucher au code du système.

Exercice 5 : Interrogation et interruption

Un dispositif d'entrées/sorties effectue dix requêtes par seconde, chacune d'entre elles nécessitant cinq mille instructions processeur pour être traitée.

1) Les entrées/sorties se font par interruption. Il faut mille instructions processeur pour sauvegarder le contexte et démarrer le gestionnaire de traitement de la requête, et de nouveau mille instructions processeur pour charger le contexte et revenir au processus principal. Combien d'instructions par seconde faut-il pour gérer les entrées/sorties ?

2) Les entrées/sorties se font maintenant par interrogation. Le processeur profite d'une interruption périodique préexistante tous les centièmes de seconde pour scruter le périphérique et voir s'il y a une requête à traiter. Il n'y a donc pas de coût supplémentaire de commutation de contexte. Chaque interrogation nécessite cinq cents instructions processeur, en plus du traitement de la requête si elle est présente. Combien d'instructions par seconde faut-il pour gérer les entrées/sorties ?

1) Chaque requête nécessite une sauvegarde du contexte, un traitement et un chargement du contexte, soit au total $1\,000 + 5\,000 + 1\,000$ instructions à chaque requête. Cela fait donc soixante-dix mille instructions par seconde pour traiter les dix requêtes du périphérique.

2) Le processeur effectue cent interrogations du périphérique par seconde, dont quatre-vingt-dix se révèlent infructueuses et dix aboutissent au traitement des requêtes. Les interrogations inutiles emploient cinq cents instructions processeur tandis que les dix autres utilisent $500 + 5\,000$ instructions (interrogation et traitement). Cela donne donc cent mille instructions par seconde :

$$90 \times 500 + 10 \times (500 + 5000) = 100000$$

Exercice 6 : Intérêt de l'interrogation

Quel peut être l'intérêt de gérer les entrées/sorties par interrogation plutôt que par interruption ?

Il est *a priori* plus long de gérer une entrée/sortie par interrogation puisque le processeur risque d'exécuter des instructions inutilement si le périphérique n'a pas de nouvelles données à envoyer. De plus, un appel du périphérique par interruption permet au processeur de réagir immédiatement si le périphérique le demande.

Il faut cependant nuancer ce propos :

- Le coût de traitement d'une interruption est assez important, et l'est relativement d'autant plus que le périphérique transfère peu de données (et donc nécessite peu d'instructions de traitement). Il peut alors être plus intéressant de profiter d'une autre interruption (qui sera traitée de toute façon) pour scruter rapidement le périphérique, sans perdre trop de temps. Une interruption périodique d'un timer se prête bien à ce fonctionnement si le périphérique ne nécessite pas de réaction instantanée (et peut attendre quelques millisecondes). Une souris peut être gérée par interrogation : aucun besoin d'une réaction immédiate du processeur, il faut juste mettre à jour la position du pointeur sur l'écran assez rapidement (en quelques millisecondes par exemple) pour que l'utilisateur ait une impression de rapidité ; de plus, il y a peu de données à récupérer et le traitement occupe peu le processeur.
- L'autre avantage de l'interrogation est que le processeur peut décider quand s'occuper des périphériques, alors qu'une interruption doit être traitée dès son arrivée (sauf traitement plus prioritaire en cours). Dans un système temps réel (où le facteur temps est pris en compte dans l'exécution des instructions), le déroulement d'un programme peut être fortement perturbé s'il peut être interrompu n'importe quand pendant son exécution. Si les périphériques sont traités par interrogation, le programmeur a l'entière maîtrise du déroulement temporel de son programme en décidant lui-même quand s'occuper des entrées/sorties.

Exercice 7 : DMA et interrogation

Un disque dur peut transférer des données sur le bus à la vitesse de 8 Mo/s.

1) Les entrées/sorties se font par interrogation. Chaque scrutation du périphérique prend $0,1\ \mu\text{s}$, temps pendant lequel sont transférés 4 octets. Combien d'interrogations faut-il chaque seconde pour soutenir le débit du disque ? Quel pourcentage de temps processeur est utilisé pour cela ?

2) Les entrées/sorties se font maintenant par DMA. $1\ \mu\text{s}$ est nécessaire pour initialiser le contrôleur DMA et encore $1\ \mu\text{s}$ pour traiter l'interruption de fin. Le disque peut transférer 4 Ko à chaque session DMA. Combien de temps prend chaque session de transfert ? Quel pourcentage de temps processeur est utilisé pour gérer le transfert de données ?

1) Chaque transfert concerne 4 octets, à la vitesse de 8 Mo/s. Il y a donc une interrogation toutes les $0,5\ \mu\text{s}$ en prenant l'hypothèse optimiste où le processeur ne scrute pas inutilement. Comme chaque scrutation occupe le processeur pendant $0,1\ \mu\text{s}$, 20 % du temps processeur est utilisé pour gérer le transfert de données depuis le disque (toutes les $0,5\ \mu\text{s}$, le processeur exécute des instructions pendant $0,1\ \mu\text{s}$ pour transférer les données).

2) Chaque session de transfert de données par DMA envoie 4 Ko à la vitesse de 8 Mo/s. Cela prend donc $0,5\ \text{ms}$. Le processeur utilise alors $2\ \mu\text{s}$ (initialisation du DMA et interruption finale) pour traiter la session. $2\ \mu\text{s}$ toutes les $500\ \mu\text{s}$ correspondent à un taux d'utilisation du processeur de 0,4 %.

Cela dit, le processeur peut être ralenti par les transferts DMA sur le bus (il risque de ne pas pouvoir utiliser le bus aussi souvent qu'il le souhaite, car il va entrer en conflit avec le contrôleur DMA) et donc perdre plus de temps que le calcul précédent ne le laisse penser.

Exercice 8 : Capacité d'une disquette

Une disquette haute densité a les caractéristiques suivantes :

- densité linéaire de 686,38 bits par millimètre pour la piste interne ;
- rayon de la piste interne de 24,6875 mm ;
- nombre de faces : 2 ;
- nombre de pistes par face : 80 ;
- nombre de secteurs par piste : 18 ;
- nombre d'octets de données par secteur : 512.

1) Calculez le nombre de bits sur la piste interne. Sachant que toutes les pistes ont le même nombre de bits, déterminez la capacité totale de la disquette.

2) Quelle est la capacité utile de la disquette ?

1) La piste interne fait :

$$24,6875 \times 2\pi = 155,6111 \text{ mm}$$

Elle contient donc $686,38 \times 155,1161 \approx 106\,469$ bits.

On a donc au total 106 469 bits par piste et par face, ce qui donne $106\,469 \times 80 \times 2 = 17\,035\,040$ bits ou 2 129 380 octets, soit environ 2,03 Mo de capacité brute.

2) On multiplie le nombre d'octets par secteur, le nombre de secteurs par piste, le nombre de pistes par face et le nombre de faces. Cela donne :

$$512 \times 18 \times 80 \times 2 = 1\,474\,560 \text{ octets}$$

Soit une capacité utile d'environ 1,41 Mo.

La perte de capacité est d'environ 30 % et elle est due aux bits de contrôle pour la synchronisation du lecteur, aux bits de contrôle d'erreurs, aux bits d'identification de chaque secteur, etc.

Bibliographie

- Carter N. P., *Schaum's Architecture de l'ordinateur*, EdiScience, 2002.
- Cazes A. et Delacroix J., *Architecture des machines et des systèmes informatiques*, Dunod, 2003.
- Ceruzzi P., *A history of modern computing*, MIT Press, 2003.
- Darche P., *Architecture des ordinateurs - Représentation des nombres en machine et codes*, Édition Gaëtan Morin, 2000.
- Darche P., *Architecture des ordinateurs - Fonctions booléennes, logiques combinatoire et séquentielle*, Vuibert, 2002.
- Darche P., *Architecture des ordinateurs - Logique booléenne, implémentations et technologies*, Vuibert, 2004.
- Goossens B., *Architecture et micro-architecture des processeurs*, Springer-Verlag France, 2002.
- Hennessy J. et Patterson D., *Architectures des ordinateurs*, Vuibert informatique, 2003.
- Hyde R., *Art of Assembly Language*, No Starch Press, 2003.
- Ifrah G. *Histoire universelle des chiffres*, Seghers, 1981.
- Irvine K., *Assembleur X86*, CampusPress, 2003.
- Ligonnière R., *Préhistoire et histoire des ordinateurs*, Robert Laffont, 1987.
- Maurette P., *Programmez en assembleur*, Micro Application, 2004.
- Press W., Teukolsky S., Vetterling W. et Flannery B., *Numerical Recipes*, Cambridge University Press, 2000.
- Rojas R. et Hashagen U., *The first computers*, MIT Press, 2000.
- Sierra K. et Bates B., *Java tête la première*, O'Reilly, 2004.
- Silberschatz A., Galvin P. et Gagne G., *Principes appliqués des systèmes d'exploitation*, Vuibert informatique, 2001.
- Strandh R. et Durand I., *Architecture de l'ordinateur*, Dunod 2005.
- Tanenbaum A., *Systèmes d'exploitation*, Pearson Education, 2003.
- Tanenbaum A., *Architecture de l'ordinateur*, Pearson Education, 2005.

Index

- 0x (préfixe hexadécimal), 3, 92
- Accès
 - aléatoire, 117
 - associatif, 117
 - direct, 117
 - séquentiel, 117
- Accumulateur, 62
- Addition
 - nombres entiers, 4, 7, 16, 32, 59, 93, 112
 - nombres flottants, 11, 59, 102
- Additionneur, 32, 42
- Adressage
 - direct, 63
 - immédiat, 62, 92
 - indirect, 63
 - indirect avec déplacement, 63, 82
 - indirect indexé, 64, 82
 - logique, 163, 173
 - par registre, 63
 - post-incrémenté, 64, 82
 - pré-décrémenté, 64, 82, 199
 - virtuel, 163
- Adresse mémoire, 63, 117
- Algèbre de Boole, 22
- Algorithmes
 - de remplacement, 142, 166, 182
- AND (fonction logique), 23, 27, 59, 93
- Anomalie de Belady, 183
- Antémémoire, 136
- Appel de fonction, 68, 83, 102, 114, 124
- Appel système, 102, 190
- Arithmétique, instruction, 93
- Arithmétiques, instructions, 59
- ASCII, code, 11
- Assemblage
 - langage, 53, 89, 102
 - programme, 53, 99, 102
- Assembleur, 53, 89, 99, 102
- Asynchrone
 - bascule, 35
 - bus, 187
- Bascule, 56
 - D, 36
 - flip-flop, 36
 - JK, 37, 48, 49
 - latch, 36
 - RS, 34, 48
 - T, 37, 48
- Base
 - 10, 4
 - 16, 3, 4, 15
 - 8, 3, 4, 15
 - changement, 3, 15
- BCD, codage, 4
- Belady, anomalie de, 183
- Big-endian, stockage, 125
- Binaire, calcul, 2
- Bit, 2
 - condition, 67, 90
 - de retenue, 67, 90
 - de signe, 6, 67, 90
 - dirty, 144, 167
 - poids faible, 3
 - poids fort, 3
- Boole, algèbre, 22
- Branchement, prédiction, 73, 85
- Bus, 186
 - arbitrage, 187
 - asynchrone, 187
 - d'adresses, 187
 - d'entrées/sorties, 57, 116, 186, 192
 - de contrôle, 187
 - de données, 187
 - PCI, 193
 - synchrone, 188
 - système, 56, 117, 127, 138, 186, 192, 199
- C (langage)
 - arrondi, 18
- compilation, 54
 - optimisation, 146, 172, 180
 - stockage des variables, 123, 133
 - type de variable, 13, 19
- Cache, 56, 70, 116, 117, 171, 178
 - associatif, 140
 - associatif par ensemble, 141
 - conflits, 140, 142, 145, 146
 - défaut de, 145
 - direct, 139
 - étiquette, 139, 141, 171
 - ligne, 138, 150
 - mixte, 141
 - multiniveaux, 147
 - multiprocesseurs, 147
 - optimisation, 146, 155, 156
 - performances, 147
 - protocole MESI, 148
 - réécriture, 143, 159
 - remplacement, 142
 - taille, 138
- Cadre de page, 163
- Calcul binaire, 2
- CD, 117, 198
- Cellule mémoire, 120
- Changement de base, 3, 15
- Chipset, 57, 187, 193
- Chronogramme, 39, 188
- CISC, 70
- Codage BCD, 4
- Codage Unicode, 12
- Code
 - ASCII, 11
 - exécutable, 54, 99
 - objet, 54
 - opération, 61, 100
 - source, 54, 100
- Comparaison, instructions de, 60
- Compilation, 54, 78
- Complément à 2
 - opération, 16

- représentation, 6, 15
- Compteur binaire, 38
- Compteur ordinal, 58, 66, 191
- Contrôleur d'entrées/sorties, 192
- Contrôleur système, 57, 187
- CPU, 55, 196
- Cross-assembleur, 99
- Cycle d'instruction, 67, 69
- De Morgan, lois de, 25
- Débordement, 8, 16, 41, 67, 83, 90
 - nombre non signés, 8
 - nombre signés, 8, 17, 83
- Décalage, instruction de, 60, 79, 94
- Décimale, base, 4
- Décodeur, 30, 122
- Défaut de page, 163, 166, 183, 190, 199
- Délai de propagation, 30, 34, 36, 39, 127
- Demi-additionneur, 32
- Directives d'assemblage, 100
- Disque dur, 117, 162, 196
 - performances, 128, 164, 197
 - taux de transfert, 118
 - temps d'accès, 118
 - temps de latence, 118
- Disquette, 196, 202
- Diviseur de fréquence, 38
- Division
 - nombre entiers, 5, 7, 59, 60, 93, 94
 - nombre flottants, 11, 59, 102
- DMA (accès mémoire), 145, 195, 201
- Double précision, 9
- DRAM, 121
- Driver, 194
- Dualcore, processeur, 77
- DVD, 117, 198
- Écroulement, 178
- Édition de liens, 54
- EEPROM, 120
- ENIAC, 52, 53
- Entrées/sorties
 - instructions, 194
- EPROM, 119
- ET (fonction logique), 23, 27, 59, 93
- Étiquette, 61, 63, 96, 99, 100
- Exception, 189
- Exécution dans le désordre, 74
- Exposant (nombre flottant), 9
- Expression algébrique, 23
- Extension de signe, 7, 59, 92
- FIFO (algorithme de remplacement), 143, 167, 182, 183
- Flash, mémoire, 120
- Flottant, nombre, 9, 17, 59, 66, 102
- Fonction
 - adresse de retour, 124
 - appel, 68, 83, 102, 114, 124
 - de hachage, mémoire virtuelle, 169
 - paramètres, 68, 83, 114, 124
- Forme canonique, 27
- Front, commande sur, 36
- Garbage collector, 124
- Gros-boutiste, stockage, 14, 125, 133
- Heap, 123, 128
- Hexadécimal, symbole, 2
- Hexadécimale
 - base, 3, 4, 15
 - constante, 62
- Hierarchie mémoire, 136
- Horloge, 35, 58, 70, 72, 187
- Hyperpage, 165
- Hyperthreading, 76
- IEEE 754 (norme), 9, 10
- Instruction, 66
 - cycle, 67, 69
 - format, 96
 - taille, 62, 75, 96
- Interprété, langage, 54, 78
- Interruption, 189, 195
 - défaut de page, 166, 190
 - masquage, 67, 191
 - priorité, 191, 200
 - traitement, 190
 - vecteur, 191, 200
- Java (langage), 54
- arrondi, 18
- optimisation, 146, 172
- stockage des variables, 124
- type de variable, 13
- Jeu d'instructions, 59, 90
- Karnaugh, tableau de, 28
- Langage C
 - arrondi, 18
 - compilation, 54
 - optimisation, 146, 172, 180
 - stockage des variables, 123, 133
 - type de variable, 13, 19
- Langage Java, 54
 - arrondi, 18
 - optimisation, 146, 172
 - stockage des variables, 124
 - type de variable, 13
- Langage machine, 53
- LFU (algorithme de remplacement), 143, 167
- Little-endian, stockage, 125
- Localité, principes de, 137
- Logique 3 états, 34
- LRU (algorithme de remplacement), 143, 167, 182
- Mantisse (nombre flottant), 9
- Mémoire
 - barrette, 123
 - cache, 56, 70, 116, 117, 171, 178
 - capacité, 116
 - cellule, 120
 - Flash, 120
 - morte, 119
 - performances, 118
 - principale, 56, 116
 - puce, 122
 - rafraîchissement, 121
 - temps d'accès, 118, 121
 - temps de cycle, 118
 - virtuelle, 117
 - vive, 119
 - volatilité, 119
- MFU (algorithme de remplacement), 167, 182
- Micro-instructions, 69
- Micro-ordinateur, 52

Microprogramme, 70, 120
 Mini-ordinateur, 52
 Minterme, 27, 28
 MMU (gestion de la mémoire), 163, 170
 Mnémonique, 53, 61
 Mode d'adressage, 62, 82
 direct, 63
 immédiat, 62, 92
 indirect, 63
 indirect avec déplacement, 63, 82
 indirect indexé, 64, 82
 par registre, 63
 post-incrémenté, 64, 82
 pré-décrémenté, 64, 82
 Mot binaire, 3
 Multicore, processeur, 77
 Multiplexeur, 31, 122
 Multiplication
 nombres entiers, 5, 7, 16, 44, 59, 60, 79, 93, 94
 nombres flottants, 11, 17, 59, 102
 Multiprocesseurs, système, 148
 Multitâche, 167
 Multithread, exécution, 75, 87
 NAND (fonction logique), 24, 28, 40
 NMRU (algorithme de remplacement), 143
 Nombre dénormalisé, 10
 Nombre entier, 66
 Nombre flottant, 9, 17, 59, 66, 102
 arrondi, 10, 17
 double précision, 9
 exposant, 9
 mantisse, 9
 simple précision, 9
 Nombre signé, 6, 13
 NON (fonction logique), 23, 59, 93
 NON-ET (fonction logique), 24, 28, 40
 NON-OU (fonction logique), 24, 28
 NOR (fonction logique), 24, 28
 Norme IEEE 754, 9, 10
 NOT (fonction logique), 23, 59, 93
 Octale, base, 3, 4, 15
 Octet, 3, 56, 117
 poids faible, 3
 poids fort, 3
 valeur maximale, 3
 OR (fonction logique), 23, 27, 59, 93
 OU (fonction logique), 23, 27, 59, 93
 Overflow, 8, 67, 90
 Pagination, 162, 173
 Parallélisme, 75, 87
 Paramètres de fonction, 68, 83, 114, 124
 PC (registre), 58, 66, 191
 PCI, 193
 Périphérique (d'entrée/sortie), 57, 116
 Petit-boutiste, stockage, 14, 125, 133
 Pile, 68, 83, 102, 123, 128
 Pilote d'entrées/sorties, 194
 Pipeline, 70, 71, 84, 190
 Poids faible (bit, octet), 3
 Poids fort (bit, octet), 3
 Pointeur de pile (registre), 68, 83, 102
 Polling, 194, 200
 Porte logique, 23, 65
 Prédiction de branchement, 73, 85
 Principes de localité, 137
 Processeur, 55
 d'entrées/sorties, 120, 196
 dualcore, 77
 multicore, 77
 Processus, 162, 191
 Programme de démarrage, 120
 PROM, 119
 Pseudo-mantisse (nombre flottant), 9
 Puce mémoire, 122
 Rafrâichissement mémoire, 121
 RAM, 119
 dynamique, 121
 statique, 121
 Ramasse-miettes, 124
 Registre, 38, 57, 61, 63, 66, 90, 116, 191
 d'état, 60, 67, 83, 95
 de segments, 173
 entier, 66
 flottant, 66
 largeur, 66
 nommage, 63, 66, 74
 Représentation
 complément à 2, 6, 15
 signe et valeur absolue, 6, 15
 virgule fixe, 9
 virgule flottante, 9
 RI (registre), 58, 66
 RISC, 70, 90
 ROM, 119
 Rupture de séquence, 60, 67, 72, 84, 95
 Segmentation, 165, 172
 Séquenceur, 58, 68
 Signe et valeur absolue, représentation, 6, 15
 Signe, extension, 7, 59, 92
 SIMD, 71
 Simple précision, 9
 SMT, 76, 88
 Soustraction
 nombres entiers, 5, 7, 16, 59, 93
 nombres flottants, 11, 59, 102
 SP (registre), 68, 83, 102
 SRAM, 121
 Stack, 123, 128
 Station de travail, 52
 Superscalaire, 73
 SWP, 71
 Synchronisme
 bascule, 35
 bus, 188
 Système d'exploitation, 163, 166, 183, 191, 194
 Table
 de vérité, 22, 30
 des pages, 164, 175
 des segments, 173

inverse, mémoire virtuelle, 168, 175	Trappe, 102, 190	VLIW, 74
Tableau de Karnaugh, 28	UAL, 57, 65, 90	Volatilité mémoire, 119
Tampon de réécriture, 144	largeur, 65	Von Neumann, architecture de, 52
Tas, 123, 128	Unicode, codage, 12	Write-back, 144
Test, instruction de, 60	Unité d'exécution, 73	Write-through, 143
TLB, mémoire virtuelle, 170, 178	Unité de contrôle, 58, 68	XOR (fonction logique), 24, 59, 93
Transfert de données, 59, 91, 125	Vecteur d'interruption, 191, 200	
Transistor, 22, 120	Virgule fixe, représentation, 9	
	Virgule flottante, représentation, 9	