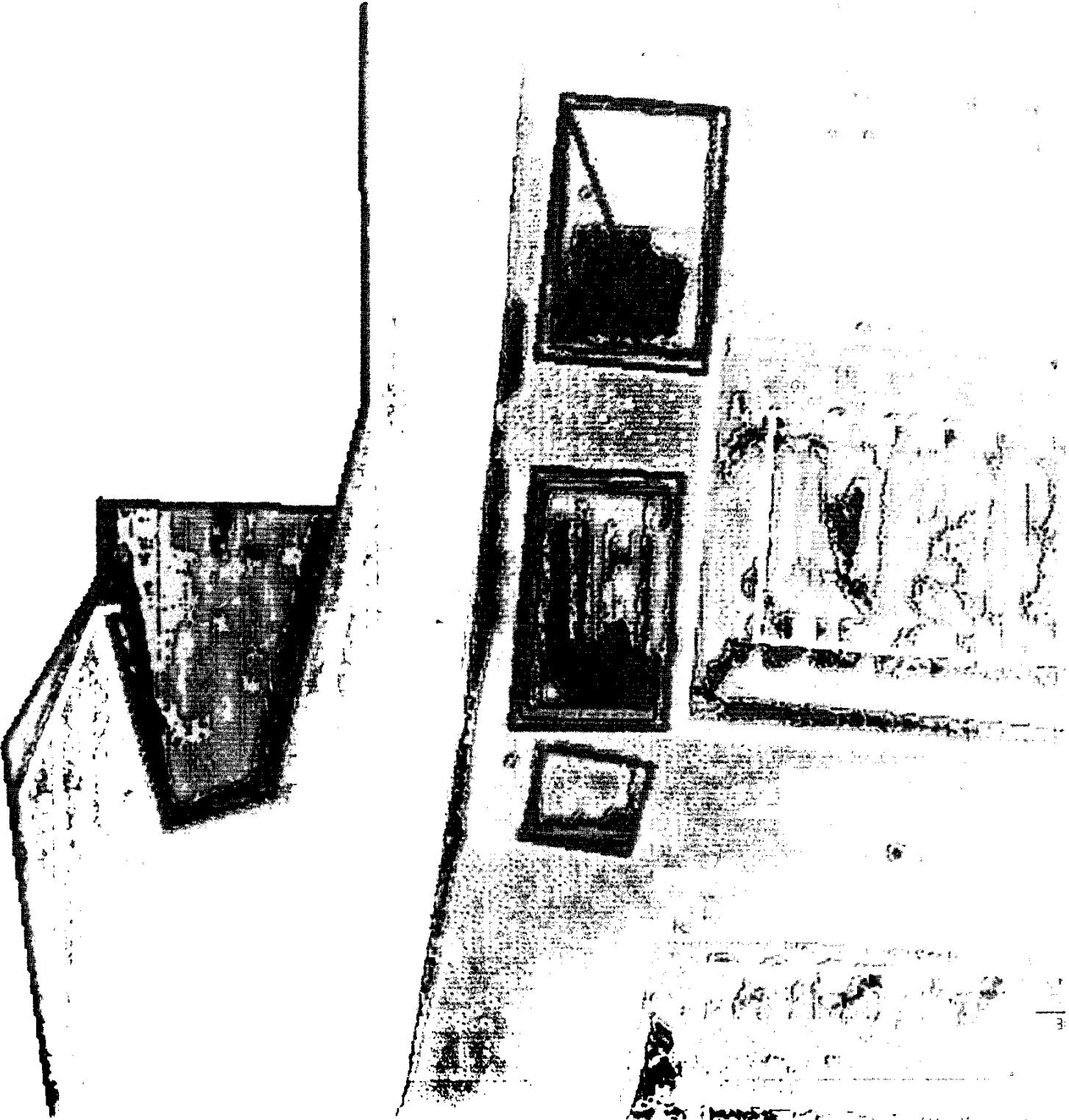


*Andrew Tanenbaum*

---

**ARCHITECTURE  
DE L'ORDINATEUR**

Troisième édition



# 4 La couche microprogrammée

<b>Chapitre 4 La couche microprogrammée</b>	221
4.1. Compléments sur la couche physique	222
4.1.1. Les registres	223
4.1.2. Les bus	223
4.1.3. Multiplexeur et décodeur	226
4.1.4. L'UAL et le décaleur	227
4.1.5. Les horloges	228
4.1.6. La mémoire principale	229
4.1.7. L'encapsulation des composants logiques	231
4.2. Un exemple de micromachine	234
4.2.1. Le chemin des données	234
4.2.2. Les micro-instructions	235
4.2.3. Cycle d'exécution de la micro-instruction	238
4.2.4. Enchaînement des micro-instructions	243
4.3. Un exemple de macromachine	244
4.3.1. La structure de pile	245
4.3.2. Jeu d'instructions de la macromachine	252
4.4. Un exemple de microprogramme	255
4.4.1. Langage de micro-assemblage	255
4.4.2. Le microprogramme	257
4.4.3. Remarques sur le microprogramme	262
4.4.4. Perspectives	263
4.5. Etude de la couche microprogrammée	264
4.5.1. Microprogrammation horizontale et verticale	264
4.5.2. Nanoprogrammation	271
4.5.3. Amélioration des performances	275
4.5.4. Traitement pipeline	278
4.5.5. La mémoire cache	284
4.6. Quelques exemples de couches microprogrammées	292
4.6.1. La micro-architecture du 8088 d'Intel	292
4.6.2. La micro-architecture du 68000 de Motorola	298
4.7. Résumé	304
4.8. Exercices	305

Dans l'ordinateur, la frontière entre matériel et logiciel n'est pas clairement établie; en outre, elle est sans cesse remise en cause, notamment par l'évolution technologique. Les premiers ordinateurs disposaient d'opérateurs spécifiques câblés (réalisés avec des circuits logiques) qui étaient utilisés pour l'exécution des instructions arithmétiques et logiques. Un examen visuel des circuits de l'UC permettait à l'œil averti de localiser les types de circuits, ceux relatifs à l'opérateur de division par exemple. Sur les ordinateurs modernes, cet examen visuel est plus délicat. En effet, il n'y a plus réellement de circuits spécifiques attachés à une fonction donnée.

Les instructions machine, relatives à la couche traditionnelle, sont extraites de la mémoire centrale par l'UC, les unes à la suite des autres, afin d'être exécutées par un interpréteur de commande qui reçoit ses directives du *microprogramme*: le programme spécifique à la couche microprogrammée. C'est ainsi que l'on ne distingue plus, par exemple, les circuits propres à l'opérateur de division car toutes les actions à entreprendre pour réaliser une division sont précisées dans le microprogramme. Il en est de même de la plupart des autres instructions qui sont également définies par le microprogramme.

En généralisant, on peut dire que tout programme est exécuté par un interpréteur de commandes qui puise ses directives dans un autre programme, qui lui-même peut être exécuté par un interpréteur de commandes qui puise, etc. Cette démarche n'est pas poursuivie longtemps. Il y a en effet, dans la machine, un niveau plancher que l'on ne peut repousser, ce sont les composants de la couche physique. C'est le niveau 0 de notre approche hiérarchique; les éléments du niveau 0 sont appelés les éléments durs, le *hardware*, de la machine.

Dans ce chapitre relatif à la couche microprogrammée qui forme le niveau 1, nous voyons comment les composants logiques sont assemblés et comment ils sont activés sous le contrôle du microprogramme. De même, on verra comment le microprogramme analyse et interprète la couche machine traditionnelle, objet du chapitre suivant. Au chapitre 8, nous étudierons une classe de machines microprogrammées, les machines RISC.

L'architecture de la couche microprogrammée constitue l'architecture de base de la machine, on l'appelle *micro-architecture* ou *micromachine*; elle définit le comportement primaire de l'ordinateur. La programmation à ce niveau 1 est souvent complexe, elle est à l'initiative du microprogrammeur. Les contraintes temporelles, les particularités fonctionnelles des composants logiques sont appréciées à ce niveau 1. Ces considérations ont conduit à dire de la microprogrammation que «c'est la réalisation de systèmes raisonnables confiée à l'initiative d'une machine déraisonnable.» (Rosin, 1974).

La couche microprogrammée a une fonction spécifique: s'adapter facilement à l'exécution de l'interpréteur de commandes d'une machine virtuelle. Cela suppose une organisation relativement souple et bien adaptée à la réalisation des cycles de base : recherche, analyse et exécution des instructions de la couche machine traditionnelle, sous les directives d'un microprogramme propre.

Ce sont les particularités d'organisation de la couche microprogrammée que nous étudierons dans ce chapitre. Ainsi nous commencerons par revoir les principaux composants de la couche physique (objet du chapitre 3), car ils sont les fondements de l'architecture de la couche microprogrammée et concernent directement le microprogrammeur. (Un microprogrammeur est quelqu'un écrivant des microprogrammes et non pas un petit programmeur.) Nous étudierons ensuite comment les instructions machine de la couche traditionnelle sont exécutées par un enchaînement de micro-instructions: celles qui forment le microprogramme; de nombreux exemples illustrent ces propos. Nous étudierons, pour finir quels sont les facteurs essentiels de conception à prendre en compte pour élaborer la couche microprogrammée. En guise de conclusion, nous examinerons les couches microprogrammées de nos composants de référence, ceux des familles Intel et Motorola.

#### 4.1.1. Les registres

Un registre est un dispositif capable de mémoriser une information. Au sein de la couche microprogrammée, les registres, souvent nombreux, sont utilisés pour assurer le stockage temporaire d'informations nécessaires à l'exécution de l'instruction en cours de traitement. Conceptuellement, registre et mémoire centrale sont semblables. Ce sont la localisation dans la machine, la capacité de stockage et le temps d'accès aux informations qui les diffèrentient. Les registres se situent à l'intérieur même de l'UC, alors que la mémoire est externe à l'UC. Un registre mémorise un certain nombre de bits, souvent un mot mémoire.

Plus les ordinateurs sont complexes et performants, plus ils disposent de registres. Sur les petites machines, peu performantes, la mémoire principale est utilisée pour stocker des résultats intermédiaires car le nombre de registres est faible.

Les  $n$  registres de la couche microprogrammée ( $R_0, R_1, R_{n-1}$ ) constituent une *mémoire locale* privée ou une *mémoire bloc-notes* pour l'UC. La mémoire centrale est plutôt considérée comme une *mémoire globale* partagée. Nous montrons à la figure 4-1 le formalisme de représentation d'un registre de 16 bits tel que nous l'utilisons dans l'ouvrage. L'information placée dans un registre est disponible à tout instant, elle est valide tant qu'une autre information ne vient pas la remplacer (et que l'alimentation électrique est maintenue!). Comme pour la mémoire principale, toute opération de lecture ne modifie en rien le contenu initial d'un registre.

Numéros d'ordre des bits

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	1	0	1	1	0	0	1	0

Figure 4-1. Représentation d'un registre 16 bits

#### 4.1.2. Les bus

Un bus est constitué d'un ensemble de fils sur lesquels on effectue la transmission de signaux en parallèle. Contrairement aux bus étudiés au chapitre 3, les bus utilisés dans la couche microprogrammée sont plus simples. Ils relient, le plus souvent, uniquement deux unités entre elles en mode point à point. Il n'y a donc pas de lignes d'adresses et le nombre de lignes de commandes est réduit au minimum. Sur les bus de la micromachine la transmission des signaux se fait en parallèle pour une efficacité maximale. Comme par exemple pour effectuer,

Un bus est constitué d'un ensemble de fils sur lesquels on effectue la transmission de signaux en parallèle. Contrairement aux bus étudiés au chapitre 3, les bus utilisés dans la couche microprogrammée sont plus simples. Ils relient, le plus souvent, uniquement deux unités entre elles en mode point à point. Il n'y a donc pas de lignes d'adresses et le nombre de lignes de commandes est réduit au minimum. Sur les bus de la micromachine la transmission des signaux se fait en parallèle pour une efficacité maximale. Comme par exemple pour effectuer,

le plus rapidement possible, le transfert du contenu d'un registre dans un autre registre. Au sens de la transmission des signaux entre deux unités *A* et *B*, un bus peut être unidirectionnel ou bidirectionnel. Sur un bus unidirectionnel les données transmettent dans une seule et même direction, de *A* vers *B*. En revanche, sur un bus bidirectionnel l'information est échangée dans les deux sens de transmission, de *A* vers *B* ou de *B* vers *A*, mais alternativement et non pas simultanément. Un bus unidirectionnel relie deux éléments (par exemple deux registres) dont l'un est source ou émetteur alors que l'autre est collecteur ou récepteur. Le bus bidirectionnel est plus général, il permet de relier divers éléments entre eux : des émetteurs, des récepteurs et des émetteurs-récepteurs.

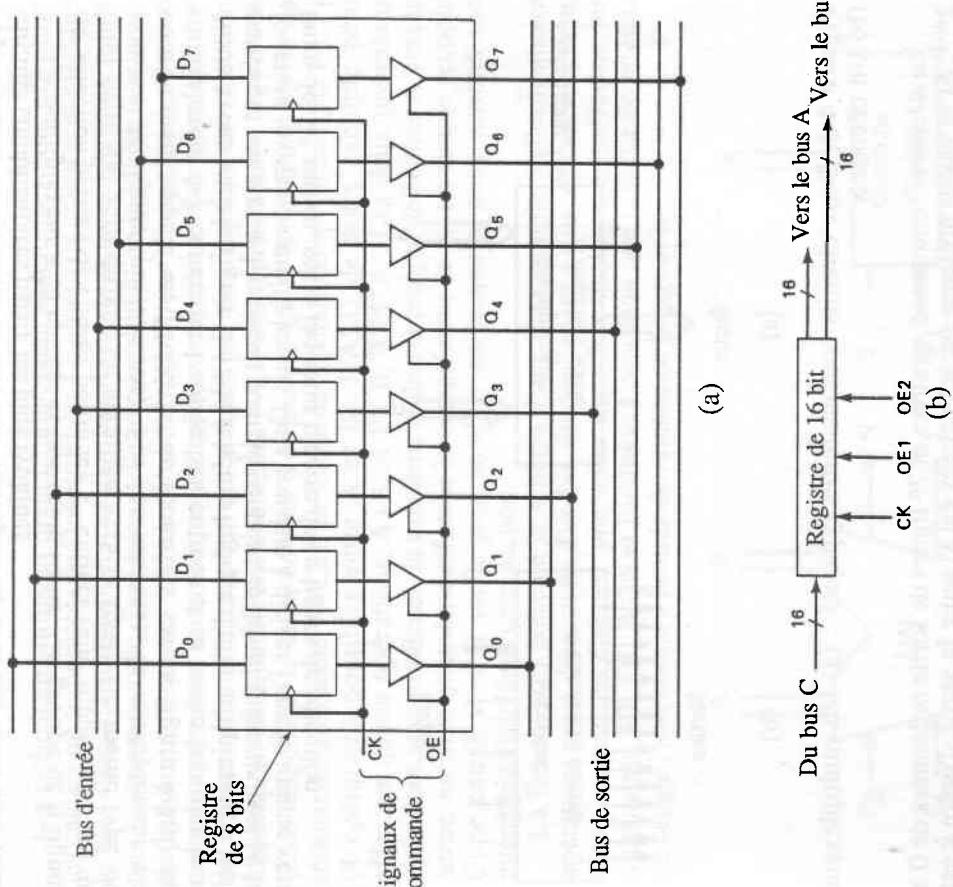
Sur un bus, bien que plusieurs éléments puissent y être connectés en permanence, il ne peut y avoir qu'une seule communication à la fois, entre un émetteur et un récepteur (quelquefois plusieurs récepteurs, on dit alors du bus qu'il est à diffusion).

En règle générale, les éléments reliés au bus peuvent s'y connecter ou s'y déconnecter électriquement. Ces connexions-déconnexions au bus se font en quelques nanosecondes. Un bus présentant ces propriétés est appelé bus à trois états, c'est-à-dire que chacune de ses lignes peut prendre les états logiques 1 ou 0 et l'état flottant. Les bus à trois états sont utilisés lorsqu'il s'agit de raccorder plusieurs éléments pouvant potentiellement émettre des données sur le bus.

Dans la plupart des micro-architectures, les registres sont connectés entre eux par des bus reliant leurs entrées et des bus reliant leurs sorties. Le registre de la figure 4-2 est constitué de huit bascules flip-flop D. Ses sorties sont reliées à un bus au travers de circuits à trois états. Le registre dispose de deux signaux de commande : un signal d'horloge, CK, réalisant le changement du registre et un signal d'activation des sorties, OE, qui place le contenu du registre sur le bus de sortie. Normalement, CK et OE sont à l'état de repos; ils sont activés suivant les opérations en cours. Lorsque CK est au repos, le registre n'est pas affecté par les informations circulant sur le bus des entrées. Dès l'activation de CK il est chargé avec la donnée présente sur le bus. Lorsque OE est au repos, le registre est en haute impédance sur le bus des sorties. Dès que OE est actif, le contenu du registre est placé sur le bus; il y est maintenu tant que OE reste actif.

C'est ainsi que, si les entrées d'un registre R<sub>2</sub> sont reliées aux sorties d'un registre R<sub>1</sub>, il est possible de copier le contenu de R<sub>1</sub> dans R<sub>2</sub>. Le signal OE de R<sub>1</sub> est activé et maintenu actif pendant un temps suffisamment long pour permettre au contenu de R<sub>1</sub> d'être stable sur ses lignes de sorties. Alors, l'horloge CK de R<sub>2</sub> est activée afin de charger la donnée en entrée dans le registre R<sub>2</sub>.

Il est fréquent que l'on réalise, dans la couche 1, des opérations de transfert de données entre registres. La figure 4-2(b) montre un registre de 16 bits disposant de deux bus en sortie. Une action sur CK charge le registre par la donnée présente sur le bus C. Les signaux OE<sub>1</sub> et OE<sub>2</sub> placent respectivement le contenu du registre sur le bus A ou sur le bus B.

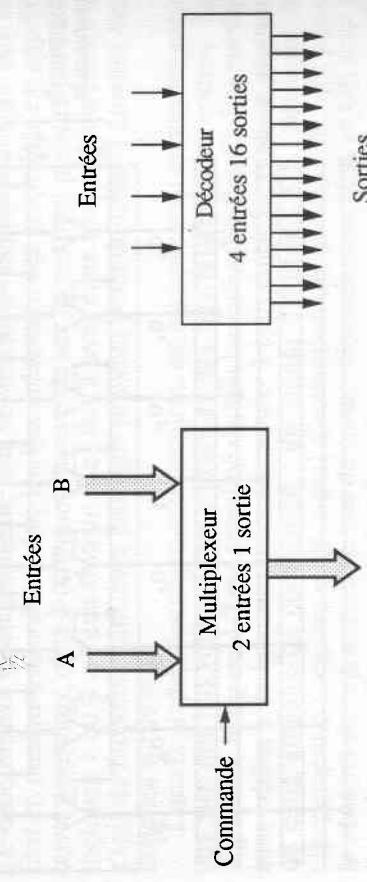


**Figure 4-2.** (a) Un registre de 8 bits connecté à un bus en entrée et à un bus en sortie (b) Représentation symbolique d'un registre de 16 bits relié à un bus en entrée et à deux bus en sortie

### 4.1.3. Multiplexeur et décodeur

Les circuits logiques qui ont une ou plusieurs entrées et réalisent une fonction logique ne tenant compte que de la valeur courante des entrées sont appelés des *circuits combinatoires*. Le multiplexeur et le décodeur sont deux circuits combinatoires parmi les plus typiques.

Le *multiplexeur* à  $2^n$  entrées et une seule sortie. Un groupe de  $n$  lignes de sélection permet de choisir parmi les  $2^n$  entrées (une simple ligne ou un bus) celle qui sera dirigée ou aiguillée vers la sortie (le même type ou nombre de lignes que l'entrée). Sur les schémas, le multiplexeur est couramment appelé un MUX. Nous montrons sur la figure 4-3(a) un multiplexeur à 2 entrées et 1 sortie; ses entrées et sa sortie sont des bus. Nous avons vu à la figure 3-12 le schéma logique d'un multiplexeur à huit entrées et une sortie. Un circuit complémentaire, le *démultiplexeur*, réalise la fonction inverse du multiplexeur. Cela consiste à diriger l'entrée unique vers l'une des  $2^n$  sorties, selon la valeur binaire des  $n$  lignes de sélection.



**Figure 4-3.** Deux circuits combinatoires typiques : (a) Un multiplexeur (b) Un décodeur

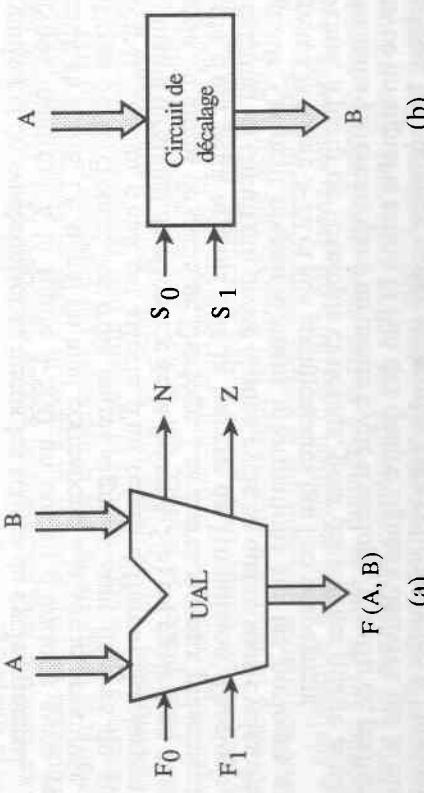
Le *décodeur* comprend  $n$  entrées et  $2^n$  lignes de sortie ordonnées de 0 à  $2^n-1$ . Si la valeur présente sur les entrées est  $k$ , seule la sortie d'ordre  $k$  est active parmi les  $2^n$ , les autres sorties restent à l'état de repos. Le décodeur analyse l'information qui se trouve sur ses entrées et fournit en sortie, de façon exclusive, l'indication ou la signification de cette information parmi plusieurs possibilités. Nous montrons à la figure 4-3(b) un décodeur 4 vers 16 (4 entrées, 16 sorties). La figure 3-14 dresse le schéma logique d'un décodeur 3 vers 8. Le décodeur a lui aussi un circuit complémentaire, l'*encodeur*, qui dispose de  $2^n$  entrées et de  $n$  sorties. Une seule entrée de

l'encodeur (parmi  $2^n$ ) est active à la fois, et sur les  $n$  lignes de sortie, on trouve, par exemple, la représentation en binaire de l'entrée active.

### 4.1.4. L'UAL et le décaleur

Les ordinateurs utilisent divers composants spécifiques pour effectuer les calculs ou traitements de l'information. Le circuit de calcul arithmétique le plus simple, l'additionneur, réalise la somme de deux nombres de  $n$  bits tout en tenant compte d'une éventuelle retenue d'addition et génère la retenue de l'addition courante. L'UAL (*Unité Arithmétique et Logique*) effectue diverses opérations arithmétiques et logiques entre deux opérandes. Le type de traitement à effectuer lui est précisé par des signaux de sélection d'opération. Nous montrons à la figure 4-4(a) une représentation type d'une UAL disposant de deux entrées d'opérandes (A et B) et d'une sortie de résultat de traitement F(A, B). Le choix des opérations est précisé par F<sub>0</sub>, F<sub>1</sub> (4 opérations possibles entre A et B). L'UAL de notre exemple peut effectuer quatre opérations, A+B, A ET B, A et A. Les deux premières opérations sont implicites, la troisième réalise la copie de l'entrée A vers la sortie, enfin la quatrième effectue le complément de A. La simple recopie d'une entrée vers la sortie peut sembler surprenante, nous verrons son intérêt plus loin dans cette section.

La figure 3-20 présente le schéma d'une UAL simple réalisant quatre opérations entre deux mots d'un bit. En dupliquant  $n$  fois, en cascade, cette UAL on peut en élaborer une autre de  $n$  bits de large qui effectue les traitements, ET, OU, NON et l'addition à condition que la sortie de retenue de l'étage  $i$  soit réunie à l'entrée de retenue de l'étage  $i+1$ .



**Figure 4-4.** Une UAL et un circuit à décalage

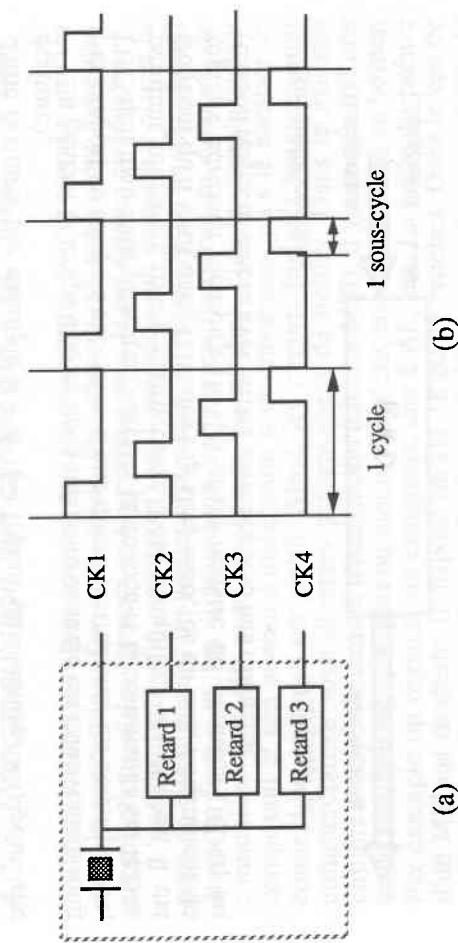


Figure 4-5. Un circuit d'horloge (a) et son chronogramme associé (b)

La plupart des circuits logiques dans l'ordinateur sont des circuits logiques synchrones commandés par une entrée d'horloge. L'*horloge* est un circuit qui transmet régulièrement, selon une périodicité déterminée, des impulsions électriques calibrées. Ces impulsions d'horloge définissent les cycles machine. Chaque cycle machine détermine une activité de base dans l'ordinateur. Par exemple l'exécution d'une micro-instruction, la recherche d'une instruction ou d'un opérande en mémoire centrale, etc.

Dans l'UC l'activité de base est variée et souvent complexe; il est fréquent que chaque cycle machine soit décomposé en sous-cycles afin d'organiser et de synchroniser au mieux les actions du cycle principal. Nous montrons à la figure 4-5(a) un circuit d'horloge avec quatre signaux en sortie. Le signal du haut correspond à la sortie principale, soit l'horloge de référence; les trois autres signaux sont dérivés du signal principal, chacun d'eux est affecté d'un retard différent. La largeur de l'impulsion du signal de référence de la figure 4-5(b) est égale à un quart de période ou temps de cycle de l'horloge. Les autres signaux présentent des retards d'un, deux et trois fois la largeur de l'impulsion d'horloge. On obtient ainsi un circuit qui divise chaque cycle en quatre sous-cycles d'égal longueur. On peut obtenir d'autres informations sur les horloges en se référant à la figure 3-22 et aux commentaires qui l'accompagnent.

Ainsi, à partir de l'exemple ci-dessus, pour déclencher quatre actions à des instants différents dans un même cycle d'horloge, il suffit de réaliser un ET avec un signal d'action et l'un des quatre signaux délivrés par le circuit d'horloge. La première action entreprise sera celle associée à l'horloge de référence, la seconde au signal d'horloge déphasé d'un quart de temps, etc.

Une UAL fournit également à l'extérieur des indicateurs relatifs à l'opération qui vient d'être traitée (sous forme de signaux à 1 ou à 0). Par exemple, un indicateur est à 1 lorsqu'un résultat en sortie de l'UAL est positif, négatif ou nul, de même lorsqu'il y a une retenue d'opération ou dépassement de capacité, etc. L'UAL de la figure 4-4(a) comprend deux indicateurs, *N*, qui indique un résultat négatif et *Z*, qui indique un résultat égal à zéro. Le bit *N* est une simple recopie du bit de signe (celui le plus à droite). En revanche, le bit *Z* est la réalisation d'une fonction NON-OU sur les sorties de l'UAL.

Bien que l'UAL réalise des traitements logiques (ET, OU, NON, décalages, etc.) il est quelquefois nécessaire qu'il y ait dans l'UC des circuits spéciaux de décalage. Ces circuits, que nous appelons décaleurs dans l'ouvrage, permettent de réaliser une opération de décalage à gauche ou à droite, sur les bits d'une information. La figure 4-4(b) montre le symbole que nous utiliserons pour représenter notre décaleur. Les signaux *S<sub>0</sub>* et *S<sub>1</sub>* précisent le sens du décalage. La figure 3-17 représente le schéma logique d'un décaleur à un bit.

#### 4.1.5. Les horloges

**4.1.6. La mémoire principale**

De façon quasi permanente, l'UC communique avec la mémoire principale pour lire ou écrire des données (instructions et opérandes). Dans un ordinateur, la mémoire principale ou centrale est reliée à l'UC par un bus parallèle. Cela sous-entend qu'il existe des lignes de données, d'adresses et de commandes permettant de synchroniser les échanges d'information entre l'UC et la mémoire.

Pour effectuer une opération de lecture mémoire, l'UC positionne sur le bus l'adresse du mot sollicité et active le signal de commande approprié précisant à la mémoire le type d'opération, par exemple le signal RD pour une lecture. Suite à cette action, la mémoire doit positionner sur le bus la donnée sollicitée. La mémoire met un certain temps (son propre temps d'accès) pour placer la donnée sur le bus. Sur certaines machines, les échanges sont synchrones, la mémoire doit positionner la donnée sollicitée sur le bus en un temps fixé. Sur d'autres machines, les échanges sont asynchrones, la mémoire «prend tout son temps», elle active en effet un signal de commande précisant que la donnée sollicitée est sur le bus.

La réalisation d'une opération d'écriture est assez semblable. L'UC positionne sur le bus la donnée à écrire ainsi que l'adresse mémoire associée. Un signal de commande est activé précisant à la mémoire l'opération d'écriture, par exemple un signal WR (écriture). Sur certaines machines, un seul signal est utilisé pour préciser s'il s'agit d'une lecture ou

d'une écriture, par exemple R / W (= 1 pour une lecture, = 0 pour une écriture).

En général le temps d'accès de la mémoire centrale est relativement long par rapport au temps d'exécution d'une micro-instruction. En conséquence, l'UC doit maintenir présente l'adresse, la donnée et la commande sur le bus pendant plusieurs micro-instructions. Pour simplifier les choses, il est fréquent qu'il y ait dans l'UC des registres associés au bus, notamment un registre adresses mémoire (RAD) et un registre de données (RDO) en relation respectivement avec le bus adresses et le bus données.

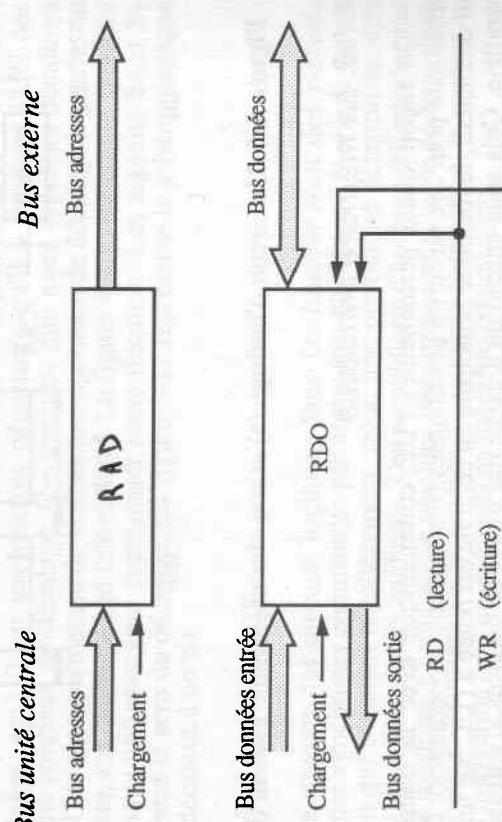


Figure 4-6. Registres tampons d'adresses RAD et de données RDO

Pour notre convenance, dans l'ouvrage, nous disposons volontairement le bus comme indiqué à la figure 4-6 : les registres RAD et RDO sont placés entre le bus de l'UC et le bus externe sur lequel sont connectés la mémoire et éventuellement d'autres circuits d'E/S. Le bus adresses est unidirectionnel de part et d'autre de RAD ; le signal de commande appliqué à RAD réalise son chargement par l'UC. Le contenu de RAD est en permanence diffusé sur les lignes d'adresses du bus externe. Le signal de commande appliquée par l'UC, à RDO réalise le changement de ce qui se trouve sur le «bus données entrée» dans le registre. Le «bus données sortie» de RDO (côté UC) est toujours actif, le contenu de RDO y est sans cesse positionné. Le bus données externes est bidirectionnel. Ce sont les signaux RD et WR qui déterminent le sens des lignes de données ; WR actif transmet sur le bus externe le contenu de RDO et RD actif prépare RDO à recevoir une donnée en provenance d'un élément sur le bus externe.

#### 4.1.7. L'encapsulation des composants logiques

Dans les sections précédentes nous avons étudié divers circuits entrant dans la conception d'un ordinateur. Ces circuits se présentent avec des organisations et des formes variées comme nous le présentons ci-dessous. L'*encapsulation* (le *packaging*) est l'opération qui consiste à placer la puce de silicium dans un boîtier protecteur (en plastique ou en céramique) dont la forme et les dimensions varient en fonction du nombre de connexions à établir avec l'extérieur. En général, plus la complexité du composant et son niveau d'intégration sont importants, plus le nombre de connexions externes est grand. Ainsi, les circuits intégrés logiques sont commercialisés sous diverses présentations de boîtier. Dans le cas des circuits à moyenne densité d'intégration (circuits MSI), il y a en général une seule fonction logique, plus ou moins complexe, encapsulée dans un boîtier. Par exemple un registre, un compteur, une UAL, etc. La taille du boîtier d'un MSI est modeste, il dispose de 16, 18 ou 24 broches. Dans le cas de circuits plus denses (LSI ou VLSI), les boîtiers sont plus imposants, le nombre de broches varie de 40 à 68, voire une centaine et plus pour des circuits très complexes.

La conception d'un système logique repose sur les circuits disponibles du marché (circuits MSI, LSI, VLSI). Le choix du type de circuits n'est pas neutre, il influe sur certains paramètres de conception, comme, par exemple, le nombre de boîtiers, l'encombrement, la consommation électrique et la dissipation thermique, la vitesse de traitement, les performances, etc. Nous montrons à la figure 4-7 trois approches différentes pour la conception d'un même système logique, par exemple un processeur de traitement.

La première approche (figure 4-7(a)) correspond à l'utilisation de circuits standard (type MSI). Cette solution nécessite bon nombre de circuits différents, mais à choisir de façon aisée parmi tous ceux disponibles sur le marché. Ces circuits une fois assemblés et câblés sont regroupés souvent sur une ou plusieurs cartes de circuits imprimés. L'encombrement est maximum, la consommation électrique et la dissipation thermique le sont également. En revanche, la vitesse de traitement est grande et les performances générales sont excellentes.

Une autre approche (figure 4-7(b)) consiste à utiliser des circuits d'un autre type, plus denses, qui regroupent en leur sein des fonctionnalités supérieures à celles des circuits MSI. Leur capacité de traitement est parfois limitée mais facilement extensible, ce sont les *circuits en tranche* ou *bit slice*. En associant plusieurs circuits en tranche identiques en cascade, il est possible d'ajuster au mieux les nécessités de traitement au format des données manipulées. Il existe des circuits en tranches (surtout des UAL) de 2, 4, 8 voire 16 bits. Il est ainsi facile de construire, par exemple, un processeur de traitement de 32 bits avec des circuits de 2, 4, 8 ou 16 bits. Par rapport à l'approche «circuits MSI», l'utilisation de circuits en tranches pour la conception est une solution plus économique et nécessite moins de circuits. L'encombrement, la consommation électrique et la dissipation

thermique sont plus faibles. La rapidité de traitement et les performances globales sont également à la baisse.

La troisième approche (figure 4-7(c)) correspond à l'utilisation de circuits à haute densité (LSI ou VLSI) du commerce, voire de circuits spéciaux fabriqués sur mesure (cette solution n'est viable que pour d'importantes quantités de circuits). Dans ce cas, on n'utilise qu'un seul circuit pour réaliser la fonction souhaitée. On est passé ainsi d'une solution multi-circuits à une solution plus simple mono-circuit. Cette approche mono-circuit offre bien des avantages (encombrement réduit, moindres consommation et dissipation, etc.), mais elle présente certains inconvénients (vitesse et performances sont beaucoup plus faibles, la souplesse de conception est également plus limitée). Si la rapidité de traitement n'est pas un critère de conception important, cette dernière approche est plus simple et plus rapide à mettre en œuvre. Les autres approches sont plus complexes et nécessitent souvent des compétences en électronique plus importantes.

Une bonne démarche de conception dans le cas d'applications standard consiste à envisager l'utilisation de circuits LSI et d'aller vers des solutions MSI ou circuits en tranches quand on ne peut faire autrement.

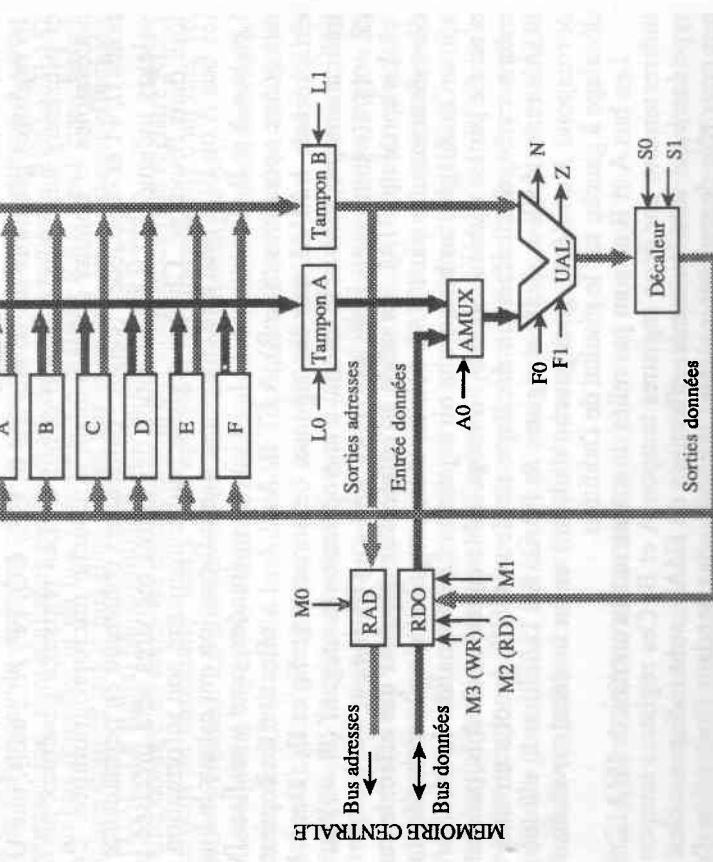
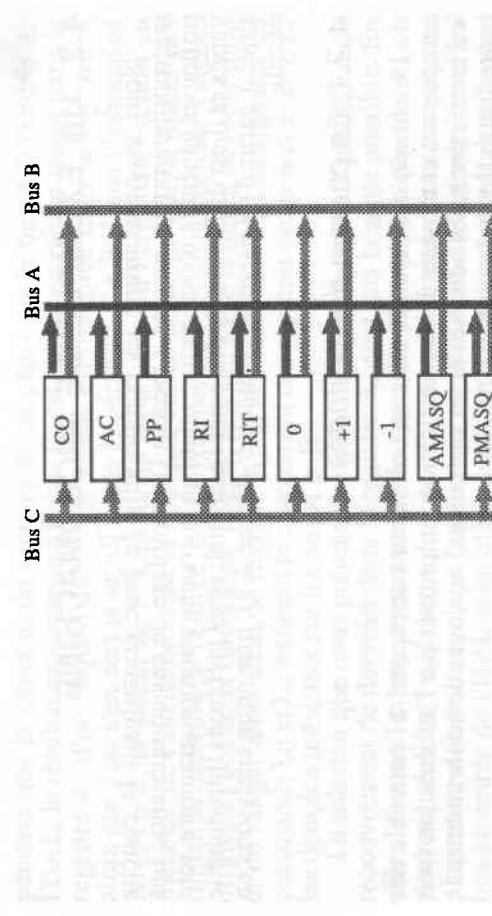


Figure 4-7. Trois approches de conception selon l'encapsulation des circuits: (a) Circuits MSI (b) Circuits en tranche (c) Circuit LSI

Figure 4-8. Chemin des données de la micromachine utilisée dans ce chapitre

## 4.2. UN EXEMPLE DE MICROMACHINE

Nous avons examiné les composants de base constituant la couche microprogrammée; voyons à présent l'assemblage de ces composants. Les principes généraux d'architecture relatifs à la couche microprogrammée sont variés et il est difficile d'en discerner l'essentiel; nous préférons introduire le sujet à partir d'un exemple particulier (figure 4-8), que nous analysons en détail.

### 4.2.1. Le chemin des données

Le *chemin des données* dans un ordinateur correspond à l'ensemble des composants et moyens de communications empruntés par l'information (soit les instructions, les adresses et les opérandes) au cours de son traitement : les registres, les mémoires, l'UAL, les bus.

Sur notre micromachine (figure 4-8), le chemin des données comprend 16 registres généraux de 16 bits (A, B, C, .. CO, PP, AC, etc.), une UAL et plusieurs bus : bus A, bus B et bus C. Les registres généraux ne sont accessibles et connus qu'au sein de la couche microprogrammée. Ceux notés 0, +1 et -1 sont des registres particuliers contenant en permanence les valeurs indiquées. La signification des autres registres sera précisée plus loin dans l'ouvrage. Chaque registre peut transmettre son information sur les bus A ou B; il peut être chargé par une information qui est sur le bus C. Les bus A et B alimentent l'UAL 16 bits en opérandes, pour y réaliser l'une des quatre opérations : (A+B), A ET B, A et A. La sélection de l'opération est précisée à l'UAL par les signaux de commande F<sub>0</sub> et F<sub>1</sub>. Deux bits indicateurs précisent si le résultat d'une opération est négatif (N = 1) ou nul (Z = 1). Ils sont positionnés par l'UAL après un traitement.

La sortie de l'UAL est reliée à l'entrée d'un décaleur qui effectue sur la donnée transmise par l'UAL soit un transfert pur et simple sans décalage, soit un décalage d'un bit à droite ou à gauche. La commande du décaleur est réalisée par les signaux S<sub>0</sub> et S<sub>1</sub>. Il est possible en une seule fois, pendant le même cycle, d'effectuer un décalage de deux bits à gauche en utilisant l'UAL et le décaleur. Soit un registre R; l'UAL fait l'addition R + R (ce qui correspond à un décalage à gauche d'un bit) enfin le décaleur réalise un décalage à gauche sur le résultat de l'addition.

Les bus A et B ne sont pas reliés directement aux entrées de l'UAL, mais indirectement, par deux registres tampon A et B. Ces registres tampon de type *latch* qui sont les entrées effectives de l'UAL, sont indispensables. Ils ont pour rôle de maintenir stables les opérandes pendant l'exécution d'une opération par l'UAL. En effet, l'UAL est un circuit combinatoire qui traite continûment les données qui se présentent sur ses entrées. Cela pose un problème si l'UAL doit réaliser, par exemple, A := A + B. En effet,

pendant que le contenu du registre A est placé sur le bus A à l'entrée de l'UAL, le résultat en sortie, sur le bus C, est chargé en même temps dans le registre A; d'où l'équivoque : une valeur est modifiée dans le registre A alors que l'on s'en sert pour effectuer l'opération A + B. Dans l'instruction A := A + B, la valeur de A à droite du signe «:=» est l'originale est non un mélange de l'ancienne et de la nouvelle valeur de A. Le fait de placer des registres tampon à l'entrée de l'UAL permet de disposer des valeurs originales des deux opérandes, A et B, pendant toute la durée de l'opération et de ranger alors dans le registre A, sa nouvelle valeur. Les signaux de commande L<sub>0</sub> et L<sub>1</sub> assurent le chargement des registres tampon A et B avec les données présentes sur les bus A et B.

La solution que nous présentons ci-dessus, pour résoudre le problème du recouvrement de données dans le registre A, n'est pas la seule possible. En effet, si tous les registres étaient de type *flip-flop* plutôt que *latch*, on pourrait réaliser l'opération A := A + B sans problèmes. Par exemple, en plaçant au début du cycle le contenu du registre A et du registre B sur les bus en entrée de l'UAL et en effectuant l'enregistrement du résultat dans le registre A en fin de cycle, sur le front de l'horloge.

Les échanges de données avec la mémoire principale, externe à notre micromachine, sont réalisées par l'intermédiaire des registres RDO et RAD via un bus données et un bus adresses. Le registre RAD peut être chargé avec une donnée sur le bus B en même temps qu'une opération est effectuée par l'UAL. Le signal M<sub>0</sub> commande le chargement de RAD.

Une donnée en sortie du décaleur peut être chargée dans le registre RDO isolément ou simultanément à une opération de chargement d'un des 16 registres généraux. Le chargement de RDO est commandé par M<sub>1</sub>.

Les signaux M<sub>2</sub> et M<sub>3</sub> commandent respectivement le positionnement du contenu de RDO sur le bus données (opération d'écriture en mémoire principale) ou l'enregistrement d'une donnée sur le bus, dans RDO (opération de lecture en mémoire principale). Pour la mémoire, M<sub>2</sub> correspond au signal RD et M<sub>3</sub> au signal WR (voir notamment la figure 4-6). Une donnée issue de la mémoire principale, stockée temporairement dans RDO, peut être utilisée comme l'une des entrées de l'UAL selon la sélection effectuée par le multiplexeur AMUX. C'est le signal A<sub>0</sub> qui précise l'origine de l'opérande fourni à l'UAL: soit RDO, soit le registre tampon A. RDO joue donc, également, un rôle de registre tampon d'opérande pour l'UAL. La micromachine de la figure 4-8 est semblable à celle de divers processeurs en tranche du commerce .

### 4.2.2. Les micro-instructions

La commande et le contrôle du chemin des données de la micromachine présentée à la figure 4-8 nécessitent 61 signaux. Nous avons divisé ces signaux en neuf groupes fonctionnels, soit :

- 16 signaux pour commander la copie du contenu d'un des seize registres sur le bus A,
- 16 signaux pour commander la copie du contenu d'un des seize registres sur le bus B,
- 16 signaux pour commander le chargement d'un des seize registres avec la donnée présente sur le bus C,
- 2 signaux de chargement des registres tampons A et B ( $L_0, L_1$ ),
- 2 signaux définissant l'opération à exécuter par l'UAL ( $F_0, F_1$ ),
- 2 signaux de commande du décaleur ( $S_0, S_1$ ),
- 4 signaux de commande de RAD et RDO ( $M_0, M_1, M_2, M_3$ ),
- 2 signaux d'indication d'opération de lecture et d'écriture avec la mémoire principale,
- 1 signal de commande du multiplexeur AMUX ( $A_0$ ).

Connaissant les valeurs de ces 61 signaux, on peut analyser le déroulement du cycle de base du chemin des données. Ce cycle fonctionnel consiste à effectuer plusieurs actions séquentielles (les unes à la suite des autres) : premièrement, placer des données sur les bus A et B, ensuite, stocker ces données dans les registres tampons A et B, puis, réaliser dans l'UAL une opération qui porte sur le contenu des registres tampons et faire subir éventuellement au résultat, en sortie de l'UAL, un décalage. Enfin, charger la donnée en sortie du décaleur dans le registre RDO et/ou dans l'un des 16 registres généraux de la micromachine. Simultanément, le registre RAD peut être chargé avec la donnée qui se trouve sur le bus B et un cycle de lecture en mémoire principale est alors générée.

En première approximation, considérons que la micromachine dispose d'un registre de commandes de 61 bits; chaque bit du registre est associé à un signal de commande. Un bit à 1 signifie que le signal associé est actif, un bit à 0 qu'il est au repos.

Au prix d'une faible augmentation du nombre de circuits, nous pouvons réduire le nombre de bits pilotant le chemin des données. En premier lieu, on dispose de 16 bits pour commander la copie d'un registre sur le bus A, cela équivaut à une potentielité de 2<sup>16</sup> registres sources, alors que seize registres seulement sont disponibles. En codant sur 4 bits la commande de sélection du registre à placer sur le bus A, on génère en sortie d'un décodeur

4 vers 16 associé, 16 signaux de commande, un par registre source. On procède de même pour le bus B.

Pour le bus C, la situation est un peu différente. En effet, le bus C transmet simultanément une donnée en entrée de tous les registres. Il est donc possible de charger plusieurs registres avec la donnée présente sur le bus C, à savoir tous ceux sélectionnés en mode chargement. Bien que cette opération soit possible, en pratique on ne la réalise pour ainsi dire jamais. C'est donc un seul des seize registres généraux qui reçoit la donnée présente sur le bus C. L'action de charger les registres est alors semblable à celle de les vider. Aussi, on code sur 4 bits la commande de direction de la donnée sur le bus C avec un codeur de type 4 vers 16, semblable aux précédents. Ainsi, notre micromachine utilise 4 décodeurs de type 4 vers 16, ce qui correspond à disposer de 12 bits. On a donc économisé  $3 \times 12 = 36$  bits; de ce fait, 25 bits ou signaux suffisent à piloter le chemin des données de notre micromachine.

Les signaux  $L_0$  et  $L_1$  valident les données des bus A et B respectivement dans les registres tampons A et B à des instants bien précis du cycle. C'est l'horloge du système qui fournit ces signaux quand il le faut, on peut s'en passer et l'on se retrouve avec 23 bits. Il est quelquefois utile de disposer d'un signal supplémentaire autorisant ou non le chargement d'un registre avec la donnée qui se trouve sur le bus C. Par exemple, supposons que l'on souhaite réaliser une opération avec l'UAL, simplement pour positionner les indicateurs N ou Z, sans vouloir tenir compte du résultat de l'opération qui se trouve forcément sur le bus C. Il suffit de ne pas générer d'ordre d'enregistrement vers les registres généraux. D'où la nécessité d'un signal de validation du chargement d'un registre que nous appelons VALC :  $VALC = 1$  pour autorisation, et  $= 0$  pour interdiction.

On constate qu'il est possible de piloter le chemin des données avec 24 bits ou signaux de commande. Notons que RD et WR peuvent très bien commander le chargement ou le vidage du registre RDO sur le bus données du système. Cette observation permet l'économie de deux autres bits; on passe alors de 24 à 22 bits pour commander le chemin des données de notre micromachine.

L'étape qui suit consiste à définir, partant des constatations ci-dessus, une micro-instruction d'un format de 22 bits. C'est elle qui sera porteuse des informations nécessaires à la génération des signaux de commandes ou *micro-commandes* du chemin des données. Nous présentons à la figure 4-9 la structure de la micro-instruction. Elle comprend plusieurs parties appelées champs. Chaque champ est associé à une fonction propre. Deux de ces fonctions n'ont pas encore été définies (nous le ferons plus loin dans l'ouvrage); il s'agit de COND et de ADDR. Ainsi la micro-instruction dispose de 13 champs, dont les onze qui suivent :

- AMUX : sélection de l'opérande gauche de l'UAL (1 bit)  
0 = registre tampon A, 1 = registre RDO
- UAL : choix de l'opération à réaliser (2 bits)  
0 = A+B, 1 = A ET B, 2 = A, 3 = A
- DCAL : sens du décalage réalisé par le décaleur (2 bits)  
0 = sans décalage, 1 = droite, 2 = gauche
- RDO : chargement de RDO avec la donnée sur le bus C (1 bit)  
0 = non, 1 = chargement
- RAD : chargement de RAD avec la donnée du registre tampon B (1 bit)  
0 = non, 1 = chargement
- RD : opération de lecture de la mémoire principale externe (1 bit)  
0 = non, 1 = lecture
- WR : opération d'écriture de la mémoire principale externe (1 bit)  
0 = non, 1 = écriture
- VALC : autorisation de chargement des registres généraux (1 bit)  
0 = interdiction, 1 = autorisation
- C : registre à charger avec la donnée sur le bus C (4 bits)  
0 = PC, 1 = AC, 2 = PP, etc.
- B : choix du registre à vider sur le bus B (4 bits)  
0 = PC, 1 = AC, 2 = PP, etc.
- A : choix du registre à vider sur le bus A (4 bits)  
0 = PC, 1 = AC, 2 = PP, etc.

La position des champs dans la micro-instruction est complètement arbitraire, elle n'a aucune incidence fonctionnelle. Nous l'avons choisie telle quelle est (figure 4-9) dans un souci de clarifier le schéma global de la micromachine présentée à la figure 4-10.

#### 4.2.3. Cycle d'exécution de la micro-instruction

Nous avons vu comment les informations portées par la micro-instruction pouvaient agir sur le chemin des données; nous allons maintenant examiner les différentes étapes du cycle fonctionnel d'exécution d'une micro-instruction.

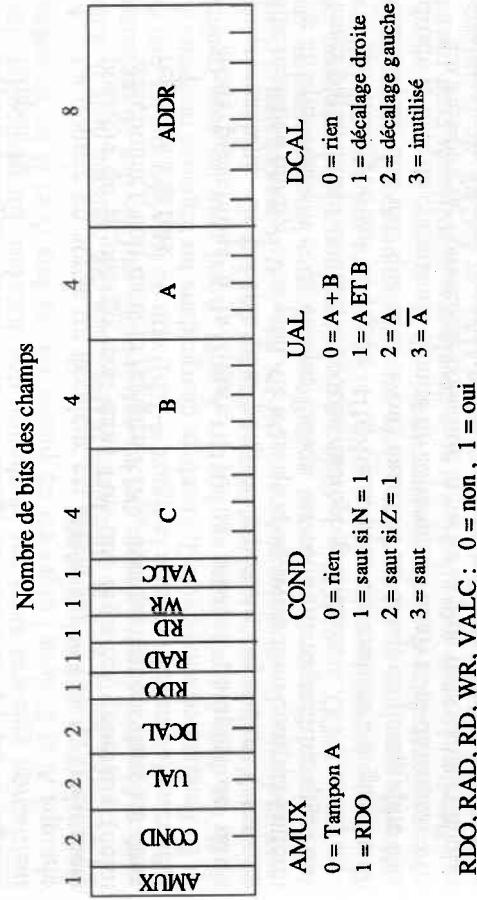


Figure 4-9. Structure par champs de la micro-instruction

Le cycle type de l'UAL, consiste à charger des opérandes dans les registres tampons A et B, à donner suffisamment de temps à l'UAL et au décaleur pour réaliser leur propre opération et à ranger le résultat d'opération dans un registre général ou dans le registre RDO. Il semble évident que l'enchaînement des ces diverses actions doit être séquentiel, c'est-à-dire qu'elles soient ordonnées dans le temps. Si l'on rangeait dans un registre général une donnée présente sur le bus C, en même temps qu'on vide deux registres sur les bus A et B et qu'on les enregistre dans les tampons A et B, il est fort probable que l'on chargerait dans A ou dans B une donnée erronée. Ce qui conduirait à un résultat faux. Pour éviter ces incidents malencontreux, il est nécessaire de disposer d'un circuit d'horloge à plusieurs phases déterminant des sous-cycles fonctionnels parfaitement ordonnés dans le temps (soit 4 sous-cycles fonctionnels, comme sur le schéma de la figure 4-5). Les actions entreprises dans les quatre sous-cycles fonctionnels sont :

1. Chargement de la micro-instruction qui doit être exécutée dans un registre particulier appelé *registre micro-instruction (RMI)*.
2. Transfert du contenu des registres sélectionnés sur les bus A et B, puis changement des données correspondantes dans les registres tampons A et B.
3. Puisque maintenant les entrées sont stables, l'UAL et le décaleur réalisent leur propre opération.

Si nécessaire, le registre RAD est chargé en même temps que le registre tampon B.

4. La donnée en sortie du décaleur est stable sur le bus C; il est alors possible de la charger soit dans l'un des registres généraux (celui sélectionné), soit dans le registre RDO, éventuellement dans les deux registres à la fois.

Nous présentons à la figure 4-10 un schéma complet de notre micromachine. Ce schéma peut paraître imposant, cependant il est très intéressant à étudier avec soin. Ce n'est qu'après avoir bien compris l'intérêt de chaque élément et de chaque liaison sur cette micromachine que vous aurez parfaitement assimilé la couche microprogrammée.

On distingue sur la figure 4-10 deux sous-ensembles : à gauche le chemin des données que nous avons examiné au début du paragraphe et à droite le bloc de commande (l'unité de commande) analysé ci-dessous.

La *mémoire de commande* constitue la partie la plus imposante et la plus importante du bloc de commande; c'est une mémoire très rapide qui contient les micro-instructions du microprogramme. Sur la plupart des machines, la mémoire de commande est une mémoire morte (à lecture seule); on l'appelle aussi mémoire de microprogramme. Sur d'autres machines, il s'agit d'une mémoire vive (à lecture/écriture) qui offre certains avantages, notamment celui de pouvoir modifier dynamiquement le microprogramme. Mais cette opération n'est pas une mince affaire!

Dans notre exemple, les micro-instructions sont codées avec des mots de 32 bits de large; la mémoire de commande peut contenir 256 micro-instructions, sa capacité est de  $256 \times 32 = 8192$  bits.

Comme toute mémoire, la mémoire de commande dispose d'un registre adressé et d'un registre de données. Nous appellerons *micro compteur ordinal* ou MCO le registre adressé. Étant donné que sa fonction consiste uniquement à pointer vers la prochaine micro-instruction à exécuter. Le registre de données constitue, quant à lui, le registre micro-instruction RMI que nous avons défini précédemment.

Nous tenons à préciser à nouveau que mémoire de commande et mémoire principale sont deux mémoires différentes qui n'ont rien de commun. La première contient le microprogramme, la seconde le programme écrit dans le langage se référant à la couche machine traditionnelle que nous étudierons au chapitre suivant.

En regardant la figure 4-10, on peut penser (étant donné la structure en boucle du bloc de commande) que la mémoire de commande passe son temps à transférer la micro-instruction adressée par le compteur MCO dans le registre RMI, et ainsi de suite. En fait il n'en est rien. En effet, ce n'est que pendant le sous-cycle 1 que le registre RMI reçoit la micro-instruction pointée par le MCO. Pendant les trois autres sous-cycles, le registre RMI n'est pas modifié, même si le compteur MCO est quant à lui normalement modifié pendant ces cycles. Pendant le sous-cycle 2, le contenu du registre

RMI est stable. Les informations contenues dans les champs de la micro-instruction agissent alors sur le chemin des données. En particulier, les champs A et B dont l'objet est de placer sur les bus A et B les données issues des registres sélectionnés.

Les décodeurs A et B effectuent le décodage des champs A et B. Les sorties des décodeurs agissent sur les entrées OE<sub>1</sub> et OE<sub>2</sub> des 16 registres généraux (comme le montre la figure 4-2(b)) afin de les vider sur les bus A et B. Pendant ce sous-cycle 2, l'horloge charge, dans les registres tampons A et B, les données présentes sur les bus A et B. Ainsi pour les sous-cycles suivants, les opérandes présentes sur les bus A et B sont chargées dans les registres tampons, un circuit spécifique effectue l'incrémentation du compteur MCO, soit MCO + 1. Cela permet de préparer le chargement de la micro-instruction suivante dans RMI, chargement qui sera effectif au début du prochain cycle machine. Cette anticipation permet d'augmenter la vitesse d'exécution des micro-instructions.

Pendant le sous-cycle 3, l'UAL et le décaleur sont sollicités pour traiter les opérations indiquées par les champs UAL et DCAL de la micro-instruction. Le champ AMUX, commandant le multiplexeur AMUX, définit la source de l'opérande transmis à l'entrée gauche de l'UAL (registre tampon A ou registre RDO); le registre tampon B, quant à lui, est toujours connecté à l'entrée droite de l'UAL. Le temps de calcul de l'UAL varie selon le type d'opération à réaliser. Il est maximum dans notre cas pour l'addition. En effet, dans ce cas, le temps de calcul est déterminé par le temps de propagation de la retenue et non pas par le délai de traversée des portes logiques de l'additionneur. Le temps de propagation de la retenue est proportionnel au nombre de bits des mots additionnés. Simultanément aux traitements de l'UAL et du décaleur, le registre RAD peut être chargé avec la donnée contenue dans le registre B, si le champ RAD le précise.

Au cours du sous-cycle 4 (dernier du cycle machine courant), la donnée qui se trouve sur le bus C est stable. Elle peut être chargée dans un des seize registres généraux et/ou dans le registre RDO, selon les indications fournies par les champs VALC et RDO de la micro-instruction. Le décodeur C est de type 4 vers 16, ses entrées sont : VALC, la ligne CK<sub>4</sub> de l'horloge et les quatre bits du champ C, il génère en sortie seize signaux pour la sélection des registres. Chaque entrée du décodeur (le champ C) est associée à une porte ET commandée par VALC et la ligne CK<sub>4</sub> de l'horloge. Ainsi, le chargement d'une donnée du bus C dans un registre n'est possible que si les trois conditions ci-dessous sont satisfaites :

1. VALC = 1,
2. le sous-cycle courant est le sous-cycle 4,
3. le registre est indiqué par le champ C.

Le registre RAD est également chargé pendant le sous-cycle 4 si le champ RAD est égal à 1.

Les deux signaux de commande, RD et WR, sont diffusés vers la mémoire principale et vers RDO, aussi longtemps qu'ils sont présents dans le registre RMI.

#### 4.2.4. Enchaînement des micro-instructions

Nous venons d'analyser le cycle d'exécution d'une micro-instruction. Voyons à présent comment se fait la sélection des micro-instructions à exécuter. Dans la plupart des cas, l'enchaînement des micro-instructions est séquentiel : la micro-instruction à exécuter se trouve normalement à l'adresse qui suit celle de la micro-instruction en cours de traitement. Dans certains cas, il est nécessaire de rompre l'exécution séquentielle des micro-instructions en introduisant dans la micro-instruction des ordres de rupture de séquence, conditionnelle ou non.

Dans notre micro-instruction, nous avons défini deux champs particuliers: ADDR, qui contient l'adresse de l'éventuelle micro-instruction suivante et COND, qui détermine le choix de la micro-instruction suivante: soit (MCO + 1), soit ADDR. Ainsi chaque micro-instruction dispose d'une commande possible de rupture de séquence : un branchement (JUMP ou saut) conditionnel ou non. Il est fréquent que l'on insère ces ordres de rupture de séquence dans la micro-instruction elle-même; bon nombre de micromachines en sont pourvues. Ceci permet un accès direct à deux successeurs potentiels. Cette alternative est plus rapide d'exécution qu'une autre où le choix est établi après traitement d'une micro-instruction de test conditionnel.

Sur notre micromachine, la sélection de la micro-instruction suivante est élaborée par un *microséquenceur* pendant le sous-cycle 4 alors que les bits N et Z sont en état stable en sortie de l'UAL. La sortie de ce microséquenceur est reliée à l'entrée d'un multiplexeur MMUX qui réalise le changement du MCO soit avec (MCO + 1), soit avec ADDR. Le champ COND de la micro-instruction définit quatre possibilités de choix de la micro-instruction suivante :

- COND = 0 pas de rupture de séquence, la prochaine micro-instruction est adressée par MCO + 1,
- COND = 1 branchement à l'adresse précisée par ADDR si N = 1,
- COND = 2 branchement à l'adresse précisée par ADDR si Z = 1,
- COND = 3 branchement à l'adresse précisée par ADDR sans condition.

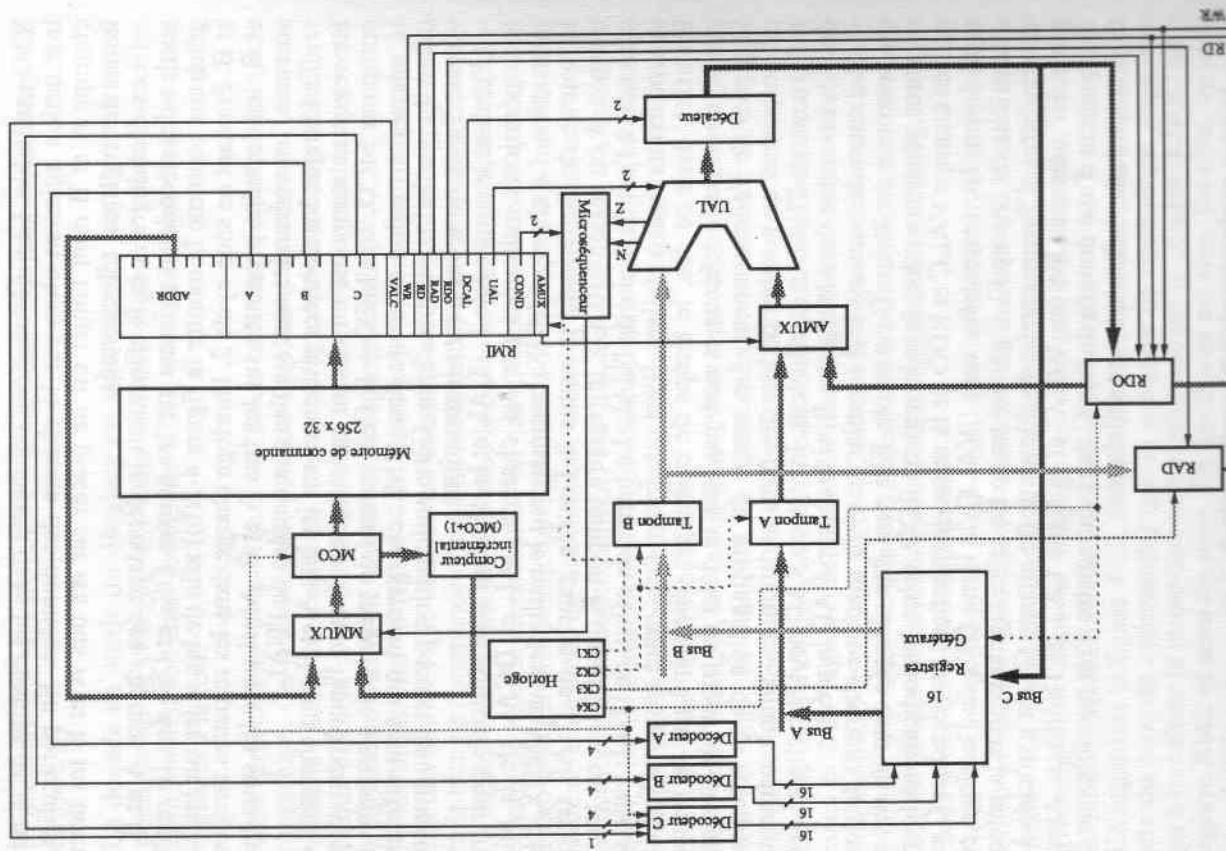


Figure 4-10. Un exemple d'architecture de micromachine

Les entrées du microséquenceur sont les bits N et Z de l'UAL et les deux bits du champ COND que nous appelons D (bit droit de COND) et G (bit gauche de COND). La sortie du microséquenceur agit sur la commande de sélection de MMUX, selon l'équation logique :

$$\text{MMUX} = \overline{G} \cdot \overline{DN} + G \cdot \overline{D} \cdot Z + GD = DN + GZ + GD$$

Ainsi, le signal de commande de MMUX est égal à 1 (réalisant le chargement de ADDR dans MCO) si  $GD = 01_2$  et  $N=1$ , ou  $GD = 10_2$  et  $Z=1$ , ou  $GD = 11_2$ . Dans les autres cas, la sortie du microséquenceur est égale à 0, ce qui entraîne une suite en séquence de la micro-instruction suivante (précisée par MCO + 1).

Typiquement, le microséquenceur est un circuit combinatoire; on pourrait le construire avec des circuits SSI comme celui de la figure 3-3(b) ou avec une partie d'un circuit PLA, comme sur la figure 3-16.

Pour que notre micromachine paraisse réaliste, il est nécessaire de préciser que le temps de cycle de la mémoire principale est supérieur à celui de la micro-instruction. En particulier, si la micro-instruction génère un cycle de lecture en mémoire principale, en activant RD=1, il faut que RD soit maintenu stable à 1 pendant l'exécution de la micro-instruction suivante. La donnée issue de la mémoire n'est valide et stable qu'après deux cycles de micro-instruction après l'activation de RD=1. Si la donnée sollicitée est nécessaire à la micromachine pour poursuivre le traitement du microprogramme, la micro-instruction qui suit n'aura pour effet que de maintenir RD stable pour garantir la lecture en mémoire principale. Cela se traduit par une perte de temps inévitable au niveau de la micromachine. Il en est de même d'une opération d'écriture en mémoire principale: la donnée doit être maintenue stable sur le bus pendant un temps suffisant à la mémoire pour l'enregistrer (soit deux micro-instructions).

Les programmeur de définir ses propres macro-instructions de langage d'assemblage. D'où la confusion à ne pas commettre. Pour éviter les confusions dans la suite de l'ouvrage, les instructions de la micromachine sont appelées *Mic-1* et celles de la micromachine *Mac-1*. Pour en revenir à nos macro-instructions, vues de la micromachine, ce sont les instructions classiques de la couche traditionnelle que nous appelons Mac-1, soit par exemple les instructions ADD, MOVE, JUMP, etc.

### 4.3.1. La structure de pile

On peut modéliser un système (notamment la micromachine) en décrivant son comportement au moyen d'un programme écrit par exemple dans un langage de haut niveau. Une partie importante de la description concerne l'adressage des opérandes. Pour illustrer ces propos, reportons-nous au programme Pascal de la figure 4-11(a). Ce programme a pour objet de réaliser le produit scalaire de deux vecteurs  $x$  et  $y$  de 20 éléments chacun. Dans le bloc principal du programme, les vecteurs sont initialisés à  $x_k = k$  et  $y_k = 2k + 1$ , puis le produit scalaire est calculé. Chaque fois qu'une multiplication est nécessaire, le programme principal fait appel à une fonction *pmul* qui effectue la multiplication. Nous supposons que le compilateur dont on dispose est peu performant, qu'il ne traduit qu'un sous-ensemble du langage Pascal et qu'il n'a pas, notamment, d'opérateur de multiplication. *Pmul* effectue la multiplication de deux nombres entiers positifs par additions successives. Supposons également que les deux nombres ne sont pas très grands.

Les langages à structures de blocs, comme Pascal, sont habituellement implémentés de telle façon qu'à l'issue du traitement d'une procédure ou d'une fonction, l'espace mémoire alloué pour stocker temporairement les variables locales est rendu au système (on dit aussi libéré). Une façon simple et efficace de réaliser ce mécanisme consiste à utiliser une structure de données particulière: la pile.

Concrètement, une *pile* est une structure mémoire dans laquelle les informations sont rangées de façon contiguë (comme une pile de linge dans une armoire). Un *pointeur de pile* (PP ou SP, *stack pointer*) indique le sommet de la pile, soit la position de la dernière donnée enregistrée dans la pile. (Sur certaines organisations de pile, on considère que PP indique l'emplacement où l'on doit enregistrer la donnée suivante.) La base de la pile est fixe; quand la pile est implantée dans la mémoire principale d'un ordinateur, la base correspond à une adresse fixe. La figure 4-12(a) présente une structure de pile qui comprend six mots mémoire. Le bas de la pile est à l'adresse 4020 (en notation décimale), PP pointe vers la dernière donnée enregistrée dans la pile, soit à l'adresse 4015. Dans ce cas particulier, la pile «grossit en taille» (évolue en nombre de mots stockés) des adresses hautes vers les adresses basses (en valeur d'adresses). Le choix inverse aurait été tout aussi bon: l'évolution ascendante ou descendante des adresses mémoire n'a aucun effet sur la pile.

## 4.3. UN EXEMPLE DE MACROMACHINE

Continuons notre cheminement dans la couche microprogrammée (couche 1) en regardant vers la couche supérieure (la couche traditionnelle) qui doit être interprétée par la micromachine de la figure 4-10. Par complémentarité avec la couche 1, on appelle l'architecture des couches 2 et 3, *macro-architecture ou macromachine*. Dans ce chapitre, on ignore volontairement la couche 3, étant donné que ses instructions sont quasiment les mêmes que celles de la couche 2. De plus, vues de la couche 1, les différences sont peu sensibles entre les couches 2 et 3. De façon similaire aux micro-instructions de la couche 1, nous avons choisi d'appeler les instructions de la macromachine, les *macro-instructions*. Toutefois, cette appellation ne doit pas être généralisée. En effet, il se trouve qu'en pratique la plupart des assembleurs (objets des couches supérieures) permettent au

## 246 Architecture de l'ordinateur

### La couche microprogrammée 247

```

program produit_scalaire (output);
{ce programme calcule le produit scalaire de 2 vecteurs x, y;
 x[1] * y[1] + x[2] * y[2] + ... + x[20] * y[20]
const max = 20;
type petit_entier = 0..100;
var array[1..max] of petit_entier;
var k : integer;
x, y : vec;
}

{multiplication des paramètres a et b
la multiplication est obtenue par additions successives}
function pmul (a, b : petit_entier) : integer;
var p, j : integer;
begin
if (a=0) or (b=0) then
  {0: réservoir 2 places dans la pile pour p et j}
  {1: si a ou b est à 0, le résultat vaut 0}
  {2: la fonction retourne 0}
else
  begin
    p := 0; {3: initialisation de p}
    for j := 1 to a do {4: addition de p+b à chaque passage}
      p := p + b; {5: addition effective}
      pmul := p {6: affectation du résultat à la fonction}
    end; {pmul}
  end; {produit interne de v et x et passage du résultat}
procedure p_scalaire (var v : vec; var resultat : integer);
var somme, i : integer;
begin
  somme := 0; {8: 2 places dans la pile pour somme et i}
  for i := 1 to max do {9: somme accumule le produit interne}
    somme := somme + pmul (x[i], v[i]); {10: calcul sur tous les éléments}
    resultat := somme {11: 1 élément dans somme}
  end; {p_scalaire}
  var somme, i : integer;
begin
  {programme principal} {14: 3 places dans la pile pour k, x et y}
  for k := 1 to max do {15: initialisation de la boucle}
    begin
      x[k] := k; {16: initialisation de x}
      y[k] := pmul (2, k) + 1 {17: initialisation de y}
    end;
    p_scalaire (y, k); {18: appel de p_scalaire}
    writeln (k); {19: impression du résultat}
  end.

```

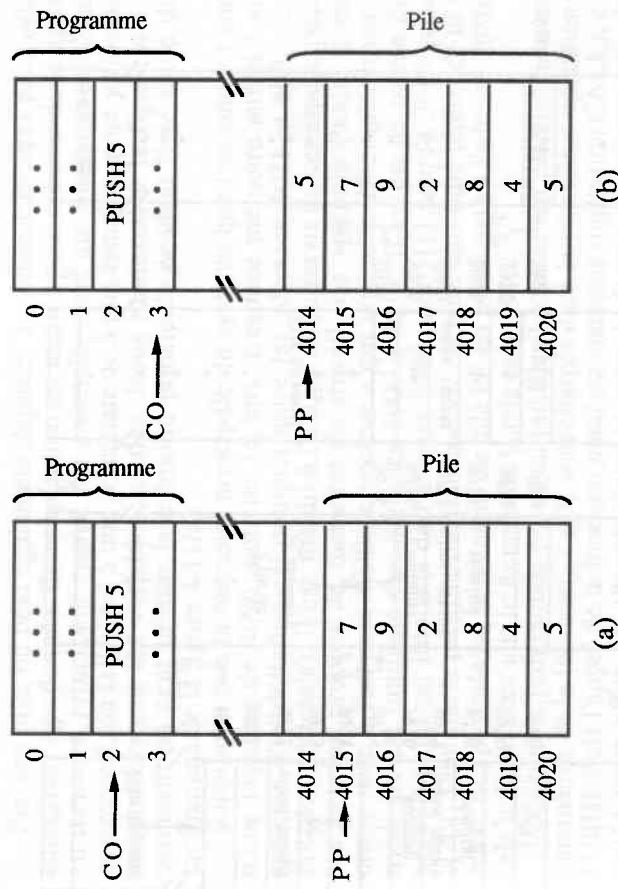
```

DEBUT: K = 4020, X = 4000, Y = 3980, A = 4, B = 3, P = 1, J = 0, V = 5
  RESULTAT = 4, SOMME = 1, I = 0
  JUMP PROGPAL /BRANCHEMENT AU PROGRAMME PRINCIPAL
  DESP 2
    LODL A
    INZE APAS0 /1
    LOC0 0 /2
    JUMP SUITE /RESULTAT NUL
    /AC := B
  LODL B
  INZE BPAS0 /JUMP VERS BPAS0 SI B<>0
  LOC0 0 /2
  JUMP SUITE /RESULTAT NUL
  APAS0: STOL P /3
  LOC0 1 /4
  STOL J /5
  LODL A /PREPARATION DU TEST DE BOUCLE
  JNEG L2 /JUMP VERS L2 SI A < 0, PAS DE BOUCLE
  JZER L2 /JUMP VERS L2 SI A = 0, PAS DE BOUCLE
  LODL P /6
  ADDL B /AC := P + B
  P := P + B /PREPARATION DU TEST DE FIN DE BOUCLE
  STOL P /7
  LOC0 1 /8
  ADDL J /AC := J + 1
  STOL J /9
  SUBL A /AC := J - A
  JNEG L1 /JUMP VERS L1 SI J < 0
  JZER L1 /JUMP VERS L1 SI J = A
  LODL P /10
  INSP 2 /11
  RETN /RETOUR
  PSCAL: DESP 2 /8
  LOC0 0 /9
  STOL SOMME /SOMME := 0
  LOC0 1 /10
  STOL I /11
  LODL X-1 /AC := X + I - 1
  ADDL I /PUSH X[I]
  PSHI LODL V /AC := ADRESSE DU VECTEUR
  ADDL I /AC := V + 1
  SUBD C1 /V COMMENCE A 1, PAS A 0
  PSHI CALL PMUL /PUSH V[I]
  INSP 2 /PMUL (X[I], V[I])
  ADDL SOMME /LIBERATION DE LA PILE
  STOL SOMME /AC := SOMME + PMUL(..)
  LOC0 1 /SOMME := SOMME + PMUL(..)
  /PREPARATION DU TEST DE FIN DE BOUCLE

```

Figure 4-11(a). Programme Pascal réalisant le produit scalaire de deux vecteurs

Figure 4-11(b). Programme «produit scalaire» en langage d'assemblage (début)

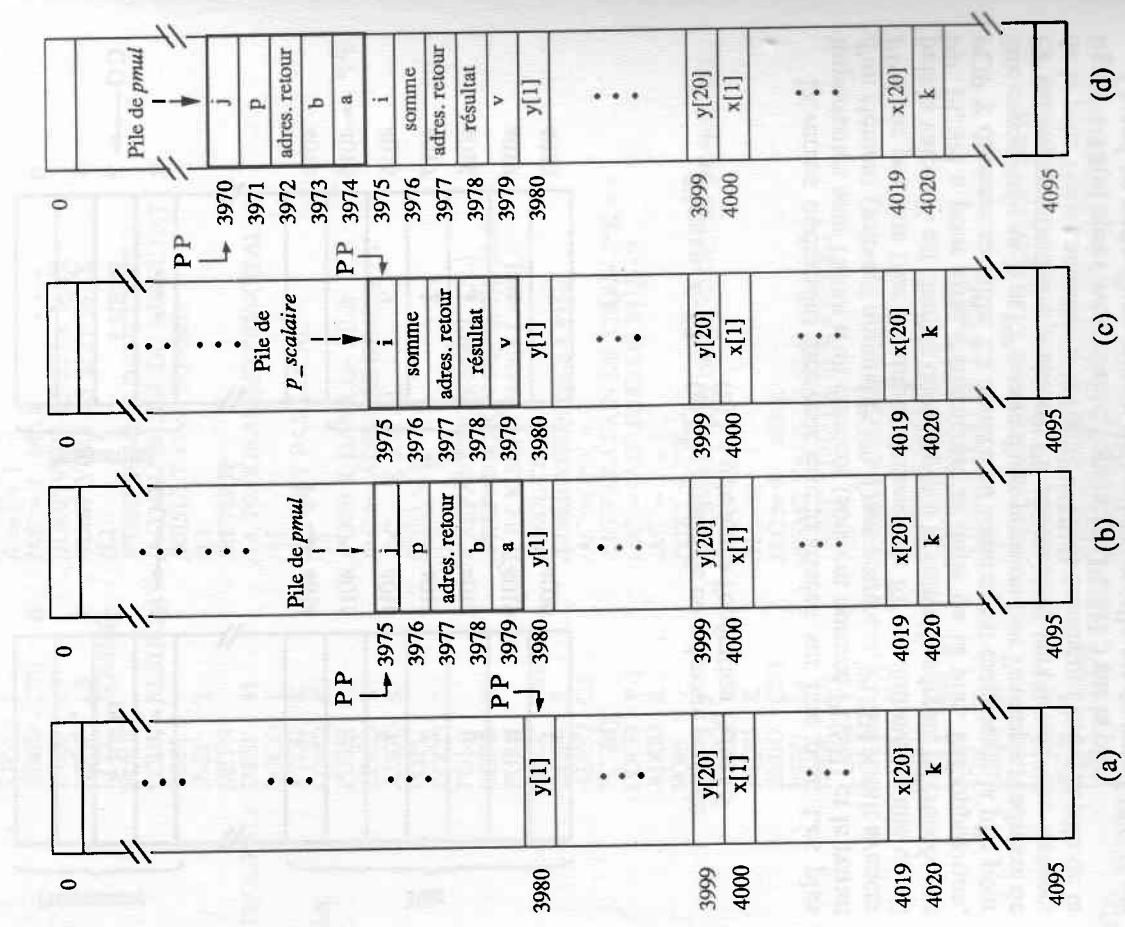


**Figure 4-12.** (a) Pile implantée en mémoire centrale  
 (b) La pile après l'opération PUSH

Diverses opérations peuvent être réalisées sur une pile. Les plus importantes sont l'ajout d'un élément (souvent nommé PUSH) et le retrait d'un élément (souvent nommé POP). Dans notre cas, PUSH X fait avancer PP d'une position (sur l'exemple, l'adresse de PP est diminuée d'une unité) puis la valeur X est chargée dans la pile à la nouvelle adresse référencée par PP. PUSH a pour effet d'augmenter la taille de la pile. Par opposition, POP Y diminue sa taille. La donnée au sommet est extraite de la pile pour être enregistrée en Y et PP recule d'une position (sur l'exemple, l'adresse de PP est augmentée d'une position); PP pointe alors vers le nouveau sommet de la pile. Nous montrons à la figure 4-12(b) comment évolue la pile de la figure 4-12(a) après avoir réalisé l'opération PUSH 5 sur la pile. Quand on parle d'une pile on dit souvent qu'elle est de type *LIFO* (*Last*

*In First Out*) : la dernière donnée enregistrée dans la pile sera la première à en sortir.

**Figure 4-11(b).** Programme «produit scalaire» en langage d'assemblage (fin)



**Figure 4-13. Evolution de la pile pendant l'exécution du produit scalaire:**  
 (a) Pendant le programme principal (b) Pendant *pmul* (c) Pendant *p\_scalaire* (d) Pendant *pmul* dans *p\_scalaire*

Sur une pile, on peut également déplacer le pointeur PP sans pour cela enregistrer une donnée. Cette opération est utilisée pour réservé de la place en mémoire principale quand, par exemple, dans un programme Pascal notamment, on fait appel à une procédure ou à une fonction, pour y stocker temporairement les variables locales. Nous montrons à la figure 4-13 comment on peut utiliser la mémoire principale pendant l'exécution du programme de la figure 4-11(a).

Supposons que la mémoire principale du système que l'on analyse soit d'une capacité de 4096 mots de 16 bits. L'espace mémoire attribué au système d'exploitation est compris entre les adresses 4021 et 4095 (en notation décimale). Il est interdit à l'usager de se servir de cet espace pour ses besoins propres de mémoire de travail. La variable Pascal *k* est enregistrée à l'adresse 4020, le vecteur *x* qui nécessite 20 emplacements mémoire est implanté aux adresses 4000 à 4019. Il en est de même du vecteur *y* qui est implanté de l'adresse 3980 pour *y[1]* jusqu'à 3999 pour *y[20]*. Tant que le programme principal est exécuté sans faire référence ni à *pmul*, ni à *p\_scalaire*, le pointeur de pile PP est positionné à l'adresse 3980; cela indique alors que le sommet de la pile est à 3980.

Dès que le programme principal fait appel à *pmul*, en premier lieu les paramètres de la fonction, 2 et *k*, sont chargés dans la pile (deux opérations PUSH). Puis l'appel de la fonction *pmul* est entrepris (opération CALL), ce qui a pour effet de charger dans la pile l'adresse de retour vers le programme principal, pour permettre sa reprise quand *pmul* est terminée. Dès le début d'exécution de *pmul*, le pointeur de pile est positionné à l'adresse 3977. La première des actions de *pmul* consiste à réserver deux places dans la pile pour y stocker les variables locales *p* et *i*, le pointeur de pile PP est déplacé de deux positions et pointe alors vers l'adresse 3975 (figure 4-13(b)). Ainsi, ces cinq informations en mémoire principale (de 3979 à 3975) constituent la pile propre de *pmul*. Elle est libérée dès que la fonction *pmul* est terminée; PP se retrouve alors à 3980.

Aux adresses 3979 et 3978 de la figure 4-13(b), on lit *a* et *b*; ce sont les noms des paramètres formels de *pmul*. En fait, lors de son exécution, on trouve dans la pile respectivement les valeurs 2 et *k*.

Quand *pmul* est terminée et que la procédure *p\_scalaire* est appelée, la configuration de la pile en mémoire principale est celle de la figure 4-13(c). Lorsque la procédure *p\_scalaire* appelle elle-même la fonction *pmul*, la configuration de la pile devient celle de la figure 4-13(d).

Nous prenons maintenant conscience d'un problème majeur. Quel code doit générer le compilateur pour permettre à la fonction *pmul* d'accéder à ses paramètres et variables locales?

Supposons que l'action de ligne *p* soit réalisée par une instruction telle que LOAD 3976. Alors *pmul* peut accéder à *p* à partir du programme principal mais pas depuis la procédure *p\_scalaire*. L'instruction LOAD 3971 permet à *pmul* d'accéder à *p* depuis *p\_scalaire* mais plus depuis le programme principal. En fait, pour que *pmul* puisse accéder correctement à ses

paramètres et variables locales lors de chaque appel, il convient que la demande d'accès mémoire soit générée en référence à la position courante du pointeur de pile PP. Dans l'exemple, pour obtenir  $p$ , il faut aller chercher en mémoire la donnée située à l'adresse PP+1. Pour qu'une Mac-1 puisse accéder à une information dans la pile, elle doit disposer d'un mode d'adressage qui se réfère à une distance X relative à la position courante du pointeur de pile PP. Nous appelons ce mode d'adressage le mode local.

### 4.3.2. Jeu d'instructions de la macromachine

Le programmeur de la couche 2 voit la macromachine comme un système qui dispose d'une mémoire de 4096 mots de 16 bits (soit 8 Koctets) et de 3 registres fonctionnels : un compteur ordinal CO, un pointeur de pile PP et un accumulateur AC. Pour le moment, considérons l'accumulateur comme un registre tampon d'usage général; nous verrons sa spécificité plus loin dans l'ouvrage. Trois modes d'adressage sont associés aux instructions Mac-1 : direct, indirect et local. En mode d'adressage direct, les instructions disposent d'une adresse absolue sur 12 bits (ceux de poids faible) qui correspond à l'adresse effective de la donnée. Ces instructions sont fort utiles pour accéder aux variables globales du programme, comme par exemple  $x$  sur la figure 4-11(a). En mode d'adressage indirect, il est nécessaire de faire un calcul d'adresse, de placer le résultat dans l'accumulateur AC pour obtenir l'adresse effective du mot mémoire à lire ou écrire. Ce mode d'adressage est fort utilisé; il permet notamment d'accéder aisément à des tableaux ou vecteurs implantés en mémoire.

Le mode d'adressage local, propre à notre machine, spécifie l'accès à une information dans la pile, relativement à la position courante du pointeur de pile PP. La valeur portée par l'instruction spécifie la valeur du déplacement relatif à PP. On l'utilise dans l'exemple de la figure 4-11(b) pour accéder aux variables locales dans la pile.

Nous montrons figure 4-14 le jeu d'instructions Mac-1. Chaque instruction contient un code opération et quelquefois une adresse mémoire ou une constante. Le tableau comprend quatre colonnes : celle la plus à gauche correspond à la configuration binaire des instructions; la deuxième fournit le code symbolique ou mnémone de l'instruction, celui que l'on utilise quand on écrit des programmes en langage d'assemblage; la troisième colonne spécifie le nom de l'instruction et la dernière, à droite, sa fonctionnalité décrite en langage Pascal. Dans cette dernière colonne,  $m[x]$  signifie que l'on fait référence au mot mémoire d'adresse  $x$ .

L'instruction LODD X effectue le chargement de l'accumulateur AC en mode d'adressage direct, avec le mot mémoire d'adresse X dont la valeur est spécifiée sur 12 bits. L'instruction LODL X réalise en mode d'adressage local le chargement de AC avec le mot mémoire situé dans la pile à une distance X du pointeur de pile PP.

Binaire	Symbole	Instruction	Opération
0000 xxxx xxxx xxxx	LODD	Chargement mode direct	ac := m[x]
0001 xxxx xxxx xxxx	STOD	Rangement mode direct	m[x] := ac
0010 xxxx xxxx xxxx	ADDD	Addition mode direct	ac := ac + m[x]
0011 xxxx xxxx xxxx	SUBD	Soustraction mode direct	ac := ac - m[x]
0100 xxxx xxxx xxxx	IPOS	Saut si > 0	if ac > 0 then co := x
0101 xxxx xxxx xxxx	JZER	Saut si = 0	if ac = 0 then co := x
0110 xxxx xxxx xxxx	JUMP	Branchement	co := x
0111 xxxx xxxx xxxx	LOCO	Chargement constante	ac := x (0 ≤ x ≤ 4095)
1000 xxxx xxxx xxxx	LODL	Chargement mode local	ac := m[pp + x]
1001 xxxx xxxx xxxx	STOL	Rangement mode local	m[x + pp] := ac
1010 xxxx xxxx xxxx	ADDL	Addition mode local	ac := ac + m[pp + x]
1011 xxxx xxxx xxxx	SUBL	Soustraction mode local	ac := ac - m[pp + x]
1100 xxxx xxxx xxxx	INEG	Saut si < 0	if ac < 0 then co := x
1101 xxxx xxxx xxxx	JNZE	Saut si ≠ 0	if ac ≠ 0 then co := x
1110 xxxx xxxx xxxx	CALL	Appel procédure	pp := pp-1; m[pp] := co; co := x
1111 0000 0000 0000	PSHI	PUSH indirect	pp := pp-1; m[pp] := nl[ac]
1111 0010 0000 0000	POPI	POP indirect	m[ac] := m[pp]; pp := pp+1
1111 0100 0000 0000	PUSH	PUSH	pp := pp-1; m[pp] := ac
1111 0110 0000 0000	POP	POP	ac := m[pp]; pp := pp+1
1111 1000 0000 0000	RETN	Retour	co := m[pp]; pp := pp+1
1111 1010 0000 0000	SWAP	Echange AC et pile	tmp := ac; ac := pp; pp := tmp
1111 1100 0000 0000	INSP	Incrémentation de PP	pp := pp+y (0 ≤ y ≤ 255)
1111 1110 0000 0000	DESP	Décrémentation de PP	pp := pp-y (0 ≤ y ≤ 255)

xxxx xxxx xxxx : adresse absolue sur 12 bits  
xxx yyyy yyyy : constante sur 8 bits  
x et y ci-dessus sont des variables

Figure 4-14. Jeu d'instructions Mac-1

LODD, STOD, ADDD et SUBD sont quatre instructions de base associées au mode d'adressage direct. Les instructions LODL, STOL, ADDL et SUBL réalisent les mêmes fonctions mais en mode d'adressage local.

On distingue cinq instructions de branchement ou de rupture de séquence: une inconditionnelle (JUMP) et quatre conditionnelles (JPOS, JZER, JNEG et JNZE). L'instruction JUMP charge sans condition le compteur ordinal CO avec l'adresse spécifiée par les 12 bits du champ adresse de l'instruction. Les autres instructions de branchement réalisent la même action mais seulement lorsque la condition est satisfaite.

LOCO X charge la valeur constante X dans l'accumulateur AC; X est codée sur 12 bits, ce qui correspond aux valeurs  $0 \leq X \leq 4095$ .

PSHI transfère dans la pile le mot mémoire dont l'adresse est indiquée dans l'accumulateur AC; il s'agit du mode d'adressage indirect. L'opération inverse, POPI, extrait la donnée au sommet de la pile et la range en mémoire à l'adresse précisée par AC. PUSH et POP sont des instructions classiques de transfert de données entre l'accumulateur AC et la pile.

SWAP est une instruction qui effectue la permutation des contenus de AC et de la donnée au sommet de la pile.

INSP Y et DESP Y réalisent l'incrémentation ou la décrémentation de PP d'une valeur Y codée sur 8 bits, soit  $0 \leq Y \leq 255$ .

Enfin, CALL est une instruction d'appel de sous-programme ou de procédure. Au début de l'exécution de CALL, l'adresse de retour vers le programme interrompu est sauvegardée automatiquement dans la pile. L'instruction RETN restitue automatiquement l'adresse de retour dans le CO, ce qui permet la reprise du programme interrompu.

Notre machine ne dispose pas d'instructions spécifiques d'E/S. Les communications avec l'extérieur sont réalisées selon le même formalisme que les communications avec la mémoire principale. On dit que les E/S sont mappées en mémoire (dans l'espace mémoire). Par exemple, une opération de lecture à l'adresse 4092 correspond à une lecture de données sur un périphérique d'entrée. Le mot mémoire d'adresse 4093 est, par exemple, le registre d'état du périphérique. Quand un caractère est disponible en 4092, le bit 15 du mot d'état 4093 est actif. Ainsi, la routine de lecture de données d'un périphérique commence par tester le bit 15 du mot d'état. Tant que ce bit 15 est au repos, rien ne se passe, il y a attente. Dès que le bit 15 est actif, le caractère est transféré du mot 4092 (le périphérique) dans l'accumulateur AC.

Une routine de sortie de données vers un périphérique se comporte de façon similaire. Tout d'abord, la donnée est envoyée vers la sortie (mot mémoire d'adresse 4094), puis le bit 15 du registre d'état du périphérique (mot mémoire d'adresse 4095) est activé pour signifier au périphérique la disponibilité de l'information.

Comme exemple d'utilisation des instructions Mac-1, nous avons écrit un programme en langage d'assemblage (figure 4-11(b)). Ce programme résulte de la compilation du programme Pascal de la figure 4-11(a). Nous

avons supposé pour des raisons pédagogiques que le compilateur n'est pas très performant, surtout quant à l'optimisation du code généré. Dans les deux programmes, les chiffres 0 à 19 en commentaires sont volontaires; ils permettent au lecteur de faire facilement le lien entre les deux programmes. CALL OUTNUM1 est un appel d'une routine de sortie de données décrite dans la bibliothèque de programmes. CALL STOP fait également appel à une routine particulière, qui a pour effet d'indiquer à l'assembleur la fin physique des instructions du programme.

#### 4.4. UN EXEMPLE DE MICROPROGRAMME

Nous avons spécifié la micro puis la macro-architecture; il nous reste à voir comment un programme exécuté par la micromachine interprète celui de la macromachine. Avant de répondre à cette question, examinons avec soin le langage dont on doit disposer pour écrire le microprogramme.

##### 4.4.1. Langage de micro-assemblage

En principe, on écrit le microprogramme en binaire. Mais c'est là une opération fort complexe, en raison de la largeur souvent importante de la micro-instruction. Pour simplifier l'écriture, on utilise un *langage symbolique* ou *mémorique*. Une façon de faire consiste, par exemple, à n'écrire qu'une seule micro-instruction par ligne de programme en nommant chaque champ non nul à l'aide de sa valeur propre. Dans le cas de notre micro-instruction (figure 4-9), pour réaliser l'addition de AC et de A puis ranger le résultat dans AC, on pourrait écrire :

VALC = 1, C = 1, B = 1, A = 10

Bon nombre de langages d'écriture de microprogrammes procèdent de la sorte. Toutefois cette notation est lourde et peu aisée à manipuler. Une autre façon de faire, plus efficace, consiste à utiliser une notation semblable à celle d'un langage de haut niveau, mais en conservant l'idée de n'écrire qu'une seule ligne de programme source par micro-instruction. Il est concevable d'écrire le microprogramme avec un langage de haut niveau standard, mais comme en microprogrammation l'efficacité est primordiale, nous lui préférerons un langage de type symbolique ou d'assemblage. Une instruction de ce langage correspond à une micro-instruction machine. Nous appellerons ce langage *Langage de Micro-Assemblage* (LMA), notre intention étant de disposer d'un LMA qui ressemble à Pascal et qui permette, entre autres, d'adopter les conventions d'écriture usuelles de Pascal. Toutes les instructions sont écrites en lettres minuscules, certains mots clés sont en caractères gras. Par exemple, en LMA, la micro-instruction que nous avons présentée plus haut devient `ac := a + ac.`

Pour indiquer les fonctions 0, 1, 2 et 3 de l'UAL, on peut écrire par exemple,

```
ac := a+ac; a := bet(ri, pmasq); ac := a; a := inv(a);
```

respectivement : bet réalise le ET logique et inv l'inversion. Les opérations de décalage sont decalq pour décalage à gauche et decald pour décalage à droite. Par exemple,

```
rit := decalg(rit+rit);
```

place le contenu du registre rit sur les bus A et B de la micromachine, réalise l'addition dans l'UAL, fait subir un décalage à gauche au résultat de l'addition et range le résultat final dans le registre rit.

Le branchement inconditionnel est précisé par l'instruction goto, les branchements conditionnels sont réalisés par tests de N et Z en sortie de l'UAL. Par exemple,

```
if n then goto 27;
```

On peut combiner sur une même ligne source une instruction de traitement, une instruction de manipulation de données et un branchement. Toutefois, un problème survient quand on veut simplement tester un registre sans transférer son contenu dans un autre registre par une opération de chargement (nous avons mentionné cette hypothèse précédemment). Comment doit-on spécifier le registre à tester? Pour résoudre ce problème, nous définissons une pseudo-variable ual qui n'a pour effet que de tester le contenu de l'UAL. Par exemple,

```
ual := rit; if n then goto 27;
```

signifie que rit est simplement testé par l'UAL, sans autre action (code opération = 2). Seuls les bits n et z sont positionnés suite à cette opération. Remarquons que l'utilisation de ual précise que VALC = 0.

Pour indiquer une opération d'écriture ou de lecture vers la mémoire principale, on écrit simplement sur la ligne source wr ou rd.

Sur la ligne de programme source, l'ordre d'écriture des actions ou instructions en langage LMA est quelconque. Cependant, pour améliorer la lisibilité du programme, nous les positionnons selon leur place respective dans la micro-instruction. La figure 4-15 donne quelques exemples de lignes d'instructions en LMA en correspondance avec la micro-instruction.

	ADD	C	VALC	CAL	CONF	AMUX	AC	DCAL	RD	RI	CO	AA
rad:= co; rd;	0 0 2 0 0 1 1 0 0 0 0 0 0											
rd;	0 0 2 0 0 0 1 0 0 0 0 0 0											
ri:= rdo;	1 0 2 0 0 0 0 0 1 3 0 0 0											
co:= co + 1;	0 0 0 0 0 0 0 0 0 1 0 6 0											
rad:= ri; rdo:= ac; wr;	0 0 2 0 1 1 0 1 0 0 3 1 00											
ual:= rit; if n then goto 15;	0 1 2 0 0 0 0 0 0 0 0 4 15											
ac:= inv(rdo);	1 0 3 0 0 0 0 0 0 1 1 0 0											
rit:= decalg(rit); if n then goto 25;	0 1 2 2 0 0 0 0 0 1 4 0 25											
ual:= ac; if z then goto 22;	0 2 2 0 0 0 0 0 0 0 0 1 22											
ac:= bet(r1,amasq); goto 0;	0 3 1 0 0 0 0 0 0 1 1 8 300											
pp:= pp + (-1); rd;	0 0 0 0 0 0 1 0 1 2 2 7 00											
rit:= decalg(ri+ri); if n then goto 69;	0 1 0 2 0 0 0 0 0 1 4 3 3 69											

Figure 4-15. Exemples d'instructions en LMA et micro-instructions associées

#### 4.4.2. Le microprogramme

Nous en sommes à présent au point de pouvoir écrire le microprogramme qui, exécuté par la micromachine, interprète les Mac-1. Ce programme est présenté à la figure 4-16; il comprend 79 lignes d'instructions écrites en LMA.

L'affectation des registres généraux de la micromachine (figure 4-8) est maintenant évidente: CO, AC et PP sont les trois registres de base de la micromachine référencés par les Mac-1. RI est le registre instruction, il contient l'instruction Mac-1 en cours d'exécution. RIT contient une copie temporaire de RI et sert à décoder le code opération de la Mac-1. Les registres +1, -1 et 0 contiennent les valeurs constantes indiquées. AMASQ est un registre masque d'adresse qui permet d'extraire les 12 bits de poids faible de la partie adresse d'une Mac-1; sa configuration est 0FFF (en notation hexadécimale). PMASQ est le registre masque de pile, sa configuration est 00FF; on l'utilise pour séparer des instructions INSP et DESP la valeur du déplacement codée sur 8 bits. Les six autres registres (A à F) n'ont pas d'affectation particulière, leur usage est général.

Comme tout interpréteur, le microprogramme comprend une partie principale, une boucle, qui a pour rôle d'aller chercher les instructions de niveau 2 du programme à interpréter (en mémoire centrale), de les analyser et de lancer leur exécution. Exammons les séquences d'instructions en LMA.

du microprogramme de la figure 4-16, dont le rôle consiste à interpréter le programme de la figure 4-11(b).

Le microprogramme commence à la ligne 0 (figure 4-16). La première action entreprise consiste à aller chercher l'instruction référencée par le CO. En attendant cette instruction, le CO est incrémenté d'une unité tandis que le signal RD est diffusé en permanence vers la mémoire principale. Quand l'instruction sollicitée est disponible, elle est enregistrée dans RI (ligne 2); simultanément, l'instruction est analysée, son bit de poids fort (le bit 15) est testé. Si le bit 15 vaut 1, le microprogramme continue en ligne 28, sinon il se poursuit séquentiellement en ligne 3.

Supposons que l'instruction reçue soit LODD. A la ligne 3, on teste le bit 14 de l'instruction pendant que RIT est chargé avec l'instruction qui subit au passage un décalage à gauche de 2 bits (un par l'UAL et un par le décaleur). Le bit indicateur N, positionné et testé en sortie de l'UAL, correspond bien au bit 14 de l'instruction après décalage de 2 bits.

Les instructions Mac-1 qui ont 00 comme valeur binaire des bits de plus fort poids, voient leur bit 13 testé en ligne 4. Les instructions commençant par 000 sont analysées en ligne 5 et celles qui commencent par 001 le sont en ligne 11. En ligne 5, la micro-instruction force VALLC = 0; en effet, seul le contenu de RIT doit être testé sans qu'il y ait ni modification, ni transfert de RIT vers un autre registre. En sortie, ce test (bit N de l'UAL) détermine s'il s'agit d'une instruction LODD ou STOD; les actions respectives sont alors entreprises.

Dans le cas de LODD, la première action assurée consiste à aller chercher le mot mémoire dont l'adresse est chargée dans RAD, celle directement portée par les 12 bits de poids faible de RI. Le registre RAD étant un registre de 12 bits, les quatre bits de poids fort de RI (le code opération) n'affectent en rien le choix du mot mémoire sollicité en lecture. A la ligne 7, aucune action n'est entreprise; seul le maintien de RD actif est assuré. Quand le mot attendu est disponible dans RDO, il est copié dans AC (ligne 8), puis le microprogramme reprend à la ligne 0; l'instruction LODD est terminée. Les instructions STOD, ADDD et SUBD ont un comportement similaire. Toutefois, on peut se demander comment l'UAL réalise l'opération de soustraction. En fait, la soustraction est réalisée par une addition avec un nombre négatif représenté en complément à 2 :

$$x - y = x + (-y) = x + (\overline{y} + 1) = x + 1 + \overline{y}$$

La soustraction effective est réalisée en ligne 18 après que la donnée à soustraire ait subi une inversion logique (ligne 17) suite à l'opération + 1 (ligne 16). A la ligne 16, on fait AC + 1 et on attend d'obtenir la donnée sollicitée en mémoire centrale avant de poursuivre (tout comme on le fait à la ligne 13).

La génération des microcommandes nécessaires à l'exécution de l'instruction JPOS commence en ligne 21. Si AC < 0, le branchement échoue, JPOS se termine alors immédiatement par un retour à la ligne 0. En revanche, si AC ≥ 0, l'adresse de branchement sur 12 bits est extraite du registre RI par une opération ET logique entre ce registre et le registre AMASQ (dont le contenu est 0FFF). Le résultat de cette opération est transféré dans le CO; puis JPOS se termine par retour à la ligne 0. Cela ne coûterait rien de ré-initialiser le CO à cet endroit, c'est pourquoi nous l'avons fait. Si cela avait nécessité une micro-instruction supplémentaire, il aurait fallu être prudent pour éviter des problèmes ou erreurs d'adressage causés par la mise à 0 des quatre bits de poids fort du CO.

D'une certaine façon, JZER (ligne 23) réalise un traitement opposé à JPOS. Avec JPOS (ligne 21), si la condition de test est satisfait (N), le branchement échoue. En revanche, avec JZER, si la condition est satisfait (Z), le branchement a lieu. Quelle que soit l'instruction JUMP, conditionnelle ou non, les microcommandes nécessaires à entreprendre le branchement sont les mêmes (ligne 22); aussi, quand c'est possible, la suite logique des instructions JUMP passe par la ligne 22. Cette technique de programmation, bien que proscrire dans les programmes d'applications, est largement utilisée en microprogrammation, toujours dans un souci d'amélioration des performances.

Les instructions JUMP et LOCO sont simples et évidentes, elles ne nécessitent pas d'examen particulier.

LODL présente quelque intérêt. Tout d'abord, pour obtenir l'adresse de la donnée à lire, il est nécessaire de la calculer en ajoutant au pointeur de pile PP la valeur du déplacement contenue dans le deuxième champ de l'instruction LODL. Une opération de lecture mémoire est alors générée. Ensuite, l'exécution de l'instruction est semblable à LODD, et se poursuit aux lignes 7 et 8.

Les instructions STOL, ADDL et SUBL ont des comportements semblables à STOD, ADDD et SUBD. De même, JNEG et JNZE sont respectivement semblables à JZER et JPOS.

Pour l'instruction CALL, le pointeur de pile commence par être décrémenté d'une unité puis l'adresse de retour est sauvegardée dans la pile. Enfin, un branchement vers la procédure appelée est entamé (lignes 47 à 49). Les lignes 49 et 22 sont presque les mêmes (wr les différences). Si elles étaient strictement identiques, nous aurions éliminé la ligne 49 en plaçant en ligne 48 un saut inconditionnel vers la ligne 22. A cet endroit, on peut maintenir wr actif pour entreprendre la recherche d'une autre micro-instruction.

Toutes les instructions restantes ont leurs bits de plus fort poids égaux à 1111. Nous verrons plus loin leur décodage d'adresse. Quant à la description de leur exécution, elle est suffisamment évidente pour que l'on ne s'y attarde pas davantage.

```

0: rad:= co; rd;
1: co:= co+1; rd;
2: ri:= rdo; if n then goto 28; {programme principal}
   {incrémentation du CO}
   {analyse et sauvegarde RDO}
3: rit:= decalg(ri+ri); if n then goto 19; {000x ou 001x?}
4: rit:= decalg(rit); if n then goto 11; {0000 ou 0001?}
5: ual:= rit; if n then goto 9; {0011 = STOL}
6: rad:= ri; rd;
7: rd;
8: ac:= rdo; goto 0; {0000 = LODD}
9: rad:= ri; rdo:= ac; wr; {0001 = STOD}
10: wr; goto 0; {0010 ou 0011?} {110x ou 111x?}
11: ual:= rit; if n then goto 15; {0010 = ADDD}
12: rad:= ri; rd; {1100 = JNEG}
13: rd;
14: ac:= rdo+ac; goto 0; {1101 = JNZE}
15: rad:= ri; rd;
16: ac:= ac+1; rd;
17: a:= inv(rdo);
18: ac:= ac+a; goto 0; {0011 = SUBD}
19: rit:= decalg(rit); if n then goto 25; {010x ou 011x?}
20: ual:= rit; if n then goto 23; {0100 ou 0101?} {1110 = CALL}
21: ual:= ac; if n then goto 0; {0110 = JPOS}
22: co := bet(ri,amasq); goto 0; {0111 000 = PSH}
23: ual: ac; if z then goto 0; {0101 = JZER}
24: goto 0; {1111 001 = POP}
25: ual:= rit; if n then goto 27; {0110 ou 0111?} {1111 010 = PUSH}
26: co:= bet(ri,amasq); goto 0; {0110 = JUMP}
27: ac:= bet(ri,amasq); goto 0; {0111 = LOCO}
28: rit:= decalg(ri+ri); if n then goto 40; {10xx ou 11xx?}
29: rit:= decalg(rit); if n then goto 35; {100x ou 101x?}
30: ual:= rit; if n then goto 33; {1000 ou 1001?} {1111 011 = POP}
31: a:= ri+pp; {1000 = LODL}
32: rad:= a; rd; goto 7; {1001 = LSDL}
33: a:= ri+pp; {1000 = STOL}
34: rad:= a; rdo:= ac; wr; goto 10; {1010 ou 1011?}
35: ual:= rit; if n then goto 38; {1010 = ADDL}
36: a:= ri+pp; {1011 = SUBL}
37: rad:= a; rd; goto 13; {1011 = STOL}
38: a:= ri+pp; {110x ou 111x?}
39: rad:=a; rd; goto 16; {1100 ou 1101?} {1110 = JNEG}
40: rit:= decalg(rit); if n then goto 46; {1110 = JNZE}
41: ual:= rit; if n then goto 44; {1110 = CALL}
42: ual:= ac; if n then goto 22; {1110 = JNEG}
43: goto 0; {1110 = CALL}
44: ual:= ac; if z then goto 0; {1110 = JNZE}
45: co:= bet(ri,amasq); goto 0; {1110 = CALL}
46: rit:= decalg(rit); if n then goto 50; {1110 = CALL}
47: pp:= pp+(-1); {1110 = CALL}
48: rad:= pp; rdo:= co; wr; {1110 = CALL}
49: co:= bet(ri,amasq); wr; goto 0; {1110 = CALL}
50: rit:= decalg(rit); if n then goto 65; {1111, adresse?}
51: rit:= decalg(rit); if n then goto 59; {1111, adresse?}
52: ual:= rit; if n then goto 56; {1111, adresse?}
53: rad:= ac; rd; {1111 000 = PSH}
54: pp:= pp+(-1); rd; {1111 000 = PSH}
55: rad:= pp; wr; goto 10; {1111 000 = PSH}
56: rad:= pp; pp:= pp+1; rd; {1111 001 = POP}
57: rd; {1111 001 = POP}
58: rad:= ac; wr; goto 10; {1111 010 = PUSH}
59: ual:= rit; if n then goto 62; {1111 011 = POP}
60: pp:= pp+(-1); {1111 011 = POP}
61: rad:= pp; rdo:= ac; wr; goto 10; {1111 010 = PUSH}
62: rad:= pp; pp:= pp+1; rd; {1111 010 = PUSH}
63: rd; {1111 010 = PUSH}
64: ac:= rdo; goto 0; {1111 010 = PUSH}
65: rit:= decalg(rit); if n then goto 73; {1111 011 = POP}
66: ual:= rit; if n then goto 70; {1111 011 = POP}

```

Figure 4-16. Exemple de microprogramme en LMA (1)

Figure 4-16. Exemple de microprogramme en LMA (2)

```

67: rad:= pp; pp:= pp+1; rd;
68: rd;
69: co:= rdo; goto 0;

70: a:= ac;
71: ac:= pp;
72: pp:= a; goto 0;

73: ual:= rit; if n then goto 76;
74: a:= bet(ri, smasq);
75: pp:= pp+a; goto 0;

76: a:= bet(ri, smasq);
77: a:= inv(a);
78: a:= a+1; goto 75;

```

Figure 4-16. Exemple de microprogramme écrit en LMA (fin)

#### 4.4.3. Remarques sur le microprogramme

Bien que nous ayons détaillé le microprogramme, nous souhaitons y faire quelques remarques. Dans le microprogramme, le CO est incrémenté à la ligne 1. Il aurait très bien pu être incrémenté à la ligne 0. Ainsi, la ligne 1 aurait eu simplement comme effet le maintien de RD actif et pendant ce temps, la micromachine serait en attente dans un état oisif. Sur les machines réelles, on utilise volontiers une telle opportunité pour réaliser une action transparente au microprogramme, par exemple le rafraîchissement des mémoires RAM dynamiques. Si nous laissons la ligne 1 telle qu'elle est, on peut améliorer la vitesse d'exécution de la micromachine en modifiant, par exemple, la ligne 8 comme suit :

```
8: rad:= co; ac:= rdo; rd; goto 1;
```

En d'autres termes, on commence le cycle de recherche de la prochaine macro-instruction à exécuter avant que celle en cours d'exécution ne soit terminée. Cette façon de procéder représente une forme primitive de machine pipe-line. On pourrait également appliquer ce mécanisme à d'autres routines d'exécution d'instructions.

Dans notre approche, il est évident qu'une part importante du temps d'exécution d'une macro-instruction est consacrée à son décodage bit par bit, afin de déterminer son type. Cette observation nous conduit à envisager qu'il serait utile de charger directement le MCO sous contrôle du microprogramme. En pratique, les ordinateurs du marché disposent dans leur micromachine d'une unité spécialisée de décodage du code opération de la macro-instruction Mac-1. Ainsi, le type de la Mac-1 est déterminé immédiatement, ce qui a pour effet de forcer directement dans le MCO

l'adresse de début de la routine d'exécution correspondante; cela correspond à un branchement multidirectionnel. Si, par exemple, on effectuait un décalage à droite de 9 bits du registre RI, que l'on mettait à zéro les 9 bits de poids fort de RI et que l'on plaçait le résultat obtenu dans MCO, on obtiendrait un branchement multidirectionnel vers 128 directions possibles, de 0 à 127. Chacun des mots associés contiendrait la première micro-instruction de la macro-instruction correspondante. Bien que cette technique de branchement gaspille quelque peu la mémoire microprogramme, elle améliore nettement les performances de la micromachine; elle est donc fréquemment mise en œuvre en pratique.

Nous n'avons pas dit grand-chose des opérations d'E/S car, dans notre cas, les E/S sont implantées en mémoire centrale et pour la micromachine, il n'y a pas de différence entre la mémoire et les entrées/sorties.

#### 4.4.4. Perspectives

Avant de poursuivre, prenons un temps de réflexion. L'idée de base en microprogrammation est de disposer d'une machine logique séquentielle simple. Dans notre exemple, elle correspond à un groupe de registres, une mémoire de commande de type ROM, un additionneur, un compteur incrémental, un décaleur et des circuits combinatoires nécessaires au multiplexage, au décodage et au séquencement. Avec cette machine logique, il est possible de réaliser un interpréteur programmé des instructions de la couche 2 traditionnelle. Au moyen d'un compilateur, on traduit un programme source écrit dans un langage de haut niveau (par exemple Pascal), en un autre programme constitué d'instructions de base de la couche 2. Ces instructions (les Mac-1) sont exécutées les unes à la suite des autres par l'interpréteur de la couche microprogrammée.

La couche 2 sert dans ce cas d'interface entre le compilateur et l'interpréteur. Toutefois, ce n'est pas toujours le cas. Sur certaines machines, le compilateur fournit directement un code exécutable, sans passer par la génération d'un langage intermédiaire. Mais dans ce cas, le nombre d'instructions exécutables est plus important (selon le degré d'optimisation du compilateur), d'où une nette augmentation des besoins en place mémoire. Dans notre exemple, chaque macro-instruction est codée sur 16 bits. En s'affranchissant de son décodage, l'exécution de la macro-instruction nécessite en moyenne l'enchaînement de quatre micro-instructions de 32 bits. Si l'on compilait un programme en langage de haut niveau, destiné à être directement exécuté par le niveau 1 (la micromachine), alors le besoin en mémoire de microprogramme augmenterait fortement (au moins d'un facteur de l'ordre de 8). Dans ce cas, le besoin supplémentaire de mémoire de microprogramme, forcément de type RAM, entraînerait une sérieuse augmentation de son coût, d'autant plus qu'elle doit être très rapide. L'alternative qui consiste à utiliser la mémoire centrale comme mémoire de microprogramme n'est pas envisageable car dans ce cas, on obtiendrait une machine très lente.

A partir de ces remarques, on voit pourquoi les machines, pensées et conçues selon une approche fonctionnelle en couches imbriquées, gagnent en simplicité et en efficacité de conception. Dans cette approche, chaque couche traite d'un certain niveau d'abstraction. A la couche 0, le concepteur logicien cherche par divers moyens (choix technologiques, algorithmes de calcul, etc.) à gagner des nanosecondes afin de rendre les circuits de traitement plus performants. A la couche 1, le concepteur de la micromachine et du microprogramme cherche quant à lui la meilleure structuration de la micro-instruction, afin d'utiliser au mieux les circuits de la couche 0, en allant vers une simultanéité poussée des actions entreprises (parallelisme des traitements). En réalité, chaque couche a ses propres objectifs; sa façon de résoudre ses problèmes techniques, les performances des services rendus, la qualité de ses relations avec les couches adjacentes, etc. Chaque couche participe à sa manière au fonctionnement de la machine. En décomposant l'ensemble des problèmes de conception d'un système en plusieurs sous-problèmes disjoints, on arrive à maîtriser bien mieux la complexité des ordinateurs modernes.

## 4.5. ÉTUDE DE LA COUCHE MICROPROGRAMMÉE

Comme toute chose en architecture, la conception de la micromachine doit être entreprise avec beaucoup d'attention. Les performances en rapidité d'exécution d'un ordinateur y sont étroitement liées. Dans ce chapitre, nous étudions en détail les particularités de la micromachine.

### 4.5.1. Microprogrammation horizontale et verticale

La différence majeure entre ces deux techniques de microprogrammation dépend de l'encodage de la micro-instruction. Si l'on construisait un circuit VLSI contenant les micro-instructions Mic-1, on pourrait faire abstraction des registres, de l'UAL, et des autres composants logiques pour ne s'intéresser qu'aux portes externes. Le fonctionnement de la micromachine est régi par la diffusion d'un certain nombre de signaux de commande, comme par exemple les seize signaux OE assurant le vidage des registres A sur le bus A, ou les signaux de sélection d'opération de l'UAL. En revanche, en regardant dans l'UAL on constate que les opérateurs internes sont commandés par quatre lignes (et non par deux) émanant d'un décodeur 2 vers 4 situé dans le coin bas, à gauche, du circuit UAL de la figure 3-20. C'est ainsi que, dans chaque micromachine un ensemble de  $n$  signaux de commandes sont appliqués à des endroits précis et à des instants déterminés pour assurer directement, sans décodage préalable, le fonctionnement de la machine.

Cette façon de faire conduit à une micro-instruction au format différent des instructions classiques : elle comprend  $n$  bits de large, soit un bit par microcommande. Les micro-instructions conçues selon ce format sont

appelées *horizontales*, elles se situent à l'une des extrémités du spectre des formats possibles des micro-instructions. A l'autre extrémité de ce spectre, on trouve des micro-instructions disposant de peu de bits (très inférieur à  $n$ ) et structurées en champs fortement encodés; on dit alors qu'elles sont *verticales*. En résumé on peut dire que la microprogrammation horizontale nécessite un petit nombre de micro-instructions très larges et qu'en revanche la microprogrammation verticale nécessite un grand nombre de micro-instructions plutôt étroites. En pratique on rencontre des formats mixtes, situés entre les deux extrêmes ci-dessus. Par exemple, celui que nous avons établi pour notre micromachine (figure 4-9). La micro-instruction comprend des bits associés directement aux microcommandes RAD, RDO, AMUX, RD, WR, etc., qui ont un effet immédiat sur les circuits de la micromachine. D'autre part, elle comprend des champs de bits codés comme UAL, A, B ou C, qui nécessitent l'usage d'un décodeur avant d'être appliqués sur les circuits correspondants.

Binaire Symbole Instruction Opération

0000	ADD	Addition	$r1 := r1 + r2$
0001	AND	ET Logique	$r1 := r1 \text{ ET } r2$
0010	MOVE	Déplacement	$r1 := r2$
0011	COMPL	Complément	$r1 := \text{inv}(r2)$
0100	DECALG	Décalage à gauche	$r1 := \text{decalg}(r2)$
0101	DECALD	Décalage à droite	$r1 := \text{decald}(r2)$
0110	GETRDO	Vider RDO dans R1	$r1 := rdo$
0111	TEST	Test registre	<b>if</b> $r2 < 0$ <b>then</b> $n := \text{true}$ <b>if</b> $r2 = 0$ <b>then</b> $z := \text{true}$
1000	DEBRD	Début RD	$rad := r1; rd$
1001	DEBWR	Début WR	$rad := r1; rdo := r2; wr$
1010	CONRD	Continuer RD	$rd$
1011	CONWR	Continuer WR	$wr$
1100			
1101	NJUMP	Saut si N = 1	<b>if</b> $n$ <b>then</b> <b>goto</b> $r$
1110	ZJUMP	Saut si Z = 1	<b>if</b> $z$ <b>then</b> <b>goto</b> $r$
1111	UJUMP	Saut inconditionnel	<b>goto</b> $r$

$$x = 16 * r1 + r2$$

Figure 4-17. Les micro-instructions Mic-2

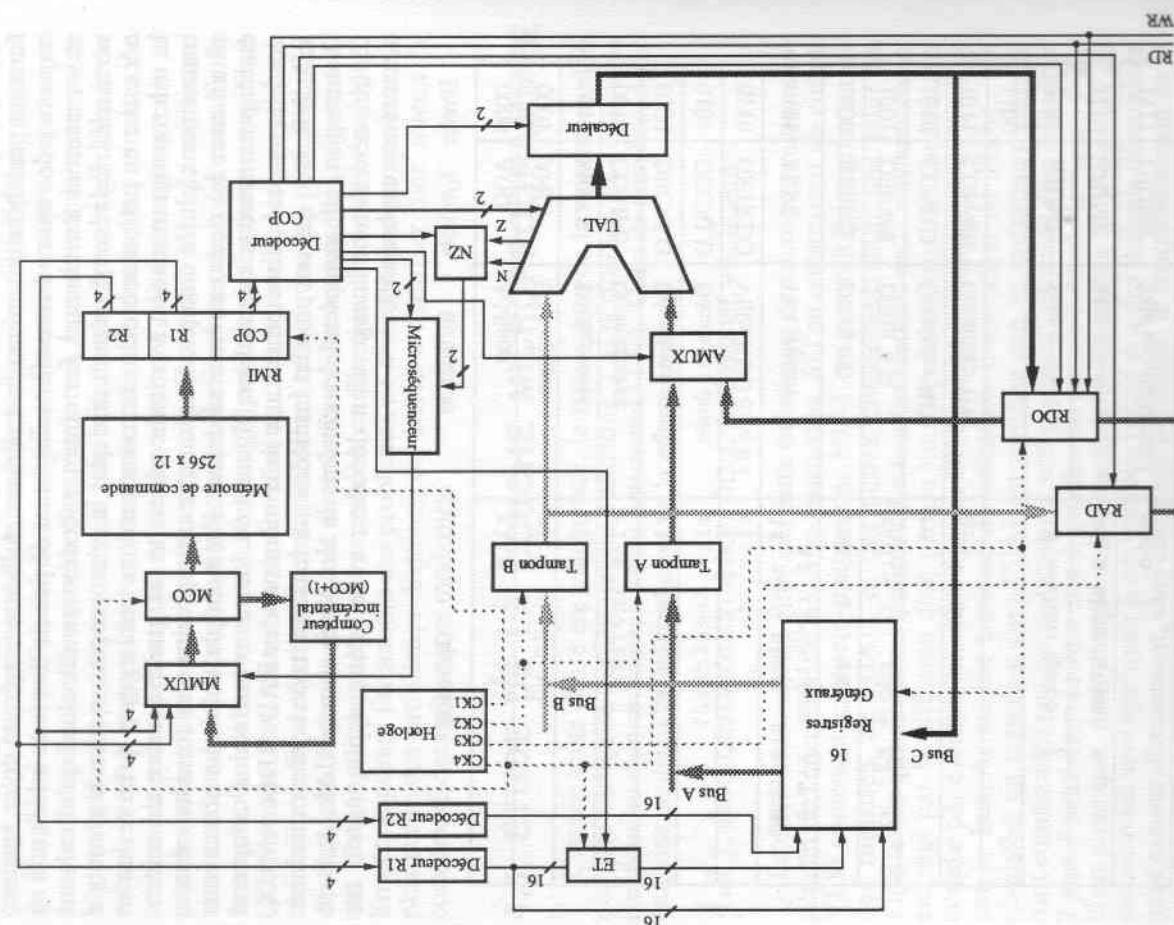
Une micro-instruction typique comprend un champ code d'opération qui peut être vu comme une généralisation de notre champ UAL et un ou plusieurs champs opérandes, comme par exemple A, B et C. Avec cette organisation, des codes opérations seraient nécessaires pour les lectures et écritures en mémoire principale, les branchements, etc., car il n'y a pas de champs spécifiques à ces actions sur notre micromachine.

Pour clarifier au mieux la distinction entre microprogrammation horizontale et verticale, nous allons revoir notre micromachine en utilisant cette fois des micro-instructions verticales. Chacune de ces nouvelles micro-instructions comprend trois champs de 4 bits, soit un total de 12 bits contre 32 bits dans la version précédente. Le premier champ est le code opération (COP) qui précise ce que la micro-instruction doit faire. Les deux autres champs sont les registres R1 et R2. Les micro-instructions de branchement (JUMP) regroupent R1 et R2 pour former un champ de 8 bits (soit R). Une micro-instruction typique «ADD PP, AC» signifie que PP doit être additionné à AC et que le résultat doit être rangé dans PP. Ces micro-instructions que nous appelons Mic-2 (par opposition aux Mic-1 précédentes) sont présentées à la figure 4-17. Remarquons que chaque Mic-2 ne réalise qu'une seule opération, par exemple la Mic-2 d'addition qui effectue  $R1 := R1 + R2$ , mais qui ne réalise ni décalage, ni chargement de RAD, ni tout ce qui ne réfère pas à l'addition. En fait les 12 bits de la micro-instruction suffisent simplement à spécifier une seule opération à la fois.

instruction suffisent à spécifier une seule opération à la fois.

Pour refléter les micro-instructions Mic-2, reconstruisons la machine originale de la figure 4-10 pour obtenir une nouvelle micromachine, celle de la figure 4-18. Les chemins des données des deux machines sont identiques. La plupart des éléments de l'unité de commande sont semblables (ceux sur la partie droite du schéma). En particulier MCO, MMUX, le compteur incrémental et le microséquenceur sont strictement identiques. On n'utilise que deux décodeurs 4 vers 16 pour R1 et R2, à la place des trois décodeurs A, B et C de la figure 4-10.

Trois éléments sont nouveaux sur la figure 4-18 : le bloc ET (en haut), le bloc NZ et le décodeur COP (en bas). Le bloc ET regroupe 16 portes ET permettant au champ R1 (à travers le décodeur R1) de commander à la fois le vidage d'un registre sur le bus A et son chargement par le bus C. Bien évidemment, ces opérations ne peuvent être simultanées, d'où la nécessité des portes ET en sortie du décodeur R1 qui valident le chargement du registre quand l'autorisation est délivrée par le décodeur COP et que le sous-cycle 4 est générée. Le contenu du registre sélectionné par R1 est positionné sur le bus A au début du sous-cycle 2, puis il est chargé dans le registre tampon A. C'est au début du sous-cycle 3 que les données sont présentes dans les registres tampons A et B. Il est donc important de ne pas charger le registre sélectionné avec la donnée sur le bus C pendant qu'éventuellement on le vide par ailleurs sur le bus A ou B. Le chargement d'un registre ne peut intervenir qu'au début du sous-cycle 4 quand le décodeur COP l'y autorise. Cette dernière action est semblable à la commande VALC de la micromachine précédente. En fait, les 16 signaux de commande du



**Figure 4-18.** Une micromachine à micro-instructions verticales

chargement des registres tampons A et B sont actifs dans les mêmes conditions que sur la micromachine précédente.

Le bloc NZ est un registre à deux bits qui enregistre les signaux N et Z en sortie de l'UAL, quand il en reçoit l'ordre du décodeur COP. On utilise cet artifice, car sur la micromachine de la figure 4-18, l'UAL ne réalise qu'une seule opération pendant une micro-instruction. Les bits N et Z ne sont testés et utilisés qu'à la micro-instruction suivante, après avoir été chargés dans le registre NZ. L'UAL est en effet un circuit combinatoire qui ne dispose pas de registres internes. Il est donc nécessaire de conserver les valeurs courantes de N et Z si l'on ne veut pas les perdre. C'est le rôle du registre NZ.

L'élément central de la nouvelle micromachine est constitué du décodeur COP, qui analyse le code opération de la micro-instruction et précise les actions à entreprendre. Ce décodeur reçoit les 4 bits du champ code opération et distribue 13 microcommandes distinctes vers le bloc ET, le microséquenceur, le registre NZ, AMUX, IUAL, le décaleur, RAD, RDO, RD et WR. Les microcommandes vers l'UAL, le décaleur et le microséquenceur sont les mêmes que celles de la micromachine originale, soit deux signaux par fonction.

Pour chacun des seize codes opérations possibles de la micro-instructions Mic-2, il est nécessaire de déterminer lesquelles des 13 microcommandes en sortie du décodeur COP doivent être actives (= 1) ou laissées dans un état de repos (= 0). On obtient alors un tableau à  $16 \times 13$  éléments qui fournit les valeurs des 13 signaux de commandes pour chacune des 16 micro-instructions Mic-2 générées. Nous présentons ce tableau à la figure 4-19. Les colonnes du tableau précisent le nom des micro-commandes. Les suffixes H et L (pour haut et bas) associés aux commandes UAL, MS et DECAL correspondent aux signaux de sélection des unités fonctionnelles : UAL, microséquenceur et décaleur. Par exemple, prenons la micro-instruction DEBRD qui initialise un cycle de lecture mémoire. Cette Mic-2 utilise la fonction 2 de l'UAL, soit UALH = 1 et UALB = 0, puis charge le registre RAD avec la sortie du décaleur et active RD. Le décaleur ne pratique quant à lui aucun décalage (DECALH = 0 et DECALB = 0). Toutes les autres commandes sont au repos.

Considérons les fonctions de branchement JUMP. Comme nous avons décidé d'être compatibles avec le microséquenceur de la micromachine originale, nous utilisons MSH et MSB pour indiquer qu'il y a un branchement conditionnel si N est actif (01) ou si Z est actif (10) ou qu'il y a un branchement inconditionnel (11). Aucun autre code opération ne génère de branchement, donc MSH et MSB valent 00.

Voyons à présent comment construire le décodeur COP. Il dispose de 4 entrées (les 4 bits du code opération) et de 13 sorties (les 13 signaux de commandes). On peut utiliser un ou plusieurs circuits PLA ou une mémoire PROM ou ROM pour réaliser ce décodeur. La figure 4-19 est une certaine forme de représentation d'une table de vérité des 13 fonctions de quatre variables (une fonction par colonne) dont les numéros de ligne définissent les valeurs des quatre variables. De ce point de vue, la question de savoir comment construire le décodeur se réduit à savoir comment implémenter la table de vérité. L'idéal serait un circuit PLA à 4 entrées et 13 sorties. Si ce circuit n'existe pas on peut utiliser, par exemple, trois circuits 74S330 à 12 entrées et 6 sorties (comme celui de la figure 3-13). Si l'on nomme les quatre bits du code opération A, B, C et D (du bit de poids fort au bit de poids faible), on obtient par exemple,

Figure 4-19. Les signaux de commandes associés aux micro-instructions Mic-2; le signe «+» signifie l'état actif de la microcommande et un «blanc» qu'elle est au repos.

Code opération Micro-instructions	UALH	UALB	DECALH	DECALB	AMUX	NZ	RD	RDO	RD	WR	MSH	MSB
0 ADD												
1 AND	+				+		+					
2 MOVE	+				+		+					
3 COMPL	+	+			+		+					
4 DECALG	+	+					+	+				
5 DECALD	+	+	+				+	+				
6 GETRDO	+	+	+				+	+				
7 TEST	+	+										
8 DEBRD	+						+	+				
9 DEBWR	+						+	+	+			
10 CONRD	+						+	+				
11 CONWR	+						+	+				
12 Inutile												
13 NJUMP	+									+		
14 ZJUMP	+									+		
15 UJUMP	+									+		

$$\begin{aligned} UALB &= \overline{A} \overline{B} \overline{C} D + \overline{A} \overline{B} C D = \overline{A} \overline{B} D \\ DECALH &= \overline{A} B \overline{C} \overline{D} \end{aligned}$$

$$\begin{aligned} \text{RAD} &= A \overline{B} \overline{C} \overline{D} + A \overline{B} \overline{C} D = A \overline{B} \overline{C} \\ \text{MSH} &= A \overline{B} \overline{C} \overline{D} + A B C D = A B C \end{aligned}$$

C'est donc au maximum quinze termes produits qui doivent être générés, car la combinaison ABCD n'existe pas. Après avoir reconstruit l'architecture matérielle de la machine, il nous faut récrire le microprogramme. Nous l'avons fait à la figure 4-20. Volontairement, nous avons conservé les mêmes numéros de labels (d'étiquettes) que ceux du microprogramme original (0 à 78), pour faciliter les comparaisons entre les deux programmes. On aurait pu écrire directement le microprogramme en notation typique du langage d'assemblage avec les symboles des Mic-2 de la figure 4-17; mais, pour faciliter sa lecture et sa compréhension nous l'avons écrit en langage LMA. Remarquez que les instructions en LMA de la forme `ua1 := reg utilisent la micro-instruction TEST pour valider les bits N et Z.`

Assurez-vous de bien comprendre les relations entre le microprogramme en binaire, tel qu'il se présente dans la mémoire de microprogramme et sa version écrite en LMA à la figure 4-20. Le microprogramme de la figure 4-20 ne comprend qu'une seule opération par ligne source de langage LMA. De ce fait il est plus lisible et plus simple que l'original (figure 4-16). Chaque ligne source du programme original a été décomposée en deux, trois ou quatre lignes source dans le second programme. Une autre conséquence de la microprogrammation verticale conduit à une augmentation sensible du nombre de micro-instructions; cela provient de l'absence d'instructions Mic-2 à trois champs d'adresses. Voyez notamment les micro-instructions des lignes 22 et 27 du microprogramme de la figure 4-16.

Le microprogramme de la figure 4-16 comprend 79 lignes d'instructions LMA, soit 79 micro-instructions de 32 bits, au total 2528 bits en mémoire de microprogramme. Le second microprogramme comprend 160 micro-instructions de 12 bits, soit 1920 bits au total. Cette différence du nombre de bits représente un gain de 24% de capacité pour la mémoire de commande, d'où une diminution de son coût dans le deuxième cas.

L'inconvénient majeur de l'approche micro-instruction verticale concerne principalement le nombre important de micro-instructions à exécuter pour interpréter une macro-instruction. Il s'ensuit une diminution très nette de la rapidité de traitement de la micromachine. De ce fait, sur les gros ordinateurs performants et rapides, on utilise une microprogrammation horizontale poussée. En revanche, sur les machines plus modestes, on préfère la microprogrammation verticale pour des raisons de simplicité et de coût de conception.

L'existence de micro-instructions fortement encodées, comme les Mic-2, nous amène à réfléchir sur la microprogrammation. En effet, les Mic-2 de la figure 4-17 sont tout à fait semblables aux Mac-1 de la couche traditionnelle de micro ou mini-ordinateurs. Le PDP-8 par exemple, qui est un mini-ordinateur à mots mémoire de 12 bits, dispose d'un jeu d'instructions dont certaines ne sont pas très différentes quant à leurs caractéristiques et à leurs

performances de celles de nos Mic-2. Considérons que la partie la plus significative des micro-instructions est réalisée avec des circuits logiques (par exemple un PLA pour le décodeur COP). On peut très bien dire, dans ce cas, que la micromachine relative aux Mic-2 n'est pas microprogrammée mais câblée. En effet, elle se comporte comme une machine logique interprétant les instructions d'une autre machine. Si le microprogramme d'une micromachine verticale était stocké en mémoire centrale (comme c'était le cas sur l'IBM 370/145) alors la distinction entre une machine à haut degré de microprogrammation verticale et une même machine câblée serait plus évidente. Des informations supplémentaires sur les techniques de codage et de parallélisme des micro-instructions peuvent être obtenues en lisant l'article de Dasgupta (1979).

#### 4.5.2. Nanoprogrammation

Les concepts présentés ci-dessus reposent sur la coexistence d'une mémoire principale (qui contient les instructions de la couche 2) et d'une mémoire de commande (qui contient le microprogramme). Une troisième mémoire, la *nanomémoire*, permet dans certains cas une association optimale de la microprogrammation verticale et horizontale. La *nanoprogrammation* correspond, d'une certaine façon, à l'optimisation du microprogramme dans le cas où certaines micro-instructions sont souvent sollicitées. Ce n'est pas le cas du microprogramme de la figure 4-16 où la micro-instruction la plus fréquente contient uniquement rd comme élément commun; de plus elle ne se présente que cinq fois.

Nous présentons à la figure 4-21 le concept de nanoprogrammation. Dans le cas (a), le microprogramme comprend  $n$  micro-instructions de  $w$  bits de large et la capacité de la mémoire de commande est de  $nw$  bits. Supposons qu'une analyse détaillée du microprogramme fasse apparaître que, parmi les  $2^w$  possibilités de représentation, on utilise seulement  $m$  micro-instructions, avec  $m << n$ . Dans le cas (b), on construit une nanomémoire de  $m$  mots de  $w$  bits qui contiennent les  $m$  micro-instructions qu'on a isolées; on appelle alors ces instructions des *nano-instructions*. Chaque micro-instruction du programme original est remplacée en mémoire de commande par l'adresse du mot nanomémoire qui contient la nano-instruction correspondante. Comme il y a  $m$  mots en nanomémoire, la largeur du mot de la mémoire de commande doit être de  $\log_2 m$  (arrondi à l'entier supérieur), comme indiqué à la figure 4-21(b).

Dans ce contexte, le microprogramme est exécuté comme suit. Tout d'abord, on charge le premier mot du microprogramme dans un registre tampon. Ce mot correspond à l'adresse de la nano-instruction à charger dans le registre RMI. Cette nano-instruction placée dans le registre RMI permet de générer, pendant le cycle courant, les microcommandes nécessaires à la micromachine. À la fin du cycle, le mot suivant du microprogramme est chargé dans le registre tampon, puis le processus est ensuite répété de façon continue.

```

0: rad:= co; rd;
1: rd;
co:= co+1;
2: ri:= rdo;
rit:= decalg(ri);
if n then goto 28;
3: rit:= decalg(rit);
if n then goto 19;
4: rit:= decalg(rit);
if n then goto 11;
5: val:= rit;
if n then goto 9;
6: rad:= ri; rd; {LDD}
7: rd;
ac:= rdo;
goto 0;
9: rad:=ri; rdo:=ac; wr; {STOD}
10: wr;
goto 0;
11: val:= rit;
if n then goto 15;
12: rad:= ri; rd; {ADD}
13: rd;
a:= rdo;
ac:= ac+a;
goto 0;
15: rad:= ri; rd; {SUBD}
16: rd;
ac:= ac+1;
17: a:= rdo;
a:= inv(a);
18: ac:= ac+a;
goto 0;
19: rit:= decalg(rit);
if n then goto 25;
20: val:= rit;
if n then goto 23;
21: val:= ac; {POS}
if n then goto 0;
22: co:= ri;
co:= bet(co, amasq);
goto 0;
23: val:= ac; {ZER}
if z then goto 22;
24: goto 0;
25: val:= rit;
if n then goto 27;
26: co:= ri; {JUMP}
co:= bet(co, amasq);
goto 0;
27: ac:= ri;
{LLOC}
ac:= bet(ac, amasq);
goto 0;
28: rit:= decalg(rit);
if n then goto 40;
29: rit:= decalg(rit);
if n then goto 35;
30: val:= rit;
if n then goto 33;
31: a:= ri; {LDL}
a:= a+pp;
rad:= a; rd;
32: rad:= a; rdo;
ac:= rdo;
goto 0;
33: a:= ri; {STOL}
a:= a+pp;
34: rad:= a; rdo:= ac; wr;
wr;
goto 0;
35: val:= rit;
if n then goto 38;
36: a:= ri; {ADDL}
a:= a+pp;
37: rad:= a; rd;
rd;
a:= rdo;
ac:= ac+a;
goto 0;
38: a:= ri; {SUBL}
a:= a+pp;
39: rad:= a; rd;
rd;
a:= rdo;
ac:= ac+a;
goto 0;
40: rit:= decalg(rit);
if n then goto 46;
41: val:= rit;
if n then goto 44;
42: val:= ac; {INEG}
if n then goto 22;
goto 0;
43: goto 0;
44: val:= ac; {INZE}
if z then goto 0;
co:= ri;
co:= bet(co, amasq);
goto 0;
45: rit:= decalg(rit);
if n then goto 73;
46: rit:= decalg(rit);
if n then goto 50;
47: pp:= pp+(-1); {CALL}
rad:= pp; rdo:= co; wr;
wr;
48: rad:= pp; rd; {RETN}
66: val:= rit;
if n then goto 70;
67: rad:= pp; rd; {RETN}
68: rd;
pp:= pp+1;
co:= rdo;
goto 0;
49: co:= ri;
co:= bet(co, amasq);
goto 0;
50: rit:= decalg(rit);
if n then goto 65;
51: rit:= decalg(rit);
if n then goto 59;
52: val:= rit;
if n then goto 56;
53: rad:= ac; rd; {PSHI}
rd;
pp:= pp+(-1);
54: pp:= pp+1;
a:= rdo;
rad:= pp; rdo:= a; wr;
wr;
goto 0;
55: a:= rdo;
56: rad:= pp; rd; {POPI}
rd;
pp:= pp+1;
58: a:= rdo;
rad:= ac; rdo:= a; wr;
wr;
goto 0;
59: val:= rit;
if n then goto 62;
60: pp:= pp+(-1); {PUSH}
61: rad:= pp; rdo:= ac; wr;
wr;
62: rad:= pp; rd; {POP}
63: rd;
pp:= pp+1;
ac:= rdo;
goto 0;
64: ac:= rdo;
65: rit:= decalg(rit);
if n then goto 73;
66: val:= rit;
if n then goto 70;
67: rad:= pp; rd; {RETN}
68: rd;
pp:= pp+1;
co:= rdo;
goto 0;
70: a:= ac;
71: ac:= pp;
72: pp:= a;
goto 0;
73: val:= rit;
if n then goto 76;
74: a:= ri; {INSP}
a:= bet(a, smasq);
75: pp:= pp+a;
goto 0;
76: a:= ri;
a:= bet(a, smasq);
77: a:= inv(a);
78: a:= a+1;
pp:= pp+a;
goto 0;
79: a:= rdo;
rad:= ac; rdo:= a; wr;
wr;
goto 0;
80: a:= rdo;
rad:= ac; rdo:= a; wr;
wr;
goto 0;

```

Figure 4-20. Microprogramme destiné aux Mic-2 (début)

Figure 4-20. Microprogramme destiné aux Mic-2 (fin)

Imaginons, par exemple, un microprogramme de  $4096 \times 100$  bits avec seulement 128 micro-instructions différentes. Une nanomémoire de  $128 \times 100$  bits est suffisante pour contenir toutes ces micro-instructions (qui seront alors des nano-instructions). La mémoire de commande dans ce cas se réduit à  $4096 \times 7$  bits ( $128 = 2^7$ , et  $\log_2 128 = 7$ ); chacun de ses 4096 mots est un pointeur vers une nano-instruction. Dans cet exemple, l'économie de mémoire est de :

$$(4096 \times 100) - (4096 \times 7) - (128 \times 100) = 368128 \text{ bits}$$

L'économie de mémoire ainsi réalisée entraîne une diminution sensible de la vitesse de traitement de la micromachine. Une machine comprenant une mémoire de commande à deux niveaux, telle qu'on l'a définie, est plus lente en vitesse d'exécution qu'une même machine purement microprogrammée. Cela est dû au fait que le cycle de recherche des micro-instructions nécessite deux accès mémoire : un vers la mémoire de commande et un vers la nanomémoire. Ces deux actions ne peuvent pas être simultanées.

alors le micromicroprogramme ne contiendra que quelques micro-instructions distinctes, chaque micro-instruction étant alors d'une fréquence d'utilisation importante. Une variante de l'idée de base de la nanoprogrammation repose sur ce principe: les mots dans la nanomémoire peuvent être paramétrés. Soit deux micro-instructions qui ne diffèrent que d'un champ, celui qui précise le nom du registre à vider sur un bus par exemple. En pliquant le nom du registre en mémoire de commande plutôt qu'en nanomémoire, les deux micro-instructions pointent alors vers la même nano-instruction. Pendant que la nano-instruction est chargée dans RMI, le nom du registre est chargé lui aussi dans RMI mais depuis la mémoire de commande. Ainsi le RMI est chargé partiellement par la mémoire de commande et partiellement par la nanomémoire; cela nécessite bien évidemment au sein de la micromachine des circuits supplémentaires (pas forcément complexes cependant). Cette variante de la nanoprogrammation peut permettre un gain sensible en capacité mémoire de la micromachine. En effet, même si la largeur de la mémoire de commande augmente, le nombre de nano-instructions peut être réduit de façon importante.

#### 4.5.3. Amélioration des performances

La nanoprogrammation a pour effet de réduire la capacité mémoire totale de la micromachine, au prix d'une diminution des performances. D'autres techniques visent à améliorer la vitesse de traitement au détriment de la capacité mémoire de commande. Ces deux approches sont incompatibles. Elles correspondent chacune à des objectifs de développement opposés, conduisant à des machines performantes ou à des machines de faible coût. Nous examinons ci-dessous quelques méthodes d'amélioration des performances de la micromachine.

Jusqu'à présent, nous avons considéré les quatre sous-cycles machine d'égale durée (figure 4-5). Cette façon de procéder est simple, mais les performances obtenues ne sont pas optimales. Il est évident qu'en analysant les sous-cycles en détail, on s'aperçoit qu'ils n'ont aucune raison d'être de même durée et qu'il en existe au moins un qui est plus long que les autres. Ajuster les sous-cycles machine sur le plus lent entraîne forcément une limitation de la vitesse de traitement de la micromachine. Une façon d'effacer cette limitation consiste à attribuer à chaque sous-cycle des durées indépendantes, en relation exclusive avec le temps nécessaire à réaliser les actions dans chaque cycle. Bien qu'il s'agisse d'une solution possible, ce n'est pas la meilleure car le temps associé à chaque sous-cycle doit être calculé pour qu'il corresponde à celui du pire des cas possibles. Considérons, par exemple, le sous-cycle 3 de notre micromachine. Si l'UAL effectue une opération d'addition, le temps nécessaire à cette opération est plus long que celui de la copie de l'entrée A sur la sortie (en raison surtout de la propagation de retenue). Ainsi la durée affectée au sous-cycle 3, lors de la conception de la micromachine, serait celui de l'addition.

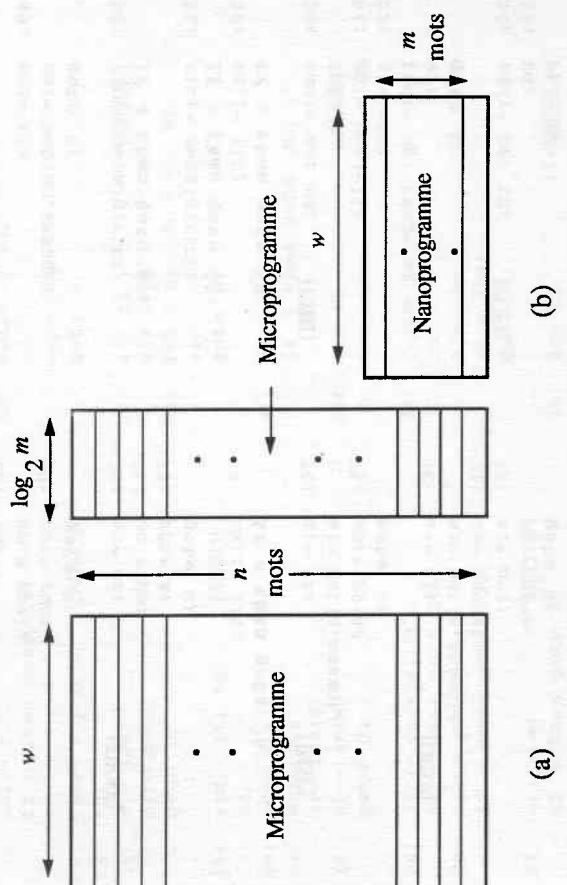


Figure 4-21. (a) Microprogramme conventionnel (b) Microprogramme et nanoprogramme

La nanoprogrammation est efficace quand une même micro-instruction est très fréquemment utilisée. Si deux micro-instructions à peu près semblables peuvent être comptabilisées comme étant réellement identiques,

Une autre façon de faire consiste à ajuster dynamiquement la durée du cycle selon l'opération réalisée, afin qu'il soit toujours le plus court possible.

L'idée ci-dessus est séduisante mais sa mise en œuvre est complexe. En effet, il est assez difficile de diminuer ou d'augmenter en cours de fonctionnement la fréquence de l'horloge du système. On préfère utiliser une horloge de fréquence fixe mais plus élevée. Sa période est calculée de façon à être plus courte que le cycle machine le plus court. La durée de chaque sous-cycle est alors égale à un nombre entier de périodes d'horloge. Dans notre exemple, si le temps de traitement de l'UAL varie de 75 à 150 ns selon l'opération à réaliser et si l'on utilise une horloge de période 25 ns, le sous-cycle associé durera entre trois et six périodes.

AC	000100	000100	077777	100001	000010
Mémoire	000050	170000	17775	000010	100001
000030	010100	100002	077771	100007	
N = 0	N = 0	N = 1	N = 0	N = 1	
V = 0	V = 0	V = 1	V = 1	V = 1	

Figure 4-22. Exemples de soustractions de deux mots de 16 bits en complément à deux. Les nombres sont en notation octale. Les bits N et V sont relatifs aux résultats des opérations.

Voyons maintenant comment indiquer à la micromachine la durée d'un sous-cycle. Il existe au moins deux façons de faire. La première consiste à distinguer dans la micro-instruction un ou plusieurs champs supplémentaires qui précisent le nombre de cycles attribués à une fonction donnée. La seconde vise à adapter le séquencement de la micro-instruction selon son code d'opération (au moyen par exemple d'un circuit PLA). On peut procéder de même pour générer certains signaux de commande en microprogrammation verticale. La première approche augmente la capacité mémoire, la seconde le temps d'exécution.

Une autre technique d'amélioration des performances de la micromachine est basée sur la souplesse des branchements conditionnels. Considérons une macro-instruction, par exemple SKIP INF, qui compare le contenu de l'accumulateur AC avec un mot en mémoire centrale et qui «saut par-dessus» la micro-instruction qui suit, dans le cas où AC est plus petit. Pour effectuer l'opération de comparaison, on réalise une soustraction entre le mot mémoire et l'accumulateur; un débordement peut apparaître comme le montre la figure 4-22. Le fait d'un possible débordement (indicateur V), ne permet pas de déterminer le plus petit nombre simplement en examinant le bit de signe du résultat (indicateur N). Par exemple, à la quatrième opération de la figure 4-22, bien que AC soit inférieur au mot mémoire, le résultat de l'opération est positif (N = 0) mais il y a

débordement (V = 1). La condition correcte à tester dans ce cas doit être «N OU-Exclusif V». L'indicateur de débordement V est mis à un, chaque fois que la retenue du bit de signe diffère du bit de signe du résultat.

En général, les UAL disposent de plusieurs bits indicateurs précisant l'état final d'un traitement, par exemple N, Z, V et C (C pour la retenue d'addition). Toutefois, si l'on ne dispose que de quatre micro-instructions de branchement conditionnel, une pour chaque bit N, Z, V et C, la macro-instruction SKIP INF nécessite alors l'exécution de plusieurs micro-instructions.

	Champ ADDR	NZVC	Adresse de branchement
		1000 0000	1001 1001
		1000 1000	1000 1001
		1000 1011	1000 1011
		1000 1011	1000 1011
		1000 1011	1000 1011
		1000 1011	1000 1011
		1000 1111	0000 1000
		1000 1111	1000 1111
		1000 1111	1000 1111
		1000 0000	1100 1000
		1000 0000	1000 1100

Figure 4-23. Exemples de branchements conditionnels multiples basés sur les bits NZVC

Pour offrir plus de flexibilité aux branchements conditionnels, sur la plupart des machines, les bits indicateurs en sortie de l'UAL ne sont pas testés individuellement mais de façon combinée. C'est ainsi qu'un bit particulier de la micro-instruction provoque la réalisation d'un OU logique entre les indicateurs NZVC et les quatre bits de poids faible du champ ADDR, on obtient alors l'adresse de branchement concernée. Quelques exemples sont présentés à la figure 4-23. Si les quatre bits de poids faible du champ ADDR sont à zéro, les bits NZVC définissent alors un branchement multiple à seize directions possibles. En revanche, si les bits de ADDR sont à un, on est en présence d'un branchement inconditionnel à une adresse se terminant par 1111. Toutes ces facilités rendent la macro-instruction SKIP INF plus simple à interpréter. Considérons une valeur du champ ADDR se terminant par 0101, par exemple 1000 0101, et analysons le branchement. Les micro-instructions d'adresses 1000 1101 et 1000 0111 sont relatives au succès de SKIP INF et celles d'adresses 1000 0101 et 1000 1111 à son échec. Aucun décodage supplémentaire

n'est exigé. On aurait très bien pu prendre comme base d'adresse 0000 à la place de 0101, mais cela aurait été trop coûteux car ces adresses sont les seules disponibles et référencées par le branchement multiple à seize directions. Ces adresses ne doivent pas être choisies à la légère.

Il apparaît qu'avec ce mode de séquencement des micro-instructions, leur placement en mémoire microprogramme devient un vrai casse-tête. La première micro-instruction exécutée après un branchement multiple doit contenir obligatoirement un branchement inconditionnel. En effet, l'instruction qui la suit est toujours utilisée comme destination possible d'un branchement. Mais alors, quelles doivent être les valeurs des adresses de branchement? Certainement pas les adresses de la forme xxxx0000 car elles sont trop prisées. Toutes les autres adresses (à l'exception de celles de la forme xxxx1111) peuvent bien évidemment convenir. Les choix doivent être faits avec beaucoup d'attention pour éviter des branchements vers des micro-instructions inopportunes. C'est le cas, par exemple, des adresses paires qui ne concernent daucune manière des micro-instructions relatives au test du bit C.

#### 4.5.4. Traitement pipeline

Une autre voie d'amélioration de la vitesse de traitement consiste à organiser la machine en plusieurs sous-unités fonctionnelles indépendantes et, dès lors, à mettre en œuvre des techniques de traitement de type pipeline. Nous avons montré à la figure 2-5 un exemple de machine à traitement pipeline basée sur cinq unités indépendantes. A la figure 4-24 nous montrons le diagramme fonctionnel de cette machine. C'est ainsi que l'instruction 1 est extraite de la mémoire lors du cycle 1 pour être réalisée ou exécutée au cycle 5. De même pour l'instruction 4 : elle est extraite de la mémoire au cycle 8, décodée et analysée au cycle 9, etc. Elle est exécutée au cycle 12. Aux conditions d'initialisation près, on constate qu'une instruction peut être exécutée à chaque cycle. Ainsi, le taux moyen d'exécution est d'une instruction par cycle avec la technique pipeline contre une instruction pour cinq cycles avec une technique séquentielle classique.

Hélas, il apparaît que, dans les programmes, on rencontre environ 30% d'instructions de rupture de séquence, ce qui remet en cause les mécanismes d'anticipation de la machine pipeline. On distingue deux types de branchement ou rupture de séquence, les branchements conditionnels ou inconditionnels et les boucles. Lors d'un branchement inconditionnel l'UC cesse d'extraire les instructions les unes à la suite des autres en mémoire pour se diriger vers celles se trouvant à une adresse imposée, communiquée par l'instruction de branchement. Un branchement conditionnel a un comportement semblable, sous réserve que la condition du branchement soit remplie. Par exemple, une instruction examinant le contenu d'un registre et qui réalise un branchement si le contenu du registre est nul. En revanche, si le contenu du registre est différent de zéro, le branchement n'est pas effectué et l'extraction des instructions continue en séquence.

Une instruction typique de boucle, décrémente un compteur d'itération et entreprend un branchement arrière, au début de la boucle, si le compteur est dans un état différent de zéro (c'est-à-dire qu'il y a d'autres itérations à effectuer). L'instruction de boucle est un cas particulier du branchement conditionnel pour lequel on sait toujours à l'avance si le branchement aura lieu ou non.

	1	2	3	4	5	6	7	8	9	10	11	12	13
Recherche des instructions	1	2	B						4	5	6	7	8
Décodage des instructions	1	2	B						4	5	6	7	8
Calcul des adresses	1	2	B						4	5	6	7	8
Recherche des opérandes				1	2	B			4	5	6		
Exécution					1	2	B			4	5		

Figure 4-24. Fonctionnement d'un pipeline à cinq étages. Les numéros sont relatifs aux instructions. L'instruction B correspond à un branchement conditionnel.

Exammons ce qu'il advient du pipeline de la figure 4-24 lorsqu'une instruction de branchement conditionnel apparaît (la lettre B sur la figure). L'instruction qui sera exécutée après B peut être celle qui suit B ou celle référencée par l'adresse de branchement contenue dans B. Comme l'unité d'extraction des instructions ne connaît pas la prochaine instruction qui sera traitée après B, elle arrête d'extraire les instructions de la mémoire jusqu'à ce que B soit exécutée. Ainsi le pipeline se vide. C'est après le cycle 7 que l'on connaît la prochaine instruction exécutée, suite à B. Le branchement a provoqué la perte de quatre cycles d'anticipation, cela correspond au concept de la *pénalisation du branchement* associé au traitement pipeline. Si l'on se trouve en présence d'une instruction de branchement en moyenne toutes les trois instructions, il est certain que les performances du pipeline seront moindres.

De nombreux travaux de recherches ont été menés pour tenter d'améliorer le fonctionnement des pipelines (DeRosa et Levy, 1987; McFarling et Hennessy, 1986; Hwu et al., 1989; Lilja, 1988). L'une des façons les plus simples de procéder, consiste à se comporter en supposant que le branchement n'aura pas lieu et de continuer ainsi à remplir le pipeline d'instructions; comme si le branchement n'était qu'une simple opération arithmétique. S'il apparaît que le branchement n'a pas lieu, il n'y a pas de temps de perdu. En revanche, si le branchement a lieu il est nécessaire d'effectuer une rupture de séquence, de vider le pipeline et de repartir sur une autre séquence d'instructions. On nomme quelquefois ce mode de gestion du pipeline, la *technique de l'érasrement ou du refoulement*, cette

technique présente quelques difficultés. Par exemple, sur certaines machines lors de la phase de calcul d'adresse il se peut qu'un ou plusieurs registres soient modifiés. Une instruction qui se fait effacer est effacée, même si certains registres avaient pu être modifiés lors de son prétraitement. Il est nécessaire, avant de repartir, de restaurer l'état initial des registres précédant l'écrasement. Il faut donc mettre en œuvre un mécanisme permettant de restituer la valeur originale des registres.

Voyons comment construire un modèle simple permettant d'évaluer la diminution des performances du pipeline. Considérons :

$P_j$  la probabilité qu'une instruction soit un branchement,  
 $P_t$  la probabilité qu'un branchement soit effectué,  
 $b$  la pénalité du branchement.

Le temps moyen d'exécution des instructions (en nombre de cycles) peut être obtenu en calculant une moyenne pondérée des deux cas possibles : des instructions classiques et des instructions de branchement.

$$\text{Temps moyen d'exécution} = (1 - P_j)(1 + P_t(1 + b) + (1 - P_t)(1))$$

Certains calculs laissent apparaître que le temps moyen d'exécution des instructions est égal à  $(1 + bP_jP_t)$ . L'efficacité d'exécution est alors égale à  $[1 / (1 + bP_jP_t)]$ . En prenant par exemple, comme valeur typique mesurée,  $b = 4$ ,  $P_j = 0,3$  et  $P_t = 0,65$  la machine fonctionne à moins de 60% de ses potentialités. Comment faire alors, pour améliorer les performances de cette machine ? Si l'on pouvait prévoir, au départ, qu'un branchement aurait lieu, on irait chercher en mémoire les instructions associées. On éliminerait ainsi la pénalité du branchement. Dans la formule de l'efficacité ci-dessus, on remplace  $P_t$  par  $P_w$ , la probabilité de se tromper.

Deux types de prévision sont possibles : l'une statique (le temps de compilation) et l'autre dynamique (le temps d'exécution). En prévision statique le compilateur fait une estimation de réussite sur chaque instruction de branchement qu'il génère. Par exemple, dans le cas d'une instruction de boucle, l'estimation du branchement arrière est maximale la plupart du temps. En revanche, lorsqu'il est question de tester une condition peu probable, comme un appel système qui retournerait un code d'erreur, alors on peut dire que le branchement n'aura pas lieu avec une forte probabilité. Le plus souvent, des instructions différentes sont utilisées pour ces cas spécifiques et un simple examen du code opération de l'instruction suffit à fournir une indication substantielle de la probabilité de branchement.

Une solution plus élaborée consiste à distinguer deux codes opérations pour chaque type de branchement. De ce fait, lorsque le compilateur estime que le branchement aura lieu il utilise le premier code opération, il génère le second en cas contraire. Pour les programmes très fréquemment sollicités, on a une alternative qui consiste à faire exécuter le programme sur un simulateur afin d'enregistrer son comportement avec l'un et l'autre des

codes opération de l'instruction de branchement. Le programme exécutable est ensuite modifié en remplaçant chaque instruction de branchement par le code opération le mieux approprié selon la probabilité de réussite ou d'échec constatée.

L'autre méthode de prévision est dynamique. Pendant l'exécution, le microprogramme élabore une table des adresses relatives à chaque branchement et conserve la trace du comportement de chacun d'eux. Cette façon de faire a tendance à entraîner une baisse sensible de la vitesse de traitement qui doit être compensée par l'usage de circuits intégrés supplémentaires. Des mesures ont mis en évidence qu'il était possible d'atteindre, sans difficulté, 90% d'efficacité avec cette méthode.

L'estimation de la probabilité de branchement n'est pas la seule parade à la pénalisation du branchement conditionnel. Il est en effet possible de déterminer assez tôt quelle sera la suite des instructions en cas d'égalité». Des tests de type «branchement en cas de supériorité» sont plus simples à estimer que d'autres, de type «branchement en cas de supériorité». Le Premier peut être effectué avec un comparateur tandis que le second nécessite un cycle complet du chemin des données pour effectuer une soustraction. Ainsi, à chaque fois qu'un branchement se présente, le microprogramme effectue un contrôle rapide sur les premiers étages du pipeline afin de savoir s'il lui est possible de calculer le branchement de façon anticipée. Dans l'affirmative, le microprogramme localise les prochaines instructions à exécuter et peut alors entreprendre d'aller les chercher en mémoire. Les compilateurs peuvent également fournir une aide précieuse à ce sujet. Par exemple, lorsqu'un programmeur écrit une boucle de 1 à 10 avec un pas de  $i$ , le compilateur élabore le test de  $i < 11$  en préférant voir si  $i = 10$  plutôt que  $i < 11$ , de telle façon que le microprogramme effectue plus facilement une comparaison qu'une soustraction.

```
if b < c
  a := b + c;
else Instruktion;
```

(a)

(b)

Figure 4-25. (a) Extrait d'un programme Pascal (b) Comment un compilateur peut traiter le programme

Pour pallier les pénalisations des branchements qui ne peuvent être résolues par anticipation, un compilateur essaiera de fournir à l'unité centrale des instructions utiles à traiter en attendant que le branchement soit exécuté. Considérons par exemple, à la figure 4-25(a), une instruction arithmétique suivie d'un test. Un compilateur quelque peu ingénieux peut fournir, par exemple, un code exécutable semblable à celui de la figure 4-25(b) qui n'est pas très légal en Pascal, mais qui montre bien l'ordre des choses. En premier il fournit le code pour effectuer le test puis celui de

l'opération arithmétique. Une fois le branchement introduit dans le pipeline, quelques «instructions normales» (c'est-à-dire sans branchement) y sont introduites à la suite. Cela est sans importance, car il n'y a pas lieu d'effectuer ni estimation d'exécution, ni érasrement de ces quelques instructions. Pour utiliser au mieux cette technique le microprogrammeur et le concepteur du compilateur doivent collaborer étroitement pendant la phase de conception.

Dans le pire des cas, il y a toujours possibilité de commencer les deux voies du branchement en parallèle. Cette approche nécessite de disposer de deux pipelines et n'élimine pas pour autant le problème de l'érasrement. Sur certains ordinateurs de haut de gamme où les performances sont recherchées à tout prix, on met quelquefois en œuvre cette technique du double pipeline. Si, dans l'une ou l'autre des voies du branchement courant, un nouveau branchement survient avant qu'il ne soit lui-même résolu, c'est alors que les complications sérieuses apparaissent. Disposer d'une demi-douzaine de pipelines en réserve pour les distribuer au fil des demandes n'est sûrement pas une des meilleures idées.

Après ces quelques réflexions générales sur le traitement pipeline, revenons à notre micromachine qui ne dispose pas d'unité spécifique d'extraction, ni de décodage ou autre, des instructions. Il n'est donc pas possible d'effectuer sur cette machine des traitements pipeline. Toutefois, quelques modifications mineures du microprogramme permettraient d'envisager un certain recouvrement des cycles d'extraction et d'exécution des micro-instructions. Il s'agirait d'une certaine forme de traitement pipeline.

Ainsi, à la figure 4-10 par exemple, si la micro-instruction suivante à exécuter pouvait, d'une certaine façon, être extraite pendant le sous-cycle 4, l'intérêt et l'utilité du sous-cycle 1 disparaîtraient. L'horloge, dans ce cas, ne généreraient que les impulsions relatives aux sous-cycles 2, 3, 4, 2, 3, 4, etc. Les problèmes apparaissent avec les micro-instructions de branchement conditionnel, comme nous l'avons vu précédemment. Si la machine doit attendre que les indicateurs d'états en sortie de l'UAL soient stables avant d'aller querrir la micro-instruction suivante, il est trop tard : le cycle est pratiquement terminé et le recouvrement dans ce cas est quasi nul. Sur certaines machines, on a résolu ce problème en se référant aux indicateurs d'états relatifs au cycle précédent. Bien sûr, au préalable, il a fallu les mémoriser dans un registre pour éviter de les voir disparaître. Ces informations sont disponibles au début de chaque micro-instruction, le cycle de recherche de la micro-instruction suivante peut être entrepris dès que celui en cours est terminé, bien avant que la micro-instruction courante soit complètement exécutée. Nul besoin de préciser, dans ce cas, que la tâche du microprogrammeur est nettement plus compliquée.

Il est techniquement possible d'écrire des microprogrammes pour de telles machines. Par exemple, supposons que l'on souhaite tester le contenu d'un registre tampon et effectuer un branchement s'il est négatif. Le microprogrammeur doit écrire une première micro-instruction permettant à

l'information de transiter au travers de l'UAL (celle-ci ne contenant pas de branchement), puis une seconde micro-instruction de branchement (qui ne contient pas de traitement). Cela revient à tester les bits indicateurs mémorisés à la première micro-instruction et à effectuer le branchement si nécessaire. Le prix à payer, dans ce cas, pour programmer de façon propre le branchement, revient à doubler le temps nécessaire au branchement, ce qui est tout à fait indésirable.

La seule façon de faire consiste à analyser au mieux la situation. Par exemple, le microprogrammeur peut estimer quel chemin du branchement est le plus fréquemment emprunté et commencer, dès la micro-instruction qui suit, les tâches correspondant au chemin le plus probable. Si, par hasard, le branchement conditionnel mène vers l'autre chemin, il est nécessaire de pratiquer un érasrement.

```

11: val:= rit;
12: rad:= ri; rd; if n then goto 16; 52: val:= rit;
                                              rad:= ac; rd; if n then goto 56;
13: rd;                                     [ADDD] 53: rad:= ac; rd; {PSHI}
14: ac:= rdo+ac; goto 0;                   54: pp:= pp+(-1); rd; {POPI}
                                              rad:= pp; wr; goto 10;
16: ac:= ac+1; rd;                      (SUBD) 55: rd;
17: a:= inv(rdo);                      56: rad; pp:= pp+1; rd;
18: ac:= ac+a; goto 0;                  58: rd;
                                              rad:= ac; wr; goto 10;

```

(a)

(b)

**Figure 4-26.** Exemple de micro-instructions sur une micromachine mettant en œuvre l'imbrication des cycles recherche et exécution

Comme exemple d'application, récrivons les lignes 11 à 18 de la figure 4-16 de notre micromachine en faisant se chevaucher les cycles d'extraction et d'exécution des micro-instructions. C'est ce que montre la figure 4-26(a). Dans cet exemple, on est quelque peu chanceux car, pendant l'exécution de la ligne 11, on sait que le code opération de l'instruction suivante est ADDD ou SUBD; en effet toutes deux commencent par la même micro-instruction. Ainsi, il est facile de placer la plus fréquente à la ligne 12 pour maintenir la machine en action pendant la réalisation du branchement.

Un exemple moins favorable est présenté à la figure 4-26(b). A la ligne 52 on sait si PSHI ou POPI sera l'instruction suivante. Contrairement à l'exemple précédent, ces deux instructions ne commencent pas par la même micro-instruction. Supposons qu'une estimation statistique des programmes Mac-1 montre que PSHI est plus fréquent que POPI. On peut alors procéder comme à la figure 4-26(b). Pour l'instruction PUSHI tout va bien, mais pour POPI, à la ligne 56 une lecture mémoire est générée vers une adresse incorrecte. Il se peut que, selon les caractéristiques d'organisation de la micromachine, il soit possible d'interrompre le cycle de lecture

mémoire entrepris sans perturber le fonctionnement de la micromachine. Dans le cas contraire, il suffit de générer un nouveau cycle de lecture mémoire vers, cette fois, l'adresse correcte. Dans l'exemple, POPI nécessite 15 micro-instructions contre 13 sur le programme original de la figure 4-16. Toutefois, si le recouvrement des cycles recherche et exécution des micro-instructions permet un gain de 15% sur leur temps d'exécution on peut dire que POPI est malgré tout plus rapide dans l'exemple actuel.

Si les branchements multiples que nous avons définis précédemment (un OU entre les bits NZVC et ceux du poids faible du champ ADDR) étaient mis en œuvre sur notre micromachine, nul doute qu'elle serait bien plus complexe. Sur la plupart des machines, le recouvrement des cycles recherche et exécution des micro-instructions est largement mis en pratique. Au début du cycle exécution de la micro-instruction  $n$ , le choix de la micro-instruction  $n+1$  peut être fait en consultant les bits indicateurs positionnés par l'UAL et mémorisés lors de la micro-instruction  $n-1$ . Nous ne poursuivrons pas plus avant dans cette direction, mais c'est à présent que l'on prend conscience de la réalité de la définition de Rosin sur la microprogrammation.

#### 4.5.5. Mémoire cache

De tout temps, les unités centrales ont été plus rapides que les mémoires principales. Bien que les performances des mémoires s'améliorent, celles des unités centrales font de même et l'écart de rapidité reste constant. Ainsi, lorsque l'unité centrale sollicite la mémoire, elle passe une bonne partie de son temps à attendre que la mémoire réagisse. Il est fréquent pour l'unité centrale de générer une lecture mémoire lors d'un cycle de lecture et de n'obtenir la donnée sollicitée que deux ou trois périodes d'horloge plus tard, même s'il n'y a pas de temps mort.

En fait, le problème n'est pas d'ordre technologique, mais économique. En effet, aujourd'hui, il est possible de construire des mémoires aussi rapides que les unités centrales mais leur coût, pour des capacités de plusieurs mégaoctets, serait prohibitif. Ainsi, les choix qui sont faits sur la grande majorité des ordinateurs (à l'exception toutefois des «super-ordinateurs» où les performances priment sur le coût) consistent à disposer d'une faible quantité de mémoire rapide associée à une quantité importante de mémoire relativement plus lente. Il va de soi que l'on préférera une grande capacité de mémoire, rapide et peu coûteuse.

D'intéressantes techniques permettent de combiner deux types de mémoires afin d'obtenir la vitesse de la plus rapide et la capacité de la plus lente, à un prix tout à fait raisonnable. La mémoire la plus rapide et de faible capacité est appelée *mémoire cache* ou plus simplement *cache* (ce mot vient du verbe *cacher*). On utilise également le mot *antémémoire*. Pour des raisons d'efficacité et de performances, le cache est le plus souvent sous contrôle du microprogramme. Nous présentons ci-dessous la technique et l'utilisation des caches dans les ordinateurs. Des informations supplémentaires peuvent

être obtenues sur les caches par la lecture des articles de : Agarwal et al., 1989; Farrens et Pleszakun, 1989; Kabakibo et al., 1987; Kessler et al., 1989; Pohm et Agarwal, 1983; Przybylski et al., 1989; Smith, 1982 et enfin Wang et al., 1989.

Depuis des années, on sait que les programmes n'accèdent pas à leur mémoire de façon complètement aléatoire. Si une référence mémoire concerne l'adresse  $A$ , il est vraisemblable que la référence mémoire suivante sera dans le voisinage de  $A$ . L'exemple le plus simple est le programme lui-même. A l'exception des branchements et des appels de procédures, les instructions du programme sont situées en mémoire à des adresses consécutives. De plus, la plupart du temps, les programmes réalisent des boucles dans lesquelles un nombre limité d'instructions sont réexécutées à chaque itération de boucle. De même, un programme travaillant sur une matrice de données effectue en toute vraisemblance bon nombre de références vers cette matrice avant de se diriger vers d'autres données, ailleurs en mémoire.

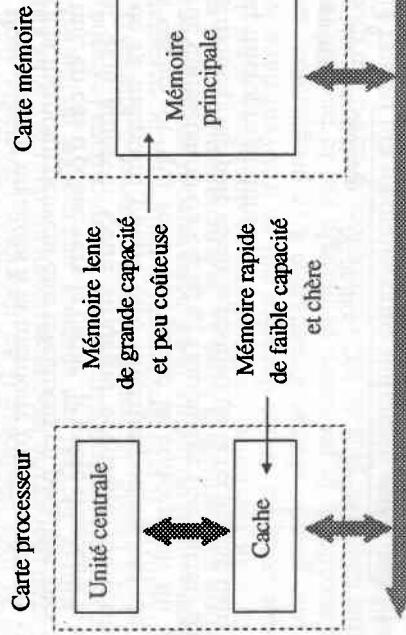


Figure 4-27. Le cache est souvent placé sur la carte processeur

Observer que les références à la mémoire, pendant un court intervalle de temps, ne portent que sur une fraction localisée de l'espace mémoire total, c'est découvrir le *principe de la localité* qui forme la base des systèmes à mémoire cache. L'idée principale consiste à faire en sorte qu'un mot référencé par l'unité centrale commence par être copié de la mémoire principale (la plus lente) dans le cache, ainsi la prochaine fois qu'il sera utilisé, on y accédera plus rapidement. Un exemple d'organisation d'une unité centrale, d'un cache et d'une mémoire principale est illustré à la figure 4-27. Si un même mot est lu ou écrit  $k$  fois pendant un court laps de temps, l'unité centrale génère une référence vers la mémoire principale et

$k - 1$  références vers le cache. Plus  $k$  est important, meilleures sont les performances du système.

On peut formaliser ces concepts par calculs, en définissant :  $c$  le temps d'accès du cache,  $m$  le temps d'accès de la mémoire principale,  $h$  le taux de présence ou de succès qui exprime le rapport du nombre de références au cache vis-à-vis du nombre de références totales. Dans l'exemple présenté au paragraphe précédent,  $h = (k - 1) / k$ . Certains auteurs définissent le taux d'insuccès ou de défaut qui correspond alors à  $1 - h$ .

A partir des définitions, on peut calculer le temps moyen d'accès,  $t_{acc}$ , aux données, comme suit :

$$t_{acc} = c + (1 - h)m$$

Dès que  $h \rightarrow 1$ , les références sont dirigées de plus en plus vers le cache et  $t_{acc} \rightarrow c$ . En revanche, lorsque  $h \rightarrow 0$ , cela signifie que la mémoire principale est référencée de plus en plus souvent et que  $t_{acc} \rightarrow c + m$ . Cela signifie aussi qu'une première tentative d'accès vers le cache ( $c$ ) se traduit par un échec, d'où s'ensuit un accès à la mémoire ( $m$ ). Sur de tels systèmes, les références à la mémoire peuvent être effectuées en parallèle avec celles au cache. De ce fait, en cas d'échec vers le cache, le cycle d'accès à la mémoire principale est déjà amorcé. Cette stratégie nécessite toutefois que la sollicitation de la mémoire puisse être interrompue en cas de réponse favorable du cache, ce qui rend complexe la conception du système. L'algorithme de consultation du cache et l'amorçage (ou l'arrêt) d'un cycle d'accès à la mémoire principale selon le résultat de la recherche dans le cache est effectué par microprogramme.

Deux organisations de systèmes à mémoire cache, fondamentalement différentes, sont couramment utilisées. On rencontre quelquefois une troisième forme qui est une association des formes principales. Dans chaque organisation la mémoire principale comprend  $2^m$  octets; elle est divisée en blocs consécutifs de  $b$  octets, soit un total de  $2^m/b$  blocs de données. Chaque bloc est situé à une adresse multiple de  $b$ . La taille  $b$  d'un bloc est normalement d'une puissance entière de 2.

Le premier type de cache, appelé *cache associatif*, est illustré à la figure 4-28. Il comprend un certain nombre de lignes d'adresses ou plus simplement lignes; chaque ligne du cache contient un bloc de données et son numéro ainsi qu'un bit de validation précisant si le bloc est en cours d'utilisation ou non. L'exemple de la figure 4-28 montre un cache comprenant 1024 lignes et une mémoire principale disposant de 224 octets divisés en 222 blocs de 4 octets. Dans un cache associatif, l'ordre des enregistrements est aléatoire.

A l'initialisation du système, tous les bits de validation sont mis à zéro pour préciser que le cache est vide. Supposons que la première instruction référence le mot d'adresse 0 en mémoire principale (soit 32 bits). Le microprogramme vérifie les enregistrements dans le cache afin de voir si le bloc 0 s'y trouve. Devant son absence, une requête de bus est générée pour extraire le mot mémoire d'adresse 0 et le copier dans le cache. Le bloc 0 du cache est alors validé; il contient le mot mémoire d'adresse 0. Si ce mot est à nouveau sollicité, il sera directement lu dans le cache en évitant une requête de bus. Ainsi, au fil du temps, le cache se remplit progressivement et les bits de validation sont activés (à l'état 1). Si le programme utilise moins de 1024 références à la mémoire (instructions et données), c'est finalement le programme complet et ses données qui se trouvent dans le cache. Cela garantit une grande vitesse d'exécution à ce programme, sans qu'il y ait besoin d'accéder à la mémoire principale via le bus. En revanche, si le programme nécessite plus de 1024 références, à un certain moment le cache est plein, il faut alors libérer un enregistrement qui ne sert plus pour faire place à un nouvel enregistrement. En pratique, le choix de l'enregistrement à libérer doit se faire rapidement (des temps exprimés en nanoseconds). Sur le VAX comme sur d'autres machines on choisit une ligne du cache au hasard. D'autres algorithmes bien plus perfectionnés sont présentés au chapitre 6, dans la section consacrée à la mémoire virtuelle où des problèmes semblables apparaissent.

La particularité du cache associatif vis-à-vis d'autres caches est que chacune de ses lignes contient le numéro du bloc et l'enregistrement correspondant. Quand une adresse mémoire se présente, le microprogramme calcule (facilement) le numéro du bloc et examine s'il est présent ou non dans le cache (cela est beaucoup moins facile). Pour éviter une recherche linéaire et séquentielle, le cache associatif dispose de circuits spécialisés qui comparent rapidement et simultanément tous les numéros d'enregistrements du cache à un numéro de bloc donné en référence, de façon plus efficace

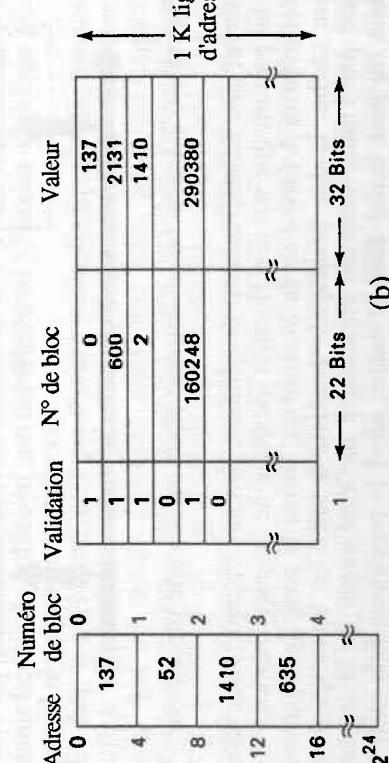


Figure 4-28. Un exemple de cache : (a) Une mémoire organisée en blocs de 4 octets (b) Un cache associatif disposant de 1024 lignes

qu'une boucle microprogrammée. Cela rend le cache associatif complexe et coûteux.

du mot dans un champ spécifique supplémentaire, appelé *indicateur ou index* (voire *tag*). La valeur de cet index ne peut être obtenue directement à partir du numéro de ligne du cache.

Voyons, sur un exemple, comment les choses se passent. Soit l'instruction MOVE 4100, 12296 située en mémoire à l'adresse 8192. Elle consiste à recopier la donnée se trouvant à l'adresse 4100 à l'adresse 12296. Le numéro du bloc correspondant à l'adresse 8192 est égal à 2048; on l'obtient en divisant l'adresse par 4 (taille du bloc). On détermine ensuite le numéro de ligne du cache où l'on rangera l'instruction; en calculant  $2048 \bmod 1024$  on obtient la même valeur qu'en examinant directement les 10 bits de poids faible de 2048. Le numéro de ligne est égal à 0. Les 12 bits de poids fort de l'adresse contiennent la valeur 2, c'est l'index cherché. La figure 4-29(a) montre l'état du cache après trois calculs d'adresse et le chargement des données associées aux blocs.

La figure 4-29(b) montre le fractionnement de l'adresse. Les deux bits les plus à gauche sont toujours égaux à zéro (étant donné que notre cache utilise un nombre entier de blocs, multiple de la taille du bloc : 4 octets dans l'exemple). Puis on trouve les 10 bits du numéro de ligne et les 12 bits de l'index. Il est ainsi facile de concevoir les circuits générant directement le numéro de ligne et l'index à partir d'une adresse mémoire.

Le fait que plusieurs blocs correspondent à une même ligne du cache cause un certain trouble. Supposons que l'instruction MOVE précédente copie la donnée d'adresse 4100 à l'adresse 12292 au lieu de 12296. Ces deux adresses correspondent à la ligne numéro 1 du cache. Selon les particularités du microprogramme, c'est l'une des deux adresses qui se trouvera dans le cache; l'autre sera ignorée voire éliminée. Cela n'est pas très grave dans notre cas, mais la correspondance multiple à une même ligne du cache risque d'entraîner dans certains cas des dégradations de performances. Ce n'est pas du tout le but recherché. Une façon de vaincre cette difficulté consiste à dilater le cache en largeur, afin d'autoriser plus d'une entrée par ligne. Le PDP11/70 comprend, notamment, deux entrées par ligne. Un cache à multiples entrées ( $n$ ) par ligne est appelé *cache associatif à  $n$  groupes ou secteurs*; il se présente comme le montre la figure 4-30.

Les caches associatifs ou direct sont des cas particuliers de caches associatifs à  $n$  groupes. En réduisant le nombre de lignes du cache à une seule, les  $n$  entrées sont placées sur cette seule et unique ligne. Dans ce cas, chaque entrée dispose d'un index propre. Il s'agit alors d'un cache associatif. En revanche, si le nombre d'entrées par ligne est  $n = 1$ , on obtient alors un cache direct qui comprend une entrée par ligne.

Qu'il soit direct ou associatif, un cache présente des avantages et des inconvénients. Le cache direct est plus simple, moins coûteux et présente un temps d'accès plus court. On obtient directement le bloc sollicité, par un mécanisme d'adressage indexé, à partir d'une fraction de l'adresse mémoire utilisée comme index. En revanche, le cache associatif présente un taux de succès plus élevé quel que soit le nombre de lignes du cache, car il n'y a

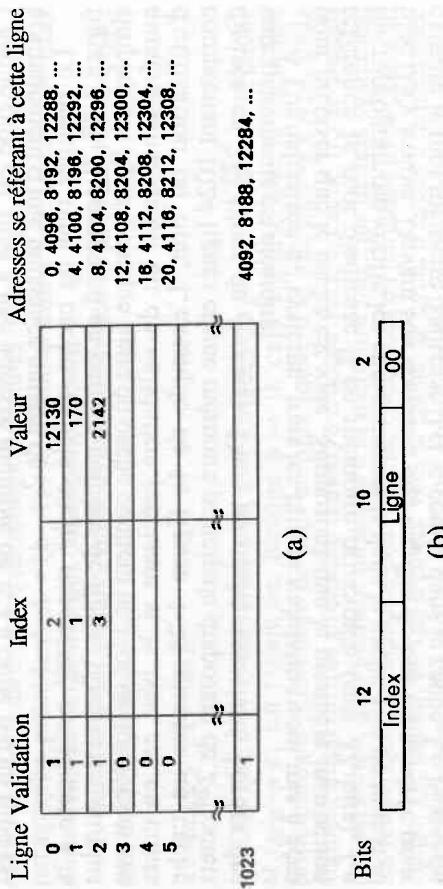


Figure 4-29. (a) Un cache direct disposant de 1024 lignes de 4 octets (b) Organisation d'une adresse de 24 bits permettant d'obtenir le numéro de ligne et l'index associés

Un autre type de cache fut élaboré afin d'en réduire le coût, le *cache à correspondance directe* ou *cache direct (mapping cache)*. Ce cache évite la phase complexe de recherche de bloc, en placent chaque bloc dans une ligne du cache dont le numéro de ligne est calculé simplement à partir du numéro du bloc. Par exemple, le numéro de ligne peut être égal au numéro du bloc *modulo* le nombre de lignes du cache. Par exemple, avec des blocs de quatre octets (soit un mot mématoire) et un cache à 1024 lignes, le numéro de ligne recevant le mot mématoire d'adresse A est égal à  $(A/4) \bmod 1024$ . Sur la figure 4-29 on constate que les mots mématoire d'adresses 0, 4096, 8192, etc., seront tous enregistrés dans le cache à la ligne 0. Ceux d'adresses 4, 4100, 8196, etc., le seront à la ligne 1 et ainsi de suite. Si le cache direct supprime le problème de la recherche des blocs, en revanche il génère un nouveau problème. Comment savoir, à un instant donné, quel est le bloc (parmi tous ceux possibles) qui se trouve dans le cache? En effet, le mécanisme de remplissage du cache a généré 1024 classes équivalentes basées sur le numéro du bloc *modulo* la taille du cache. C'est ainsi que sur la figure 4-29(a), la ligne 0 du cache peut très bien contenir les mots d'adresses 0, 4096, 8192, etc. Une façon simple de s'en sortir consiste à mettre, dans la ligne du cache, ouvre la valeur du mot, une partie de l'adresse

jamais de conflit d'accès. A aucun moment,  $k$  mots ne peuvent se trouver localisés à un même endroit dans le cache. Lors de la conception d'un cache il est nécessaire d'effectuer par simulation une estimation du cache afin de déterminer quelles seront ses performances et à quel coût. En plus de la définition du nombre de lignes, le concepteur doit également définir la taille des blocs. Dans notre exemple, c'est pour des raisons de relative simplicité que nous avons utilisé des blocs de quatre octets correspondant à un mot mématoire de 32 bits. En pratique, des blocs de 2, 4, 8 mots ou plus sont souvent utilisés.

Ligne	Validation	Index	Valeur	Entrée 0	Entrée 1	Entrée $n-1$
0	$\approx$					
1	$\approx$					
2	$\approx$					
3	$\approx$					
4	$\approx$					
5	$\approx$					

**Figure 4-30.** Un cache associatif à  $n$  groupes comprenant  $n$  entrées par ligne

L'avantage d'utiliser des blocs de taille importante est de réduire la surcharge du bus en extrayant de la mémoire des blocs de 8 mots plutôt que d'un seul mot. Cela est d'autant plus vrai, s'il est possible d'effectuer sur le bus des transferts par blocs de données. L'inconvénient des blocs volumineux est lié au fait que tous les mots copiés dans le cache ne sont pas toujours nécessaires, cela entraîne donc une surcharge importante sur le bus.

tojours nécessaires, cela entraîne donc une surcharge inutile sur le bus. Un choix important dans la conception d'un cache est relatif à la politique de gestion des opérations d'écriture. Quand l'unité centrale exécute une instruction qui est supposée modifier le contenu d'un mot mémoire, il se pose un problème de cohérence. En effet, il peut exister deux copies de ce mot dans le système, une dans le cache, une autre en mémoire principale. Pour pouvoir exploiter toutes les possibilités d'un cache, si une copie est présente, elle doit être mise à jour. Deux stratégies de mise à jour prédominent. La première, dite à *écriture immédiate*, consiste à écrire une donnée simultanément dans le cache et en mémoire. Cette approche garantit une conformité totale entre les informations du cache et de la mémoire.

L'autre stratégie, dite à *écriture différée*, consiste à effectuer l'écriture en mémoire uniquement au moment du remplacement d'un bloc modifié par un nouveau bloc. Cette technique ne maintient pas à jour la mémoire principale à chaque écriture dans le cache. De plus, il est nécessaire de disposer d'un bit indicateur précisant si le bloc a été modifié depuis son chargement dans le cache.

Dans tous les aspects de conception d'un cache, des compromis doivent être faits entre le coût, les performances, la complexité, etc. La technique d'écriture directe simultanée entraîne un trafic plus important sur le bus que celle à écriture différée. De plus, si l'unité centrale lance une opération d'E/S entre la mémoire et le disque, il se peut qu'avec l'écriture différée, des informations non à jour soient écrites sur le disque. Ce problème peut être évité, mais il augmente la complexité du système.

Si le rapport du nombre de lectures au nombre d'écritures est élevé, il est plus efficace de mettre en œuvre la technique d'écriture simultanée tout en acceptant un trafic plus important sur le bus. En revanche, dès que le nombre d'écritures augmente, il est préférable d'utiliser l'écriture différée et d'effectuer par microprogramme la purge complète du cache avant d'y entreprendre une nouvelle opération de chargement.

Un autre critère de conception consiste à définir ce qu'il faut faire en cas d'erreur d'écriture dans le cache. Une façon de faire consiste à récrire le mot en erreur dans le cache à partir de sa version antérieure qui se trouve en mémoire. Ceci n'est possible qu'avec l'écriture immédiate. Une autre stratégie consiste à écrire directement en mémoire et à n'écrire dans le cache qu'en cas d'opération de lecture mémoire. Ces aspects concernent la

*stratégie d'écriture*; l'une et l'autre des techniques présentées ci-dessus sont d'un usage courant.

L'amélioration de la technologie de fabrication des composants permet de confectionner aujourd'hui des microprocesseurs disposant de caches intégrés très rapides. Ces caches sont de faibles capacités, étant donné surtout que l'espace réel d'intégration (la superficie) du composant est limité. Il est donc souhaitable, dans ce cas, de disposer sur une carte processeur de deux caches distincts, l'un intégré dans le microprocesseur, l'autre sur la carte processeur. De ce fait, en cas de besoin, si un mot n'est pas présent dans le cache interne, le processeur consulte le cache sur la carte processeur. En cas de nouvel échec, c'est alors la mémoire principale qui est

Tout ce qui est relatif aux caches est techniquement important dans la recherche d'amélioration des performances d'une machine. la quasi-totalité des moyens et grands ordinateurs utilisent et mettent en œuvre diverses formes de caches. On peut trouver des informations supplémentaires sur les caches dans les articles de : Hill, 1988; Przybylski et al., 1988; Short et Levy, 1988 et Smith, 1986, 1987.

## 4.6. QUELQUES EXEMPLES DE COUCHES MICROPROGRAMMÉES

Dans cette section, nous examinons les couches microprogrammées de nos deux systèmes cibles : les familles de composants d'Intel et de Motorola. Cet examen est volontairement superficiel, car en réalité ces systèmes sont très complexes. Dans les deux familles, nous analysons les éléments les plus simples en évitant les détails superflus.

### 4.6.1. La micro-architecture du 8088 d'Intel

Tous les microprocesseurs d'Intel sont élaborés à partir d'une micro-architecture assez semblable fondée sur la pionnière d'entre elles, celle du microprocesseur 8086 (McKeivitt et Bayliss, 1979). Le 8088 est virtuellement le même microprocesseur que le 8086, à l'exception de son bus externe qui est orienté 8 bits plutôt que 16 bits. Leur chemin des données est identique (l'UAL, les registres, les bus internes, etc.). Il en est de même du 80286 qui est, lui, organisé en quatre unités fonctionnelles indépendantes comme le montre la figure 3-40. Le 80386 dispose d'un chemin des données de 32 bits et de huit unités fonctionnelles; il est quelque peu différent des trois autres.

Le 8088 que nous étudions ci-dessous, même s'il est considéré comme le plus simple, présente une base de conception reposant sur des idées communes aux autres microprocesseurs. C'est sur des questions de détail que le 8088 est différent des autres. La micro-architectture du 8088 met en relation étroite les circuits logiques et les micro-instructions, dans un souci d'en minimiser la taille. La microprogrammation est verticale. Les micro-instructions comprennent des champs de plusieurs bits spécifiant de manière codée les fonctions à réaliser, plutôt que d'agir directement sur les composants du chemin des données. Le cœur du 8088 est organisé autour d'un chemin des données principal tel que nous le montrons à la figure 4-31. Il comprend deux sous-ensembles indépendants qui fonctionnent en parallèle (les parties inférieure et supérieure sur la figure).

La partie inférieure du chemin des données est semblable à celles des machines que nous avons prises comme exemples dans l'ouvrage. On y distingue une UAL qui reçoit, sur ses deux entrées, des données issues de l'un des trois registres tampons de 16 bits, TMPA, TMPC ou TMPB. Le chargement de ces registres intervient au début du cycle associé à l'UAL. En sortie, le résultat de l'opération est placé sur un bus spécifique, le bus UAL, pour être enregistré soit dans l'un des trois registres tampons A, B ou C, soit dans un registre général du chemin des données. L'UAL effectue aussi bien des opérations sur 8 bits que sur 16 bits. Les bits indicateurs de condition, en sortie de l'UAL, sont mémoireés dans un registre d'état sous contrôle du microprogramme. Les registres généraux sont AX, BX, CX, DX, SI, DI, BP et SP. Leur contenu peut être placé sur le bus UAL afin

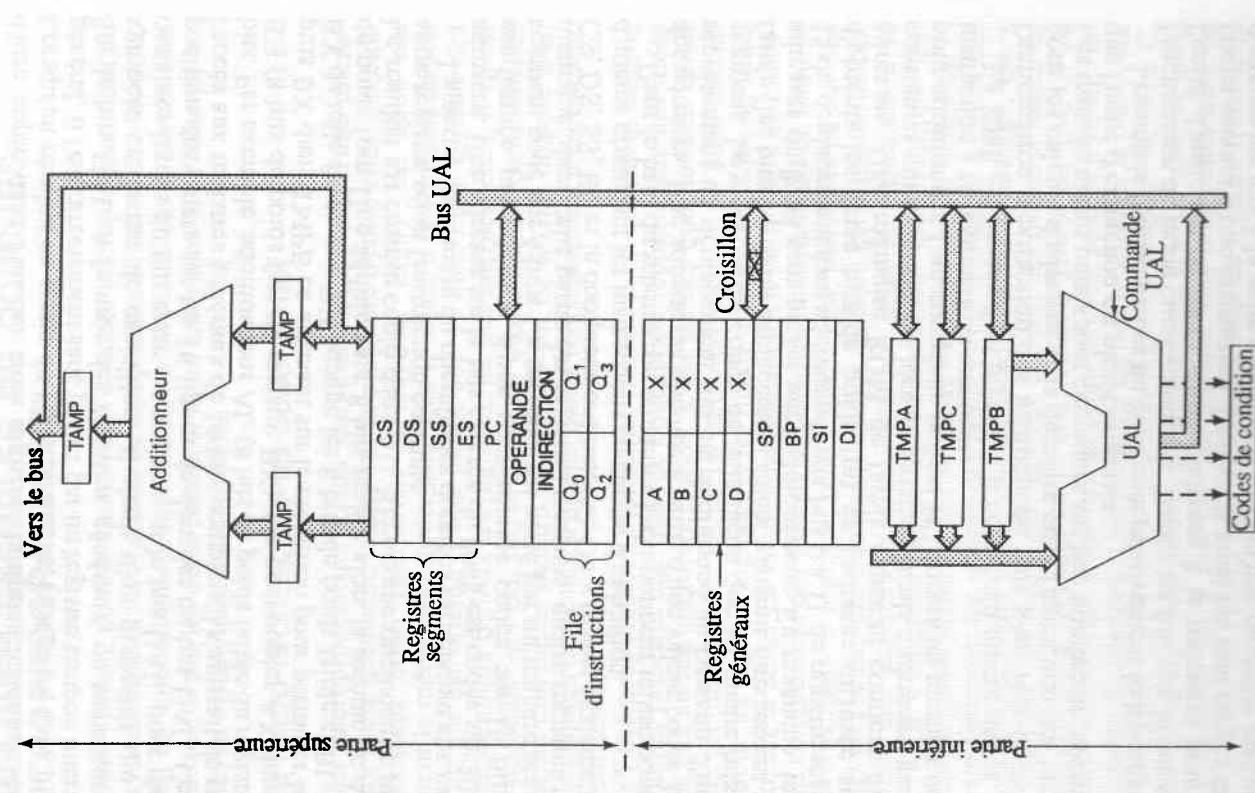
d'être copié dans l'un des trois registres tampons à l'entrée de l'UAL. Lorsqu'un résultat en sortie de l'UAL doit être enregistré dans un registre général, il l'est directement sans passer par un registre tampon intermédiaire. Sur le bus UAL, à l'entrée des registres généraux, on remarque un petit symbole en forme de croisillon. Il s'agit d'un dispositif qui permette certaines lignes du bus entre les registres et le bus UAL, pour modifier la position des octets sur les 16 lignes de données du bus UAL. Cela assure l'accès aux registres généraux de 16 bits comme s'il s'agissait de registres 8 bits. Par exemple, additionnons AL (8 bits de poids faible du registre AX) et BH (8 bits de poids fort du registre BX). On peut copier AX dans TMPA, puis BX dans TMPB en agissant sur le croisillon pour permuter les bits de BX de telle façon que BH occupe les 8 bits de poids faible et BL les 8 bits de poids fort. Une addition sur 8 bits réalise alors la somme de AL et BH. Le résultat est ensuite copié dans AL, BH ou une quelconque partie d'un autre registre selon l'instruction courante.

La partie supérieure du chemin des données est relative aux calculs des adresses. Une adresse sur le bus externe du 8088 est codée sur 20 bits; elle est formée par addition d'une sous-adresse codée sur 16 bits et d'un segment codé sur 4 bits. Ce segment est porté par un registre segment. On distingue, dans cette partie des registres tampons, quatre registres segments, CS, DS, SS, ES et le compteur ordinal, PC. On y trouve également une file d'attente contenant les instructions en attente d'exécution.

Dans le haut du schéma, on remarque l'additionneur utilisé pour le calcul des adresses. Une adresse est obtenue à partir d'une valeur codée sur 16 bits représentant un déplacement qui est additionné au contenu d'un registre segment. Le système est conçu de telle façon que les quatre bits de poids faible (les bits 0 à 3) du déplacement soient transmis directement sur le bus addresses du 8088 sans passer par l'additionneur. En revanche, les bits 4 à 15 du déplacement sont additionnés aux bits 0 à 11 du registre segment afin d'obtenir les seize bits de poids fort de l'adresse diffusée sur le bus addresses. Une mémoire ROM de faible capacité contient différentes constantes qui sont utilisées lors de certains calculs d'adresses, par exemple pour additionner 1 au compteur ordinal PC, lorsqu'un premier octet d'une instruction a été extrait de la mémoire.

Le 8088 est organisé en pipeline. Il comprend quatre unités, une pour l'anticipation d'extraction des instructions, une pour leur décodage, une pour les calculs d'adresses et la dernière pour l'exécution des instructions. Les trois dernières unités sont classiques sur bon nombre de machines, alors que l'unité d'extraction est plutôt originale.

Les unités fonctionnelles du 8088 sont relativement indépendantes, elles fonctionnent de façon quasi autonome. Toutes les fois que le bus est inactif, l'unité d'anticipation d'extraction s'adresse à la mémoire pour y querrir l'octet suivant d'une instruction. Les octets ainsi lus sont rangés dans la file d'attente les uns à la suite des autres. Lorsqu'une nouvelle instruction doit être exécutée, elle est prélevée de cette file d'attente interne.



La taille de la file d'attente est un paramètre important de conception. Si elle est trop faible, l'unité centrale ne cesse d'attendre les instructions. Si elle est trop importante ce n'est pas pour autant que ce soit plus efficace. L'unité d'anticipation d'extraction ne connaît pas la signification des octets extraits de la mémoire; elle se contente simplement d'alimenter la file d'attente tant qu'il y a de la place. En particulier, en cas de branchement conditionnel ou non, l'anticipation d'extraction des instructions continue. Si la file est trop grande, l'anticipation entraîne un gaspillage de bande passante du bus sachant qu'il faut extraire d'autres instructions de la mémoire lorsqu'un branchement intervient. La file d'attente du 8086 est de six octets, elle n'est que de quatre octets sur le 8088. Hormis la largeur du bus, ce sont les seules différences majeures entre ces deux microprocesseurs.

Pour piloter le chemin des données, le 8088 dispose d'une unité de commande microprogrammée (voir la figure 4-32). Dès qu'une nouvelle instruction de la couche traditionnelle se présente, l'octet du code opération est extrait de la file d'attente pour être chargé dans le registre RI. Un module de décodage analyse et décide les informations du registre RI pour les diffuser dans la machine en conséquence. Les registres M et N reçoivent les contenus des champs opérands de RI; ils sont utilisés dans les calculs comme adresses source et destination. Le registre X contient le code opération de l'instruction machine afin d'indiquer à l'UAL le type d'opération à réaliser. Le module de décodage élabore également, à partir du code opération de l'instruction, les commandes spécifiant si les traitements portent sur 8 ou 16 bits. Ces commandes précisent également le format des transferts (sur 8 ou 16 bits) entre les registres internes.

Le microprogramme du 8088 comprend 504 micro-instructions de 21 bits de large qui sont stockées dans une mémoire de commande de type ROM de 504 × 21 bits. Lorsqu'une instruction machine est extraite de la mémoire principale, un PLA transforme son code opération en une adresse relative à la mémoire de commande. Cette adresse correspond à celle de la première micro-instruction associée au traitement de l'instruction machine. Contrairement aux exemples du livre, dans lesquels les instructions sont examinées bit par bit, aucun microprogramme d'analyse n'est utilisé dans le 8088. En effet, pour des raisons d'efficacité et de performances, l'analyse se fait de façon câblée.

Le microprogramme est structuré en groupes ou séquences d'un nombre variable de micro-instructions (jusqu'à seize micro-instructions par séquence). Chaque séquence est associée à l'exécution d'une ou plusieurs instructions machine; elle contribue à la fourniture d'un service dans l'exécution globale des instructions, comme par exemple les calculs d'adresses. Deux types de microbranchements sont définis : un branchement local et un branchement étendu. Le microbranchement local permet d'effectuer des branchements à l'intérieur d'une séquence (codage d'adresse sur 4 bits). Un microbranchement étendu permet de réaliser un branchement quelconque dans le microprogramme. On distingue également des appels de microprocédure. Ces derniers ont un comportement semblable aux

Figure 4-31. Schéma simplifié du chemin des données du 8088

microbranchements étendus à la différence toutefois qu'ils conservent, dans le registre RS, l'adresse de retour à la séquence appellante. Les microprocédures ne sont pas imbriquées les unes dans les autres. Les 504 mots de la ROM sont organisés en quelque 90 séquences de 5 à 6 micro-instructions chacune.

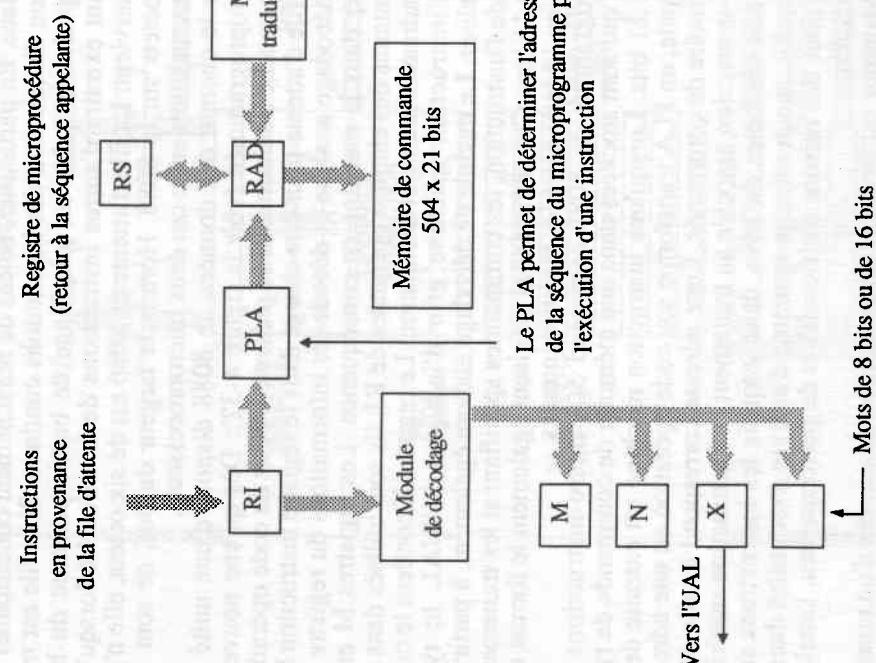


Figure 4-32. Schéma simplifié de l'unité de commande du 8088

Voyons comment, en général, les instructions arithmétiques et logiques de la couche traditionnelle sont interprétées par le microprogramme. En premier lieu l'appel d'une microprocédure permet d'effectuer les calculs d'adresses nécessaires. Les instructions utilisent les registres M et N comme champs source et destination des opérande. Quand la microprocédure de calcul d'adresses est terminée, les opérande sont dans les registres appropriés. Le registre X contient le code opération de l'instruction (son

chargement est effectué directement par un dispositif câblé). Ensuite, la séquence principale lance l'exécution d'un cycle du chemin des données avec les opérande et selon les opérations mentionnés dans les registres appropriés. Cette façon de faire permet de s'affranchir de la connaissance préalable du type d'opération à effectuer. Ainsi, les instructions machines ADD, SUB, AND, OR et quelques autres, utilisent une même séquence de microprogramme, bien que chaque instruction corresponde à un traitement différent. Les instructions PUSH, CALL et JUMP sont totalement différentes, chacune est associée à une séquence spécifique.

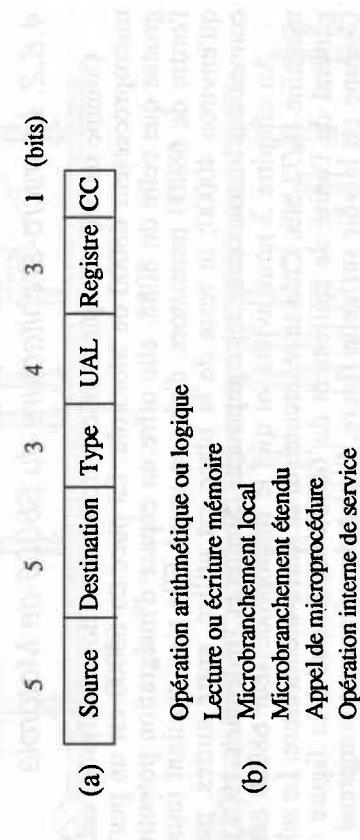


Figure 4-33. (a) Structure d'une micro-instruction de type arithmétique ou logique du 8088 (b) Les micro-instructions typiques

L'organisation typique d'une micro-instruction du 8088 est présentée à la figure 4-33(a). Elle comprend deux parties, chacune définissant des actions différentes réalisées simultanément. La partie gauche comporte deux champs de cinq bits qui permettent à la micro-instruction d'effectuer un transfert de données entre deux registres. Les trente-deux registres ainsi accessibles, comme source et destination, sont à choisir parmi les registres généraux et les registres tampons de 8 ou 16 bits. La partie droite de la micro-instruction est à encodage vertical. Elle comprend 11 bits organisés en champs dont un champ *Type* qui définit six micro-instructions listées à la figure 4-33(b). Dans le cas d'une micro-instruction faisant intervenir l'UAL, on distingue (voir la figure 4-33(b)) : le champ *UAL* (4 bits), le champ *Registre* (3 bits) et le champ *CC* (1 bit). Le champ *UAL* indique à l'UAL soit d'effectuer directement l'opération mentionnée, soit d'utiliser le code opération de l'instruction qui se trouve dans le registre *X*. Le champ *Registre* définit l'opérande et le champ *CC* précise si les bits du registre de condition doivent être ou non pris en compte. Chaque micro-instruction est complètement exécutée en un seul cycle d'horloge. Il n'est donc pas possible d'effectuer, par exemple, le transfert d'un registre général dans un registre tampon et d'utiliser ce dernier en même temps comme opérande d'entrée de l'UAL.

Les microbranchements étendus et les appels de microprocédure présentent quelques restrictions. Ils nécessitent en effet (pour référencer une quelconque adresse de destination en mémoire de commande) plus de bits que n'en contient la micro-instruction. Une solution à ce problème consiste à utiliser une mémoire ROM de traduction d'adresses qui établit une relation entre les cinq bits d'adresses générés par la micro-instruction et une zone d'adresse de destination en mémoire de commande. Cette stratégie limite le nombre des destinations possibles à des zones de 32 adresses; malgré cette restriction ce nombre est amplement suffisant.

#### 4.6.2. La micro-architecture du 68000 de Motorola

Comme second exemple de micro-architecture, nous analysons celle du microprocesseur 68000 de Motorola. La puce du 68000 est un peu plus grosse que celle du 8088, elle offre un espace d'intégration potentiel de l'ordre de 68000 transistors, d'où son nom! Elle n'en contient toutefois qu'environ 40000; le reste de la place est utilisé, entre autres, par les connexions internes entre les composants (Stritter et Tredennick, 1978).

Au chapitre 3 nous avons vu que le microprocesseur 68000 est une machine 16/32 bits. Cela se répercute sur sa micro-architecture. Le schéma général de l'unité de traitement du 68000 est montré à la figure 4-34. Comme on le voit sur cette figure, l'unité de traitement comprend trois parties, en fait trois chemins des données distincts de 16 bits de large. Chaque partie fonctionne indépendamment des autres, en parallèle. La partie gauche traite les 16 bits de poids fort (adresses hautes) et la partie centrale les 16 bits de poids faible (adresses basses) des calculs d'adresses. La partie droite traite exclusivement les données. Chaque chemin des données comprend deux bus bidirectionnels assurant des échanges de données entre les registres et l'additionneur ou l'UAL. Il permet aussi de ranger les résultats de traitement dans ces mêmes registres. Au sommet de chaque bus on remarque la présence d'un dispositif interrupteur dont la commande d'ouverture ou de fermeture est réalisée par microprogramme. L'ouverture d'un interrupteur réalise la connexion électrique des deux parties de bus qu'il sépare. Si les six interrupteurs sont ouverts simultanément, par exemple les registres de la partie gauche de l'unité de traitement peuvent être utilisés comme opérande de l'UAL de la partie droite.

Exammons les trois chemins des données du 68000, en commençant par celui de gauche. Le bloc marqué «D0-D7, Supérieur» contient les 16 bits de poids fort des huit registres généraux du chemin des données. Le bloc en dessous, marqué «A0-A7, Supérieur», contient les 16 bits de poids fort des huit registres addresses du chemin des données. Puis on distingue les 16 bits de poids fort de plusieurs registres : un registre pointeur de pile, SP, à l'usage exclusif du système d'exploitation, deux registres tampons, DT (pour les données) et AT (pour les adresses) et le compteur ordinal, PC. Chacun de ces vingt registres peut être vidé sur le bus droit ou gauche du chemin des données par action microprogrammée.

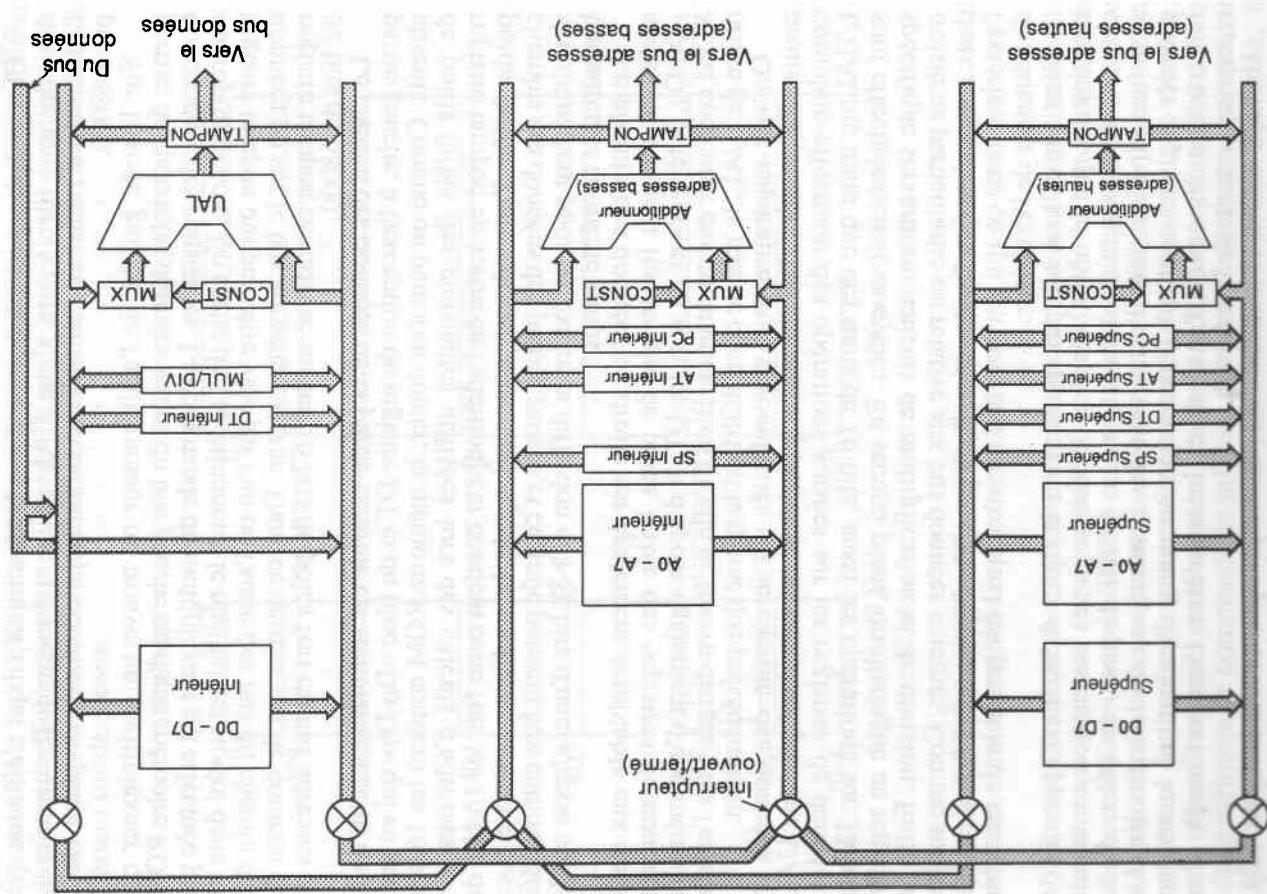


Figure 4-34. Schéma très simplifié du chemin des données du 68000

Chaque bus est relié à l'entrée d'une UAL simplifiée qui n'effectue que des additions sur 16 bits. Cette UAL utilise uniquement les circuits de l'additionneur; ainsi sa réalisation conduit à une économie de place sur la puce 68000.

Sur l'entrée gauche de l'additionneur on trouve un multiplexeur qui permet de choisir les données issues du bus gauche ou d'une mémoire ROM qui contient des constantes. La commande du multiplexeur est effectuée par microprogramme. En sortie de l'additionneur, le résultat est stocké dans un registre tampon qui peut être vidé sur l'un ou l'autre des bus du chemin des données ou sur le bus adresses externe. Dans ce dernier cas, le contenu du registre tampon est diffusé sur les 16 bits de poids fort du bus adresses de 32 bits du 68000.

Le chemin des données de la partie centrale est semblable à celui de la partie gauche, à l'exception du registre DT et du bloc «D0-D7» qui en sont absents. Comme on peut s'en douter, la mémoire ROM contient les 16 bits de poids faible des constantes utilisées lors des calculs d'adresses. Le registre tampon en sortie de l'additionneur contient cette fois les 16 bits de poids faible d'une adresse transmise sur le bus adresses externe. Les chemins des données des parties gauche et centrale peuvent être commandés simultanément pour permettre la diffusion des 32 bits d'une adresse sur le bus adresses en même temps.

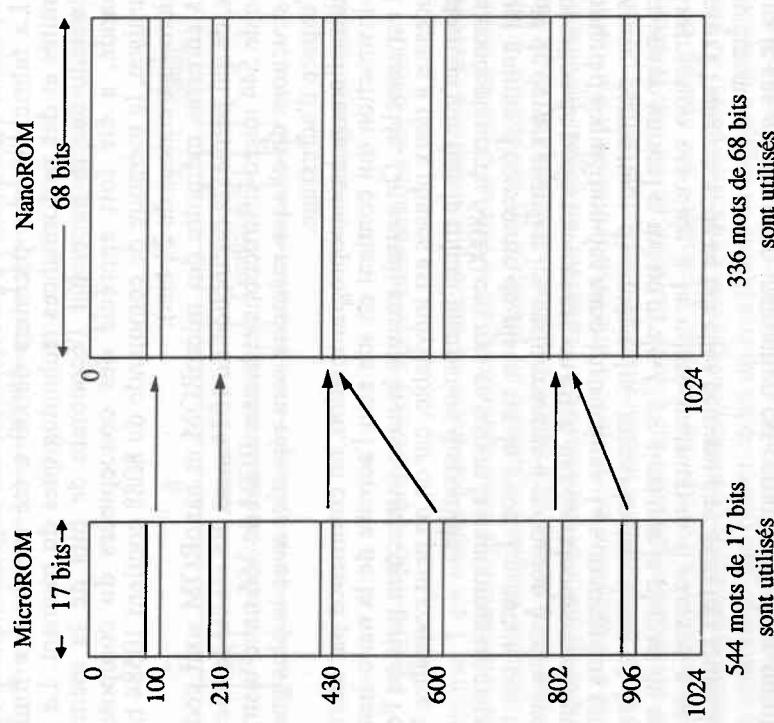
Le chemin des données de droite est également semblable aux deux autres. Il contient les 16 bits de poids faible des registres de données «D0-D7, Inférieur» et du registre DT. Il dispose également d'une mémoire ROM contenant des constantes. Bien qu'elle se trouve dirigée vers l'entrée droite de l'UAL, le rôle de cette ROM est le même que précédemment.

On voit apparaître trois particularités sur le chemin des données de droite. En premier, l'additionneur précédent est remplacé par une UAL complète effectuant des opérations variées sur les registres de données. L'UAL ne traite que des mots de 16 bits, aussi les opérations sur 32 bits sont effectuées en deux cycles. En second point, on distingue un registre spécifique servant aux calculs de multiplication et de division. Enfin la troisième particularité est relative aux bus données externes. Comme sur les deux autres chemins des données, il est possible d'émettre des données vers l'extérieur mais ce qui est nouveau ici, c'est qu'il est possible de recevoir des données de l'extérieur.

Bien que cela ne soit pas apparent sur la figure 4-34, il est opportun de mentionner que le 68000 dispose de trois registres instruction contenant chacun une instruction à exécuter. Le 68000 dispose, en effet, d'une structure pipeline pour l'extraction des instructions de la mémoire. Le premier registre contient l'instruction en cours d'exécution, le second l'instruction en cours d'analyse et le troisième l'instruction la plus récemment extraite de la mémoire.

Un autre élément du 68000 n'est pas non plus présent sur la figure 4-34, c'est un dispositif de permutation qui devrait figurer sur les bus du chemin des données de la partie droite. Comme sur le 8088 (le croisillon sur la

figure 4-31), ce dispositif assure une permutation de position des octets d'un mot de 16 bits pendant qu'il est vidé sur le bus ou qu'il est transmis du bus vers un registre. Cela permet de réaliser des transferts et des calculs portant sur des octets.



**Figure 4-35.** Le 68000 dispose d'une mémoire de commande à deux niveaux : une microROM et une nanoROM sont utilisées

Le 68000 utilise une mémoire de commande à deux niveaux, l'une pour le microprogramme (la microROM) et l'autre pour le nanoprogramme (la nanoROM ou nanomémoire) comme le montre la figure 4-35. Bien que les détails de cette organisation soient un peu différents des concepts généraux de la nanoprogrammation présentés plus avant dans ce chapitre, son objectif essentiel est le même. Il s'agit de réduire le nombre total de bits de la mémoire de commande en utilisant des micro-instructions qui ne font que référencer des séquences de nano-instructions. Ces dernières, à encodage fortement horizontal, diffusent directement vers les composants du chemin des données les signaux de commande appropriés. Le

micropogramme comprend 544 micro-instructions de 17 bits et le nanoprogramme contient 366 nano-instructions de 68 bits; cela correspond à un total de 32096 bits de mémoire de commande. Une mémoire de commande à un seul niveau, comprenant 544 micro-instructions de 68 bits, aurait nécessité 36992 bits, ce qui représente une augmentation de l'ordre de 13%. La fabrication des premiers 68000 a été menée aux limites des possibilités et des performances technologiques du moment. Le gain de place obtenu sur la puce, par l'économie de bits de la mémoire de commande, a été fort apprécié des concepteurs du composant. En comparaison, la mémoire de commande du 8088 contient 10584 bits (soit 504 micro-instructions de 21 bits).

Les adresses mémoire des microROM et nanoROM sont codées sur 10 bits, ce qui permet la distinction de 1024 mots. En réalité la microROM dispose de 544 micro-instructions et la nanoROM de 366 nano-instructions. Les instructions de chaque mémoire sont réparties avec le plus grand soin dans l'espace d'adressage.

Habituellement en nanoprogrammation, on commence par extraire une micro-instruction qui contient en son sein l'adresse de la nano-instruction qui lui est associée. On extrait ensuite la nano-instruction puis on l'exécute. Ce processus à deux phases est inévitable, car on ne peut connaître la nano-instruction qu'une fois la micro-instruction disponible.

Les concepteurs du 68000 ont mis en œuvre la nanoprogrammation dans un souci unique d'économie de place sur la puce. En aucun cas ils n'ont envisagé de devoir sacrifier les performances d'exécution à cette économie. La diminution de performances est, en effet, liée au ralentissement provoqué par le retard d'accès des nano-instructions. Les concepteurs ont utilisé une astuce permettant d'accéder aux micro-instructions et les nano-instructions en un seul et même cycle. C'est ainsi que la plupart du temps, la nano-instruction associée à la micro-instruction d'adresse *n* se trouve également à l'adresse *n*; de ce fait elles peuvent être extraites toutes les deux simultanément.

Dans le cas où une nano-instruction est commune à deux ou plusieurs micro-instructions, un élément (en fait un transistor) est supprimé du circuit de décodage d'adresse de la nanoROM afin de faire correspondre à deux ou plusieurs adresses une seule et même nano-instruction. Par exemple en supprimant un transistor, l'adresse 0000011111 peut référencer le même mot que l'adresse 0000010111; c'est ainsi que les micro-instructions d'adresses 31 et 23 utilisent la même nano-instruction d'adresse 23. Le choix précis d'implantation des micro-instructions et des nano-instructions en mémoire est assez délicat, à tel point que les concepteurs ont utilisé un ordinateur afin de les aider à assembler ce puzzle.

Le 68000 utilise deux formats de micro-instructions comme le montre la figure 4-36. Les micro-instructions ne sont pas exécutées séquentiellement comme sur le 8088. En effet, chacune d'elles réfère explicitement la micro-instruction qui la suit. Cette façon de faire résulte de la correspondance d'adresses entre les micro-instructions et les nano-instructions.

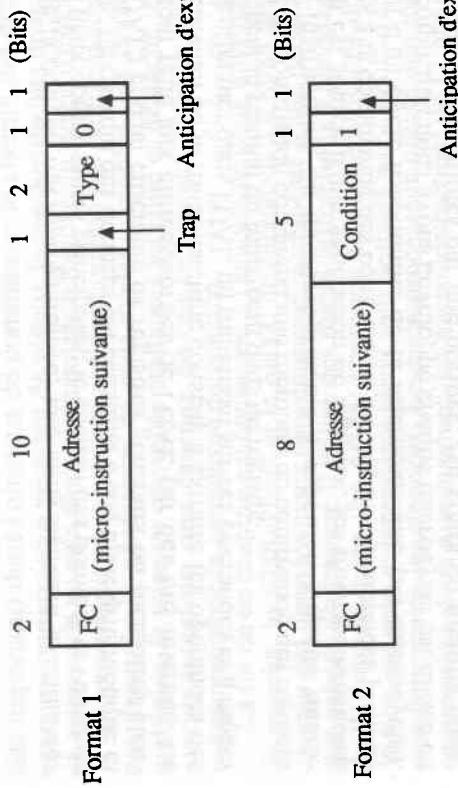


Figure 4-36. Formats des micro-instructions du 68000

Le format 1 est utilisé par toutes les micro-instructions, à l'exception de celles de branchements conditionnels. C'est le format 2 qui est utilisé dans ce cas. Chaque format comprend un champ d'un bit destiné à anticiper l'extraction de la micro-instruction suivante, un champ de deux bits, FC, utilisé pour générer et transmettre à l'extérieur un signal précisant si une référence à la mémoire est relative à une instruction ou à une donnée, s'il s'agit d'un acquittement d'interruption ou tout autre chose. Le format 1 contient divers champs dont un champ *Adresse* de 10 bits référençant la micro-instruction suivante. Le format 2 contient un champ de 5 bits précisant la condition de branchement qui doit être testée parmi 32 possibles. Cela est semblable aux bits et codes de conditions du registre instruction. Le résultat du test produit 2 bits qui, associés aux 8 bits du champ *Adresse* de la micro-instruction suivante, fournissent quatre destinations de branchements possibles.

Les 68 bits de la nano-instruction sont à encodage fortement horizontal, organisés en quelque 38 champs distincts d'un ou deux bits de large. Chaque champ est associé à une commande du chemin des données : le chargement/déchargement des registres, la sélection des registres, la commande des interrupteurs, l'activation des bits de conditions, la fonction de l'UAL, l'utilisation des constantes de la ROM, la sélection de l'entrée du multiplexeur, les opérations de lecture ou d'écriture mémoire, etc.

## 4.7. RÉSUMÉ

Vue de la couche microprogrammée, une unité centrale d'ordinateur comprend deux composantes principales: le chemin des données et l'unité de commande. Le chemin des données est constitué d'une unité arithmétique et logique (UAL) et d'un ensemble de registres généraux ou mémoires bloques, le tout relié aux entrées/sorties de l'UAL par des bus internes. Le cycle fonctionnel de la micromachine consiste à extraire les opérandes des registres, à effectuer dans l'UAL un traitement sur ces opérandes et à ranger éventuellement le résultat du traitement dans un registre.

L'unité de commande comprend une mémoire de commande qui contient les micro-instructions du microprogramme à exécuter. Chaque micro-instruction est source d'une génération de signaux, les microcommandes, nécessaires à la commande des composants du chemin des données pendant l'exécution d'un cycle fonctionnel de la micromachine (un microcycle). Dans l'exemple que nous avons développé chaque microcycle est divisé en plusieurs sous-cycles pilotés par une horloge. Au cours du premier sous-cycle, la micro-instruction est extraite de la mémoire de commande pour être chargée dans un registre particulier, le registre micro-instruction. Au cours du second sous-cycle, les opérandes sont positionnés aux entrées de l'UAL, le plus souvent dans des registres tampons. Pendant le troisième sous-cycle, l'UAL et le décaleur effectuent les traitements indiqués par la micro-instruction. Enfin, pendant le quatrième sous-cycle le résultat du traitement de l'UAL est éventuellement enregistré dans un registre, s'il en est besoin lors d'une autre micro-instruction.

Le séquencement des micro-instructions est propre à chaque machine. Certaines machines disposent d'un compteur ordinal dans la micromachine (le microcompteur ordinal). Quelquefois la micro-instruction contient elle-même l'adresse de base de la micro-instruction suivante. L'adresse définitive est alors obtenue par réalisation d'un OU logique entre l'adresse de base et certains bits d'états particuliers de la micromachine. Sur certaines machines l'adresse de base d'une micro-instruction est combinée avec des bits d'états résultant de la micro-instruction précédente pour produire une adresse de branchement qui sera utilisée à la micro-instruction suivante.

Les micro-instructions peuvent être à encodage horizontal correspondant le plus souvent à un bit par microcommande, ou à encodage vertical c'est-à-dire plusieurs champs de bits, nécessitant souvent un décodage complexe pour fournir les microcommandes ou encore à une solution intermédiaire. L'organisation horizontale conduit à des micro-instructions très larges en nombre de bits et à des micromachines à haut degré de parallélisme et performantes en vitesse d'exécution. L'organisation verticale concerne plutôt des petites machines, lentes, moins complexes et disposant de micro-instructions de moindre largeur.

Les performances de la micromachine peuvent être améliorées en utilisant diverses techniques. L'une d'entre elles, la nanoprogrammation, consiste à définir des micro-instructions qui sont en fait des pointeurs (d'un

faible nombre de bits) qui réfèrent des nano-instructions (d'un nombre de bits important) dont l'objet est de commander le chemin des données. La nanoprogrammation a pour but de réduire le nombre de bits de la mémoire de commande au prix d'une augmentation relative du temps d'exécution. D'autres démarches permettent l'amélioration des performances, comme par exemple la génération de sous-cycles variables, les branchements multiples, la mise en œuvre de structures pipeline et de cache, etc. Il faut savoir, toutefois, que chacune de ces techniques apporte une relative augmentation de la complexité du système.

La fin de ce chapitre est consacrée à l'analyse de deux microprocesseurs du marché, le 8088 d'Intel et le 68000 de Motorola. Le 8088 dispose d'un chemin des données assez conventionnel, agrémenté d'une partie spécifique aux calculs d'adresses. Il utilise des micro-instructions à encodage vertical de 21 bits de large. Le 68000 comporte trois chemins des données de 16 bits qui sont indépendants, deux sont destinés aux calculs d'adresses, le troisième traite les données. Le microprocesseur utilise une mémoire de commande à deux niveaux avec des micro-instructions de 17 bits de large et des nano-instructions à encodage horizontal de 68 bits de large.

## 4.8. EXERCICES

- Décrire des avantages et des inconvénients de la microprogrammation.
- Traduire la fonction Pascal suivante en un programme constitué d'instructions Mac-1:

```
function min(i,j,k:petit_entier): petit_entier;
var m : petit_entier;
begin
  if i<j then m:= i else m:= j;
  if k<m then m:= k;
  min:= m;
end;
```

Le type `petit_entier` est le même que celui déclaré dans le texte à la figure 4-11(a).

- Traduire l'instruction Pascal
- `if i < j then goto 100;`

```
if i < j then goto 100;
```

en un programme constitué d'instructions Mac-1; `i` et `j` sont des entiers sur 16 bits en complément à 2. Attention, lorsque `i = 32767` et `j = -10`, une soustraction entre `i` et `j` produit un débordement indétectable.

4. A la ligne L4 de la figure 4-11(b), on charge la constante K dans l'accumulateur AC. Les deux instructions précédant la ligne L4 précisent que K se trouve déjà dans AC. Pourquoi recharge-t-on K une nouvelle fois?
5. Calculer les temps d'exécution, en nombre de micro-instructions Mic-1, de chacune des macro-instructions Mac-1.
6. Refaire l'exercice précédent, mais avec cette fois des micro-instructions Mic-2.
7. Traduire l'instruction Pascal suivante,

```
n := a[i]
```

directement à l'aide de micro-instructions Mic-1 sans passer par l'étape intermédiaire des Mac-1. Expliquer votre démarche.

8. Considérer une nouvelle version des micro-instructions Mic-1, dans laquelle les champs ADDR et COND sont supprimés. On obtient alors des micro-instructions de 22 bits de large. Les branchements sont alors spécifiés par des micro-instructions distinctes. Les bits N et Z sont mémorisés dans un registre d'état et sont donc disponibles pour les tests conditionnels de branchements ultérieurs. Comparer la taille du programme original écrit avec les Mic-1, à celle du même programme écrit à l'aide des nouvelles micro-instructions.
9. Est-ce que les deux instructions Mic-1 suivantes réalisent la même fonction? Expliquer.

```
a := a+a; if n then goto 0;
a := decalq(a); if n then goto 0;
```

10. Votre unique listing d'un microprogramme écrit en Mic-1 est détruit accidentellement. Vous avez besoin de ce programme; aussi pour le reconstruire, vous devez partir des informations binaires qui sont contenues dans la mémoire de commande, soit

```
11001000000100011001000000001000
```

Récrire ce programme binaire en langage d'assemblage.

11. Combien de configurations binaires de micro-instructions pouvez-vous écrire pour l'instruction Mic-1 suivante :

```
ac := rdःa; if n then goto 100;
```

12. Imaginez-vous constructeur d'ordinateurs. Vous fabriquez une machine avec les Mic-1 comme instructions de la couche 1 et les Mac-1 pour la couche 2. Votre fournisseur de circuits intégrés vous propose une forte promotion sur des circuits UAL qui réalisent des calculs en complément à 1 plutôt qu'en complément à 2 comme c'est le cas sur la micromachine originale. Vous saisissez l'opportunité et en achetez une grosse quantité. Vous devez alors substituer cette nouvelle UAL à l'ancienne. Est-ce que le microprogramme de la figure 4-16 continue à fonctionner correctement? Bien entendu, sur la nouvelle machine les nombres négatifs sont représentés en complément à 1.
13. Supposer que les opérations de lecture et d'écriture en mémoire principale nécessitent trois micro-instructions plutôt que deux dans le cas des Mic-1; est-ce que la taille du microprogramme sera affectée par cette nouvelle opération? Si oui, pourquoi et de combien?
14. Le champ COND de la micro-instruction Mic-1 comprend 2 bits, qui déterminent quatre codes différents. Si l'on a besoin d'un nouveau code, par exemple pour tester le bit de débordement d'une nouvelle version d'UAL, quel est celui des quatre codes existants que vous choisissez de supprimer?
15. Proposer une modification simple des Mic-1 pour permettre l'appel direct de microprocédures. L'appel et le retour d'une microprocédure peuvent être réalisés de diverses façons. Pour simplifier, on ne permet pas d'appeler une microprocédure quand on est déjà dans une microprocédure. Vous pouvez modifier la micro-architecture mais essayez de minimiser les changements et les restrictions de fonctionnement.
16. Supposez que l'on vous charge de modifier la micro-instruction Mic-1 en ajoutant un bit supplémentaire au champ COND ou ADDR. Vous ne pouvez pas toucher à la largeur de la micro-instruction, donc il vous faut réduire d'un bit l'un des 11 autres champs. Quel champ diminuer sans que cela perturbe trop la fonctionnalité de la Mic-1?
17. Y a-t-il des séquences du microprogramme Mic-2 où le programmeur a dû faire des choix entre la rapidité d'exécution et la capacité mémoire, de façon différente de ce qui avait été fait dans le microprogramme Mic-1? Si oui, donner un exemple.
18. Les micro-instructions Mic-2 sont plus compactes que les Mic-1 (12 bits contre 32 bits). Peut-on envisager des Mic-3 encore plus compactes? Expliquer.

19. Le Président de la société Nano-Milli-Micro, fort impressionné par le chiffre des ventes de ses nanomémoires, envisage d'ajouter une pico-mémoire à sa gamme de produits. Dans une machine, la picomémoire est référencée par la nanomémoire comme la mémoire de commande référence la nanomémoire. Cela revient à dire que chaque nano-instruction contient l'adresse d'une pico-instruction. Vous êtes, dans l'entreprise, chargé du service «La chasse aux mauvaises idées». Que pensez-vous de la proposition du Président?
20. On désire planter le programme de la figure 4-16 dans une structure nanoprogrammée; quelle est alors la taille mémoire nécessaire, exprimée en bits (taille totale: mémoire de commande et nanomémoire)? Comparer avec la taille de la mémoire de commande dans la version originale.
21. Supposez que vous mettiez en œuvre un pipeline à  $n$  étages à la place de celui à 5 étages de l'exemple du livre. Que devient dans ce cas la pénalité du branchement?
22. Considérer une machine dans laquelle 20% des instructions sont des branchements conditionnels et 10% des boucles. Les branchements conditionnels peuvent être prévus avec une probabilité de succès de 60% et les boucles de 90%. La pénalité en cas d'échec est de quatre cycles. Il n'y a pas de pénalité pour les branchements inconditionnels, ni pour les estimations réussies. Quelle est l'efficacité d'un pipeline sur une telle machine?
23. A la figure 4-28(b), quelle est l'adresse du mot qui contient la valeur 2131?
24. Un cache offre un taux de présence de 95%, un temps d'accès de 100 ns en cas de succès et de 800 ns en cas d'échec. Quel est le temps moyen d'accès de ce cache?
25. On conçoit un cache pour un ordinateur comprenant  $2^{32}$  octets de mémoire centrale. Le cache dispose de 2048 lignes et manipule des blocs de 16 octets. Calculer, dans le cas d'un cache associatif et d'un cache direct, combien d'octets au maximum peuvent se trouver présents dans le cache.
26. Au premier abord, la présence ou l'absence de cache ne semble pas avoir grand-chose à voir avec la décision de mapper ou non les entrées/sorties en mémoire. Réfléchissez et voyez s'il y a, ou non, un problème sous-jacent.